UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

Métodos Heurísticos en Problemas Geométricos

Visibilidad, Iluminación y Vigilancia

TESIS DOCTORAL

Santiago Canales Cano

Ldo. en Ciencias Matemáticas

2004

DEPARTAMENTO DE MATEMÁTICA APLICADA

Facultad de Informática

Métodos Heurísticos en Problemas Geométricos Visibilidad, Iluminación y Vigilancia.

TESIS DOCTORAL

Presentada por

Santiago Canales Cano

Ldo. en Ciencias Matemáticas

Dirigida por

Manuel Abellanas Oar

Doctor en Ciencias Matemáticas

Gregorio Hernández Peñalver

Doctor en Ciencias Matemáticas

2004

A toda mi familia

Sólo una cosa debemos aprender y enseñar: "humildad"

Resumen

Dentro de la Geometría Computacional, uno de los campos que ha suscitado mayor interés entre la comunidad científica internacional ha sido el de la Visibilidad, es decir, el conjunto de problemas que están relacionados con los conceptos de iluminación y vigilancia de estructuras geométricas, con todas sus posibles variantes.

Los resultados obtenidos en este ámbito por los investigadores, se pueden aplicar en algunos casos para solucionar problemas industriales reales relacionados con la iluminación o vigilancia, tales como iluminación de calles o de naves comerciales. Sin embargo, en muchas otras ocasiones, los elementos físicos reales que existen en la actualidad, discrepan en algún sentido de los modelos teóricos utilizados, con lo cual los resultados obtenido no son aplicables. Por ello, es necesario utilizar definiciones de visibilidad o iluminación que se acerquen cada vez más a situaciones reales.

Siguiendo este objetivo, presentamos en la primera parte de esta memoria resultados combinatorios y algorítmicos utilizando dos definiciones de iluminación que añaden condiciones a los conceptos de vigilancia utilizados tradicionalmente: la primera de estas definiciones fue presentada por Ntafos en 1992, se denomina visibilidad de alcance limitado y añade una restricción a la distancia máxima de iluminación desde un determinado punto; la segunda que hemos denominado $t-$buena iluminación, se presenta en esta memoria por primera vez y su idea fundamental se basa en que una estructura geométrica sólo está bien iluminada si todos los puntos que la iluminan están "bien distribuidos" alrededor de ella. Respecto a la visibilidad de alcance limitado presentamos resultados combinatorios para polígonos escalera y polígonos pirámide, mientras para la $t-$buena iluminación presentamos resultados algoritmos que permiten calcular las regiones iluminadas con esta definición, por luces situadas en diferentes posiciones respecto a un polígono P.

Por otra parte existen problemas en el ámbito de la Geometría Computacional que o bien son de naturaleza $\mathcal{NP}$-dura, o bien no se han encontrado hasta el momento algoritmos eficientes que los solucionen. Sin embargo, en ambos casos, puede existir la necesidad real de aportar respuestas a dichos problemas, aunque dichas respuestas sean aproximadas o heurísticas.

Así, en la segunda parte de esta memoria, presentamos procedimientos metaheurísticos que abordan problemas con estas características. Esquemáticamente se han abordado dos problemas. El primero de ellos es el problema de minimización del número de luces que iluminan un polígono P, cuya naturaleza $\mathcal{NP}$-dura fue demostrada por Lee y Lin en 1964 y el segundo es el problema de la búsqueda de un nuevo punto en un Diagrama de Voronoi dado, tal que la región asociada a este nuevo punto tenga área máxima en el nuevo Diagrama de Voronoi construido. Para este segundo problema no se han encontrado hasta el momento soluciones algorítmicas eficientes que lo solucionen cuando los puntos se encuentran en posición general, aunque recientemene se han

presentado soluciones para puntos situados en posición convexa.

Para atacar heurísticamente el primero de estos problemas necesitamos solucionar previamente el problema de la búsqueda del conjunto de k luces, con $k \geq 1$, cuyo área conjunta iluminada en el interior de un polígono P sea máxima. Este problema que se analiza por separado para el caso $k = 1$ y $k > 1$ constituye el contenido fundamental de la segunda parte de esta memoria.

Abstract

Within the Computational Geometry, one of the areas that has raised more interest between the international scientific community has been that of the Visibility, that is to say, the set of problems that are related to the lighting concepts of the geometric structures, with all their possible varying.

The results obtained by the researchers, can be applied in some cases to solve real industrial problems related to the illumination or alertness, such as lighting of streets or of industrial zones. However, in many other occasions, the real physical elements that exist at present, in some sense disagree of the used theoretical models, therefore the obtained results are not applicable. Because of this, it is necessary to use visibility or illumination definitions that are approached increasingly to real situations.

Continuing with this objective, we present in the first part of this report results using two illumination definitions, that add conditions to the alertness concepts traditionally used: the first of these definitions was presented by Ntafos in 1992, and it is designated limited visibility and it adds a restriction to the distance maximum of illumination; the second that we have designated $t-$good illumination, is presented in this thesis for the first time and his fundamental idea is based in the fact that a geometric structure only is well lit if all the points that illuminate it are well grouped around it. With respect to the limited visibility we present combinatorial results for polygons staircase and polygons pyramid with lights located in the vertices, while for the $t-$good illumination we present algorithms to compute the regions illuminated with this definition, by lights located in different situations.

On the other hand they exist problems in Computational Geometry that either they are of nature $\mathcal{NP}$- hard, or there has not been found until the moment, efficient algorithm that solve them. However, in either case it can exist the real need of providing answers to said problems, though such solutions will be approximated or heuristic.

Thus, we present in the second part of this thesis, metaheuristics that approach problems in these conditions. The first of them is the problem of seeking the minimal number of lights that illuminate a polygon P, whose nature $\mathcal{NP}$- hard was demonstrated by Lee and Lin in 1964 and the second is the problem of the search of a new point in a Voronoi diagram, such that the region associated with this new point has area maximum in the new Voronoi diagram. For this second problem there it has not been found until the moment solutions efficient when the points are found in general position, though recently there have been presented solutions for points located in convex position.

To solve the first of these problems we need to solve previously the problem of the search of the set of k light, with $k \geq 1$, such that the illuminated area for all the lights in the interior of a polygon P is maximum. This problem that is analyzed separately for the case $k = 1$ and $k > 1$

constitutes the fundamental content of the second part of this thesis.

Agradecimientos

Son las dos de la mañana y tengo ganas de decir "*gracias a todo el mundo*" y ya está. Sin embargo, son muchas las personas que han hecho posible que esta memoria vea la luz y creo que merece la pena hacer un pequeño esfuerzo final, para que quede constancia de mi profunda gratitud hacia todos ellos. Pues nada, me pongo los cascos, enciendo la radio y vamos a por ello.

En primer lugar me gustaría mostrar mi más sincero agradecimiento a los Profesores Manuel Abellanas Oar y Gregorio Hernández Peñalver, directores de esta memoria, no sólo por el trabajo en equipo que hemos realizado durante todos estos años y el interés que han despertado en mí hacia la investigación matemática en este apasionante mundo de la Geometría Computacional, sino también por la confianza, la amistad y el afecto que me han manifestado siempre. Los consideró mis amigos y maestros.

A todos los miembros del Dpto. de Matemática Aplicada de la Facultad de Informática de la Universidad Politécnica de Madrid, por hacerme sentir en su departamento como en mi propia casa y por el café de las once que tantas veces he compartido con ellos, especialmente a la profesora Francisca Martínez por sus constantes ánimos.

Mi agradecimiento hacia el profesor Enrique Alba y todos los miembros de su equipo de trabajo, en el Dept. de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, por sus impagables ayudas en la elaboración de la segunda parte de esta memoria y por la grata acogida que he sentido durante mis estancias en dicho Departamento.

Gracias también a todos mis compañeros del Dpto. de Matemática Aplicada y Computación de la Escuela Técnica Superior de Ingeniería de la Universidad Pontificia Comillas de Madrid, y muy en especial, a los profesores Estrella Alonso, Felix Alonso, Santiago Cano, Alicia Castellano, Lucia Cerrada, María Luisa Guerrero, Carlos Gutiérrez-Cañas, Angela Jiménez, Antonio López, Isabel Paniagua, Javier Rodrigo, Loli Sotelo, Francisco Javier Rodríguez y Agustín de la Villa, por su constantes ánimos y por ocuparse de algunas de mis tareas docentes, como corregir tantos exámenes en mi nombre, cuando ha sido necesario. También me gustaría mostrar mi agradecimiento a mis compañeros de despacho, especialmente al profesor Eugenio Lázaro, por ayudarme a no desanimarme en los peores momentos.

Finalmente, pero no por ello menos importante, a toda mi familia, por la paciencia que han demostrado conmigo, especialmente a mis hermanos Manuel y Manuela, que al convivir conmigo han tenido que soportar mis angustias y mi paliza durante tantas y tantas noches y a mis padres, que me han apoyado siempre, a quienes todo se lo debo y cuyas vidas son para mí un ejemplo que nunca seré capaz de imitar bien.

Gracias.

Madrid, junio de 2004.

Índice

Introducción general

Esta memoria se ha dividido en dos partes: una primera parte que hemos llamado "*Problemas de Visibilidad*" y una segunda denominada "*Métodos heurísticos en Problemas Geométricos*". Aunque al inicio de cada una de estas partes se presenta una introducción con los antecedentes y problemas estudiados, exponemos en esta introducción general un resumen de dichos problemas y de las motivaciones de esta memoria.

El problema original de la *Galería de Arte* propuesto por V. Klee y que preguntaba cuántas luces son suficientes para iluminar todo punto interior a un polígono P con n vértices, abrió un nuevo campo en *Geometría Computacional*. En este nuevo campo se engloban todos los problemas que de alguna manera están relacionados con la *iluminación o vigilancia* [99] de cualquier estrutura o elemento geométrico. El problema propuesto por V. Klee fué solucionado por Chvátal, quien demostró que $\lfloor \frac{n}{3} \rfloor$ guardias o luces son siempre suficientes y a veces necesarias [24] para iluminar un polígono P con n vértices. Además muchas variaciones de este teorema han sido ya estudiados como podemos comprobar en [80] y [99].

Sin embargo, la mayoría de los resultados teóricos obtenidos respecto a la *iluminación o vigilancia*, son difícilmente aplicables para solucionar problemas reales, ya que las definiciones de iluminación que se utilizan se alejan de los elementos físicos de iluminación con los que se cuenta en la actualidad. Por tanto, sería deseable añadir a las definiciones clásicas de iluminación, restricciones que las hagan más reales. En este sentido, presentamos en la primera parte de esta memoria, resultados algorítmicos y combinatotios utilizando dos nuevos tipos de iluminación: *visibilidad de alcance limitado* y $t-$*buena iluminación*.

La primera de ellas fue presentada por Ntafos en 1992 en [77], donde se define el concepto de *visibilidad de alcance limitado d*: *"dos puntos son d-visibles si son visibles y su distancia es a lo más d"*, como se ilustra en la Figura 1. En el Capítulo 3, analizamos esta iluminación

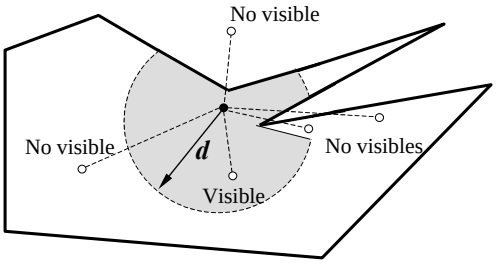


Figura 1: Visibilidad de alcance d

para dos tipos de polígonos : *pirámides y escaleras*. De forma resumida estos son los problemas

analizados:

numero de luces-vertice de alcance L que iluminan un polígono escalera
CombN-v-Es(P,L):
ENTRADA: Un polígono P de n vértices y un alcance de iluminación L.
PREGUNTA: ¿Cuál es el número mínimo k de *luces-vértice* de alcance L necesarias para iluminar el polígono P?

número de luces-vertice de alcance L que iluminan un polígono pirámide
CombN-v-Pi(P,L):
ENTRADA: Un polígono P pirámide de n vértices y un alcance de iluminación L.
PREGUNTA: ¿Cuál es el número mínimo k de *luces-vértice* de alcance L necesarias para iluminar el polígono P?

El segundo tipo de iluminación estudiado se denomina $t-$*buena iluminación* y es presentado por primera vez en esta memoria. Si tenemos un punto p en el plano euclídeo y un conjunto $F = \{f_1, ..., f_k\}$ de k focos se dice que "*un punto p está $t-$bien iluminado por F si todo semiplano con borde en p contiene al menos t focos que lo iluminan*". En la Figura 2 el punto p está $1-$*bien iluminado* por los focos f_1, f_2 y f_3, y el punto q no. En el Capítulo 4, resolvemos los

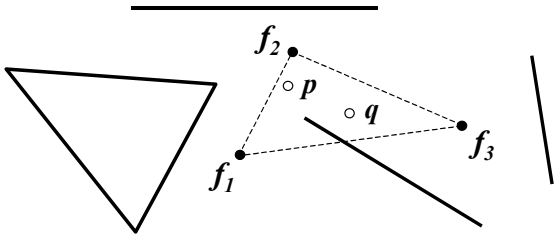


Figura 2: Un ejemplo de $1-$buena iluminación

siguientes problemas algorítmicos utilizando este tipo de iluminación:

buena $t-$iluminación en un polígono no convexo
B2$-$IPol(P):
ENTRADA: Un polígono P no convexo de n vértices.
PREGUNTA: ¿Cómo podemos calcular la *región* $2-$*bien iluminada* por n focos situados en los vértices de P?

buena $1-$iluminación para un obstáculo convexo
B1$-$ICk(C, F):
ENTRADA: Un polígono convexo C con n vértices y conjunto F de k focos exteriores a C.
PREGUNTA: ¿Cómo podemos calcular la *región* $1-$*bien iluminada* por los focos de F?

Además, utilizando técnicas ya estudiadas en *Geometría Computacional*, analizamos el problema de la $t-$*buena iluminación* cuando los focos se encuentran en los vértices de un polígono convexo, (problema B$t-$ICon(P)), y cuando los focos se encuentran en el plano en posición

general, (problema $\mathsf{Bt{-}IGenk}(F)$).

Para finalizar esta primera parte de la memoria, presentamos en el Capítulo 5 algoritmos para el cálculo de la región iluminada, (ahora sin limitación en el alcance), por un conjunto $F = \{f_1, ..., f_k\}$ de k focos interiores a un polígono P. Para este problema que enunciamos:

iluminacion múltiple en un polígono
p-Pvk(P, T):
ENTRADA: Un polígono P de n vértices y un conjunto T de k *luces-punto* interiores a él.
PREGUNTA: ¿Cómo podemos calcular la zona iluminada por los k focos a la vez?

se presentan dos algoritmos: uno incremental de complejidad $O(kn)$ y otro de doble barrido de complejidad $O(n + k\log(kn))$.

Por otra parte, existen problemas geométricos que o bien son de naturaleza $\mathcal{NP}$-dura, o bien no se han encontrado hasta el momento algoritmos óptimos que los solucionen. Para este tipo de problemas tiene sentido el estudio de técnicas heurísticas, como pueden ser *simulated annealing* [35, 47] o los *algoritmos genéticos* [29, 55, 40], para solucionarlos. Con el auge de la computación, este tipo de técnicas aproximadas está siendo motivo de atención por parte de la comunidad científica internacional. El propio Michael Atiyah hace mención en su artículo "*Polyhedra in Physics, Chemistry and Geometry",* [4], de la necesidad de la utilización de dichas técnicas númericas en el estudio de problemas geométricos.

En este sentido, en la segunda parte de esta memoria, ("*Métodos heurísticos en Problemas Geométricos"*), presentamos técnicas heurísticas para solucionar fundamentalmente dos problemas.

El primero de ellos es el problema MinN-p-Pvk(P), que consiste en minimizar el número de luces que iluminan completamente un polígono P de n vértices y que se estudia en el Capítulo 8.

minimización del número de luces punto que iluminan un polígono
MinN-p-Pvk(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cuál es el número mínimo k de *luces punto* necesarios para iluminar el polígono P?

Este problema ha sido estudiado por Lee y Lin [71] y utilizando una reducción al problema 3-Sat [45], probaron que es un problema $\mathcal{NP}$-duro. Para atacar este problema heurísticamente, necesitamos calcular el conjunto de k focos interiores a un polígono P, tal que el área iluminada por ellos sea máxima. Este problema se ha estudiado por separado para $k = 1$, (problema MaxA-p-Pv1(P) en el Capítulo 7) y $k > 1$, (problema MaxA-p-Pvk(P, k) en el Capítulo 8).

búsqueda del punto de máxima iluminación interior a un polígono
MaxA-p-Pv1(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿En qué punto interior a P debo colocar una luz para que el área iluminada por

dicha luz sea máxima?

búsqueda de los k puntos de máxima iluminación interiores a un polígono

MaxA-p-Pvk(P, k):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde debo colocar k *luces punto* en el interior del polígono P para que el área iluminada por dichas luces sea máxima?

El segundo problema tratado de forma heurística en la segunda parte de esta memoria, es un problema estudiado recientemente por Dehne y otros [30]. Supongamos que tenemos la red de metro de un ciudad, como por ejemplo Madrid, como se muestra en la Figura 3[1], y que queremos construir una nueva estación en un punto q, que debemos buscar, tal que la región del plano que contiene al conjunto de puntos más cercanos de q que de cualquier otra estación, sea lo más grande posible. Si asignamos a cada estación de metro la región del plano que contiene a los

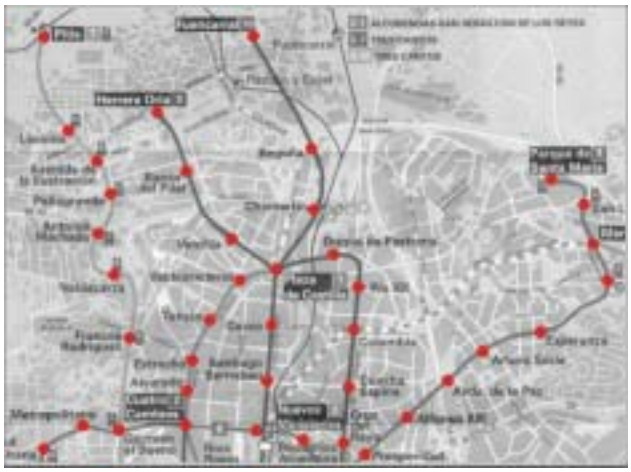


Figura 3: Vista de la red de metro de la ciudad de Madrid

puntos más cercanos a esa estación que a cualquier otra, encontramos una subdivisión de plano conocida en *Geometría Computacional* y que se denomina *Diagrama de Voronoi*, (ver Figura 4).

Por tanto, nuestro problema se puede formular en términos geométricos como la búsqueda de un nuevo punto q en un *Diagrama de Voronoi* dado, tal que la *región de Voronoi* asociada a ese punto en un nuevo diagrama, tenga área máxima.

maximización de la región de voronoi asociada a un nuevo punto

MaxA-p-Vor(N):
ENTRADA: El *diagrama de Voronoi* de un conjunto $N = \{p_1, p_2, ..., p_n\}$ de n puntos en una región R del plano.
PREGUNTA: ¿Cuál es el punto $q \in R$, tal que el área de la *región de Voronoi* $Rv(q, N \cup \{q\}, R)$, que se asocia a q en el *diagrama de Voronoi* $Vor(N \cup \{q\}, R)$ sea máxima?

Para este problema no se ha encontrado hasta el momento un algoritmo óptimo que lo

[1] Plano de la red de metro de Madrid tomada de www.dma.fi.upm/tfcs/flips/Intercambios/Intercambios.html

Figura 4: Vista de la red de metro de la ciudad de Madrid

solucione, ni se ha demostrado su naturalera $\mathcal{NP}$-dura. Dehne y otros [30], lo han estudiado para el caso concreto de que los vecinos de nuevo punto q estén en posición convexa. Presentamos en el Capítulo 9 de esta memoría dos técnicas heurísticas para abordarlo, (*simulated annealing-SA* y *random search-RS*).

A modo de resumen presentamos en las dos siguientes tablas los problemas estudiados y sus referencias en esta memoria. Mostramos en la Tabla 1 los problemas tratados en la primera parte y en la Tabla 2, los problemas para los cuales se aporta una solución heurística.

Problema Estudiado	*Posición luces*	*Solución Aportada*	*Referencia*
CombN-v-Es(P, L)	vértices de P	$\left\{ \begin{array}{ll} 1 & \text{si } L = r \\ \lfloor \frac{n}{4} \rfloor + c & \text{si } \frac{r}{2} \leq L < r \end{array} \right\}$	Teorema 3.2.5
CombN-v-Pi(P, L)	vértices de P	$\lceil \frac{n}{6} \rceil$ si $L \geq r$	Proposición 3.3.4
B$t-$IGenk(F)	general	$O(k^2)$	Sección 4.2
B$t-$ICon(P)	convexa	$\Theta(n)$	Teorema 4.3.4
B2$-$IPol(P)	vértices polígono P	$O(n^2)$	Teorema 4.3.5
B1$-$ICk(C, F)	exteriores convexo C	$O(k(\log n + \log k) + n)$	Teorema 4.4.2
p-Pvk(P)	interior polígono P	$O(kn)$	Sección 5.3
p-Pvk(P)	interior polígono P	$O(n + k\log(kn))$ / $O(k + n)$	Teorema 5.4.10

Tabla 1: Problemas estudiados en la primera parte de la memoria

Incluimos en la Tabla 2 las técnicas heurística aportadas para cada problema, así como la

sección en la que se encuentra su descripción.

Problema Estudiado	*Posición luces/puntos*	*Solución Aportada*	*Referencia*
MaxA-p-Pv1(P)	interiores polígono P	Aprox. *Simulated Annealing – SA*	Sección 7.2
		Aprox. *Algoritmos Genéticos – GA*	Sección 7.3
		Aprox. *Random Search – RS*	Sección 7.4
		Aprox. *Gradiente – GRAD*	Sección 7.5
		$\approx O(n^4)$	Sección 7.8
MaxA-p-Pvk(P,k)	interiores polígono P	Aprox. *Simulated Annealing – SA*	Sección 8.3
		Aprox. *Algoritmos Genéticos – GA*	Sección 8.4
		Aprox. *Random Search – RS*	Sección 8.5
MinN-p-Pvk(P)	interiores polígono P	Estrat. *Secuencial*	Sección 8.7.1
		Estrat. *Binaria*	Sección 8.7.1
MaxA-p-Vor(N)	interiores región R	Aprox. *Simulated Annealing-SA*	Sección 9.2
		Aprox. *Random Search-RS*	Sección 9.3

Tabla 2: Problemas estudiados en la segunda parte de la memoria

El estudio de las técnicas diseñadas en la segunda parte de esta memoria, precisa de un conjunto amplio de datos de entrada generados aleatoriamente, que permitan establecer comparativas entre todas las heurísticas. Por ello se ha implementado un *generador aleatorio de polígonos* que denotaremos con *RPG*, cuyos detalles se encuentran en el Apéndice A y cuyo código comentado se muestra en el Apéndice B. *RPG* está basado en el estudio sobre generación aleatoria de polígonos realizado por Thomas Auer y Martin Held en [6]. Igualmente para obtener conclusiones del problema MaxA-p-Vor(N), se ha implementado un algoritmo incremental que construye el *diagrama de Voronoi* de una nube de puntos generada aleatoriamente. Los detalles, para este segundo caso, se encuentran en el Capítulo 9.

Capítulo 1

Complejidad, problemas $\mathcal{NP}$-duros y algoritmos aproximados

---

Esta memoria se divide en dos partes: una primera parte, en la que se presentan los resultados teóricos obtenidos, para solucionar distintos problemas geométricos relacionados de alguna manera con la *iluminación* o *vigilancia* de polígonos, y una segunda parte en la que se presentan resultados *heurísticos* o *aproximados* para problemas geométricos, para los que o bien se sabe que son de naturaleza $\mathcal{NP}$*-dura*, o bien, no se conoce hasta el momento ninguna solución algorítmica óptima de dicho problema.

De este modo, presentamos en este pimer capítulo una breve introducción a los conceptos generales de *complejidad computacional* y las *técnicas heurísticas* fundamentales, que serán utilizadas en la segunda parte de esta memoria. En este sentido, toda la información detallada de este Capítulo 1 se debe considerar como un resumen de los diferentes temas, que de alguna manera están relacionados con esta memoria y cuya información ha sido obtenida de diferentes referencias bibliográficas, (que se detallan en cada momento), relacionadas con ellos.

1.1 Primeros Conceptos y Resultados

La teoría de la complejidad algorítmica se desarrolla para justificar la dificultad en encontrar la solución de determinados problemas matemáticos. Según se explica en el libro de Garey y Johnsom [45], cuando a alguien le plantean un problema y no lo resuelve, no es conveniente responderle:

" No he encontrado un procedimiento de solución, no tengo ni idea sobre el método a seguir".

mejor sería responderle:

"No he encontrado la solución, pero es imposible resolver este problema".

Al analizar la complejidad algorítmica de un problema, se justificará la dificultad intrínseca del problema. Como se verá más adelante, no se puede hacer una afirmación tan tajante como la respuesta última. El tipo de respuesta que se ofrecerá será la siguiente:

"No he encontrado un procedimiento de solución para el problema planteado, pero nadie, entre los numerosos especialistas que estudian este problema y otros problemas equivalentes en dificultad, lo ha conseguido".

Después de que la teoría de la computabilidad fuera desarrollada, era natural preguntarse acerca de la relativa dificultad computacional de las funciones computables. Este es el objetivo de la parte de las Ciencias de la Computación que se conoce como Complejidad Algorítmica o Computacional. Rabin, (1960), fué uno de los primeros en plantear esta cuestión general explícitamente: ¿Qué quiere decir que **f** sea más difícil de computar que **g**? Rabin sugirió una axiomática que fué la base para el desarrollo de la teoría de la complejidad abstracta de Blum y otros.

Una segunda aportación ,(1965), que tuvo una relevante influencia en el desarrollo posterior de esta materia fué el artículo de J.Hartmanis and R.R. Stearms, *On the computational complexity of algorithms.* Este trabajo fue ampliamente leído y puso nombre a este cuerpo de conocimiento. En él se introduce la noción fundamental de medida de complejidad definida como el tiempo de computación sobre una máquina de Turing multicinta. El artículo también pone de relieve una cuestión intrínseca que todavía permanece abierta. ¿Se puede computar cualquier número irracional algebraico, (por ejemplo $\sqrt{2}$), en tiempo real?, esto es, ¿existe una máquina de Turing que imprima la expresión decimal del número a una velocidad de un dígito cada 100 pasos por ejemplo?

Un tercer hito en los comienzos del tema, (1965), fué el trabajo de A. Cobham, *The intrinsic computational difficulty of funcions.* Cobham enfatizó el término "intrínseco", es decir, él estaba interesado en una teoría independiente de las máquinas. Se preguntó si la multiplicación es más compleja que la suma, y consideró que la pregunta no podría ser contestada hasta que no se desarrollara própiamente la teoría. Este autor también definió y caracterizó la importante clase de funciones llamadas **£**: las funciones de números naturales que son computables en tiempo acotado por un polinomio cuyo argumento es la longitud decimal de la entrada.

Otros tres artículos que tuvieron una gran influencia en el posterior desarrollo de la complejidad algoritmica, son los de Yamada (1962), Bennett (1962), y Ritchie (1963). Es interesante hacer notar que Rabin, Stearms, Bennett y Richie fueron todos estudiantes de Princeton y prácticamente al mismo tiempo.

Varios de los autores pioneros en el tema se cuestionaron cual debía ser la medida de complejidad. La mayoría mencionaron el tiempo o el espacio como elecciones obvias, pero no estaban convencidos de que fueran las únicas posibles. Por ejemplo, Cobham sugiere que alguna medida relativa a la noción física de trabajo produciría el mejor análisis. Rabin introdujo los axiomas que una medida de complejidad debería satisfacer. Con las perspectiva de 25 años de experiencia, se puede decir que el tiempo y el espacio, (especialmente el tiempo), están ciertamente entre las más importantes medidas de complejidad. Parece ser que el primer mérito a la hora de evaluar la eficacia de un algoritmo es su tiempo de ejecución; sin embargo, recientemente se está empezando a considerar el tiempo en paralelo y el tamaño del hardware como importantes medidas de complejidad.

Otra importante medida de complejidad que podemos atribuir, (de alguna forma al menos), a Shannon (1949) es la complejidad de circuitos Booleanos, (o complejidad combinatorial). Para esta medida es conveniente asumir que la función **f** en cuestión transforma cadenas finitas de

bits en cadenas finitas de bits , y la complejidad $C(n)$ de **f** es el tamaño del menor circuito Booleano que calcula **f** para todas las entradas de longitud **n**. Esta medida está muy relacionada con el tiempo de computación, (ver [84]), y tiene una teoría propia bien desarrollada, (ver [89]).

Otra importante cuestión puesta de relieve por Cobham es definir con precisión lo que constituye un paso de computación. Esto nos lleva a la cuestión de cual es el modelo de computación adecuado para medir el tiempo de ejecución de un algoritmo. Las máquinas de Turing multicinta son las habitualmente utilizadas en la literatura, pero tienen restricciones artificiales desde el punto de vista de la implementación eficiente de algoritmos. Por ejemplo, no hay ninguna razón convincente de por qué los datos deben almacenarse en cintas lineales. ¿Por qué no almacenarlos en árboles?¿Por qué no se permite una memoria de acceso aleatorio?

De hecho, bastantes modelos de computación han sido propuestos desde 1960. Dado que los ordenadores tienen memoria de acceso aleatorio, parece natural permitir esto en el modelo, pero cómo hacerlo se ha convertido en una cuestión difícil. Si la máquina puede almacenar enteros en un solo paso, entonces alguna cota sobre su tamaño debe ponerse. (Si el número se duplica 100 veces, el resultado tendrá 2^{100} bits, y no podrá ser almacenado en ningún modelo de memoria existente. Cook, (1972), propone el uso de la máquinas de acceso aleatorio con coste de almacenaje. En este modelo se asigna un coste, (número de etapas), de aproximadamente $\log |x|$ cada vez que un número x se almacena. Esta es una hipótesis de trabajo que funciona, pero no es completamente convincente. Más popular es el modelo de máquinas de acceso aleatorio utilizadas por Aho, Hopcroft y Ullman en 1974, en las que cada operación envolviendo un entero tiene coste único, pero no se permiten enteros excesivamente grandes, (por ejemplo, su magnitud debe estar acotada por algún polinomio fijo sobre el tamaño de la entrada). Probablemente el modelo mas satisfactorio desde el punto de vista matemático es la máquina de modificación de memoria de Schönhage (1980), la cual puede interpretarse como una máquina de Turing que construye su propia estructura de memoria, o bien como una máquina de acceso aleatorio de coste unitario, que solo puede copiar, sumar o restar uno, o guardar o recuperar en un solo paso. La máquina de Schönhage, que es una generalización de la máquina de Kolmogorov-Uspenski (1959), representa la máquina más general que puede construirse haciendo una cantidad acotada de trabajo en un paso.

Volviendo a la cuestión de Cobham sobre que es un paso, lo que ha quedado claro en los último 25 años es que no hay ninguna respuesta clara. Afortunadamente, los distintos modelos de computación no son demasiados diferentes en cuanto a tiempo de computación. En general, cada uno puede simular a los otros en, a lo más, un tiempo de computación cuadrático. Entre los modelos más populares de acceso aleatorio, solo hay un factor logarítmico del tiempo de computación.

Esto nos conduce al último concepto importante desarrollado en 1965, la identificación de la clase de problemas que se pueden resolver en tiempo acotado por un polinomio sobre la longitud de la entrada. La distinción entre algoritmos de tiempo polinomial y algorítmos de tiempo exponencial fué hecha por primera vez en 1953 por Von Neumann. Sin embargo, la clase no fue definida formalmente ni estudiada hasta que Cobham introduce la clase de funciones O en 1964. Cobham apuntó que la clase estaba bien definida, independientemente del computador elegido, y dió una caracterización en el marco de las funciones recursivas. La idea de que los problemas computables en tiempo polinomial se corresponden con los problemas tratables fue expresada

por primera vez por Edmonds 1965, quien llamó a los algoritmos en tiempo polinomial "buenos algoritmos". La notación, ahora estandar, de $\mathcal{P}$ para la clase de los lenguajes reconocibles en tiempo polinomial fue introducida posteriormente por Karp 1972.

La identificación de $\mathcal{P}$ con los problemas tratables ha sido aceptada generalmente en el área desde los años 70. No es obvio el por qué esto debería ser cierto, pues un algoritmo cuyo tiempo de ejecución sea n^{1000} no es eficaz, y por contra, un algoritmo con tiempo de ejecución $2^{0.0001n}$ si que es eficaz en la práctica. Sin embargo, parece ser un hecho empírico que no existen problemas con este tiempo de ejecución tan extremos, (ver [45]). El ejemplo más significativo de un algoritmo en tiempo exponencial es el algoritmo del simplex de la programación lineal. Smale (1982) intenta justificar esto demostrando que el tiempo de ejecución en media es rápido, pero es importante hacer notar que Khachian (1979) ha demostrado que la programación lineal está en $\mathcal{P}$ usando otro algoritmo. Así , la tesis general de que estar en $\mathcal{P}$ equivale a ser un problema tratable, no se viola.

En este sentido a partir del análisis de complejidad se han definido varias clases de problemas: los problemas que se resuelven en tiempo polinomial por una máquina determinista forman la clase $\mathcal{P}$ y los problemas que se resuelven en tiempo polinomial por una máquina no determinista forman la clase $\mathcal{NP}$.

La clase $\mathcal{NP}$-completo es aquella formada por los problemas más difíciles dentro de la clase $\mathcal{NP}$, y para probar si un problema pertenece a esa clase se requiere que algún problema $\mathcal{NP}$-completo pueda transformarse a él. Lo anterior, produce la importancia que tiene en computación el problema de Satisfactibilidad (**SAT**), ya que en 1971, Cook demostró que este problema era $\mathcal{NP}$-*completo*, y que sirve como base para demostrar que otros problemas son $\mathcal{NP}$-*completo*.

1.2 Complejidad Computacional

Dentro de la evolución de la teoría de complejidad, se han encontrado problemas con características similares, que pueden ser agrupados en categorías para su estudio.

Los *problemas de optimización* son aquellos en los cuales se busca minimizar o maximizar, (es decir optimizar), el valor de una solución en un grupo de soluciones generadas para una entrada específica, (instancia). Los *problemas de decisión* son aquellos donde se busca una respuesta de "si" o "no". Cualquier problema de optimización puede ser manejado como un problema de decisión incluyendo un valor objetivo K para la instancia n. A continuación formalizamos estos conceptos:

Definición 1.2.1 *Un problema de decisión π es un problema cuya respuesta sólo puede ser afirmativa o negativa.*

Cualquier problema de optimización se puede traducir en un problema de decisión sin más que acotar la función objetivo para preguntar si existe una solución que mejora dicha cota o no.

Aunque puede parecer más restrictivo un problema de decisión que el correspondiente de optimización, eligiendo una sucesión de problemas de decisión se puede resolver el problema de optimización. En cualquier caso, una vez analizada la complejidad de los problemas de decisión, se volverá al análisis del problema de optimización. El problema de decisión π se plantea a

partir de un ejemplo genérico, donde se introducen los parámetros del problema, y una cuestión planteada sobre el ejemplo genérico y cuya respuesta es afirmativa o negativa.

Al conjunto de todos los ejemplos del problema π se le denomina D_π. Dado un ejemplo $I \in D_\pi$, si la respuesta al problema π es afirmativa, se dice entonces que $I \in Y_\pi$. En teoría de complejidad, se trabaja con problemas de decisión, ya que un problema de optimización puede ser reducido a un problema de decisión.

En computación, al hablar de complejidad, no se está refiriendo a la dificultad que se tendría para diseñar un programa, o a lo rebuscado de un algoritmo. La teoría de la complejidad tiene que ver con dos medidas: tiempo y espacio. La complejidad computacional de un problema es una medida de los recursos computacionales, (generalmente el tiempo) requeridos para resolver el problema.

La *complejidad temporal* tiene que ver con el tiempo que tarda un programa para ejecutarse. La *complejidad espacial* estudia la cantidad de espacio de almacenamiento que es necesario para una operación. La gran mayoría de estudios de complejidad están orientados hacia el desarrollo de los algorítmos en función al tiempo, por lo que de aquí en adelante, al hablar de complejidad, se asumirá que es en el tiempo.

Es evidente que al medir el tiempo que tarda un programa en ejecutarse, no sería informativo decir que tarda cinco segundos en una máquina de 166 MHz y que tarda menos en una más rápida. Para que sea relevante el definir la complejidad temporal de un algoritmo, ésta es expresada no en términos de la máquina, sino en un interesante parámetro: el *tamaño de la entrada*.

Al analizar la complejidad de un algoritmo, el tiempo está expresado en término de pasos de computación elementales, (asignaciones, comparaciones, multiplicaciones, etc.), por ejemplo, una operación de asignación ocupa una unidad de tiempo para ejecutarse, un ciclo ocupa el número de iteraciones en que está definido, etc.

1.2.1 Función de complejidad de un algoritmo

Obviamente, el tiempo de cálculo de un algoritmo dependerá de la *dimensión del ejemplo* elegido, por lo que la medida de la eficacia deberá ser una función de la dimensión del ejemplo.

La dimensión de un ejemplo depende de la memoria necesaria para almacenarlo. Evidentemente, el número de símbolos dependen del esquema de codificación. No obstante, cualquier esquema de codificación *razonable* es válido y no afectará a la complejidad algorítmica. Se entiende por esquema de codificación *razonable* cualquiera que no incluya implícitamente un conjunto de cálculos para determinar la solución del problema.

Considerando esta restricción natural, y con el fin de evitar la dependencia explícita del esquema de codificación, se suele especificar la dimensión en función de alguno de los parámetros que identifican el ejemplo.

Con el fin de valorar los algoritmos de forma independiente al soporte utilizado para realizar los cálculos, en lugar de utilizar el tiempo como medida se utiliza el número de **operaciones básicas**. Por operaciones básicas se entiende la suma, multiplicación, asignación, comparación, etc. En la práctica se observa que un algoritmo ocupa diferente tiempo de ejecución para entradas del mismo tamaño, pero con diferentes datos. Por ejemplo, un método de ordenación de valores

puede tardar menos tiempo si su entrada está ordenada, o tardar mucho mayor tiempo incluso con los mismos valores, pero presentados en desorden, es decir, con el mismo tamaño de entrada tiene diferente comportamiento. Por lo anterior, se adopta el criterio de tomar siempre, como base de análisis de la complejidad de un algoritmo, el caso en el cual consuma el mayor tiempo de ejecución, es decir, el **peor caso.** La acotación de la definición de complejidad considera esta posibilidad al suponer siempre el "peor de los ejemplos". Se dice así que la complejidad es un criterio pesimista y en los libros ingleses llaman al análisis de los problemas basado en ella "worst case analysis".

Definición 1.2.2 *Un algoritmo tiene función de complejidad $f(n)$ cuando el número de operaciones básicas que necesita para resolver cualquier ejemplo con dimensión n está acotado por el valor $f(n)$.*

Así en el siguiente ejemplo podemos observar que la función de complejidad del algoritmo que se define es $f(n) = 4n + 3$.

Ejemplo 1.2.3 *Estudiemos la complejidad del siguiente segmento de código:*

```
[1]  suma = 0;
[2]  for(i = 1; i <= n; i++)
[3]    suma+ = i;
```

La instrucción 1 *ocupa una unidad de tiempo. La instrucción* 3 *ocupa dos unidades de tiempo, una para la suma y otra para la asignación, y es ejecutada n veces, por lo que ocupa $2n$ unidades de tiempo. La instrucción* 2 *tiene involucrada una asignación que utiliza una unidad de tiempo, $n+1$ comparaciones y n incrementos, por lo que ocupa $n+2$ unidades de tiempo. Así, el tiempo total del algoritmo es $f(n) = 1 + 2n + 2 + 2n = 4n + 3$.*

Con el fin de simplificar los cálculos y considerando que la complejidad de un algoritmo debe valorar su comportamiento con ejemplos límite, cuando la dimensión del ejemplo se hace suficientemente grande, se intoduce la **complejidad algorítmica** de un algoritmo considerando el grado mayor del polinomio *f(n)* y evitando las constante.

Definición 1.2.4 *Una función de complejidad es del tipo $O(g(n))$ cuando existe una constante c, independiente de la dimensión n, tal que:*

$$|f(n)| \leq c\,|g(n)| \quad \forall n \geq n_0 \tag{1.2.1}$$

para un n_0 fijo.

La constante c así introducida también se puede interpretar para eliminar la dependencia del tipo de ordenador que procesa las operaciones del algoritmo, ya que un ordenador potente hará todas las operaciones básicas más rápidamente que otro menos potente, y se supone que la rapidez guarda la misma proporción en todas ellas. Veamos algunos ejemplos que clarifiquen las definiciones anteriores.

Ejemplo 1.2.5 *El orden de un algoritmo cuyo tiempo de ejecucución está dado por* $f(n) = 3n^3 + 2n + 6$ *es* $O(f(n)) = O(n^3)$.

Existen algunas reglas que son útiles para determinar el orden de un algoritmo:

- **Regla 1 (Ciclos For):** El tiempo de ejecución de un ciclo **For** es al menos el tiempo de ejecución de las instrucciones dentro de él multiplicado por el número de iteraciones.
- **Regla 2 (Ciclos For anidados):** Se analiza desde dentro hacia afuera. El tiempo total de una instrucción dentro de un conjunto de ciclos anidados es igual al tiempo de ejecución de las instrucciones internas multiplicado por el producto del tamaño de los ciclos.
- **Regla 3 (Condicional):** El tiempo de ejecución nunca es mayor que el tiempo de ejecución de la condicional más el mayor de los tiempos de ejecución de las alternativas.

Ejemplo 1.2.6 *El orden del siguiente algoritmo es* $O(n^2)$.

```
[1]  if(i == 1)
[2]    for(i = 1; i <= N; i++)
[3]      suma+ = i;
[4]  else
[5]    for(i = 1; i <= N; i++)
[6]      for(j = 1; j <= N; j++)
[7]        suma+ = i;
```

Debido a que el tiempo de ejecución de la condición es de orden $O(1)$ *más el mayor tiempo de ejecución de las alternativas: cuando se cumple la condición es* $O(n)$ *y cuando no se cumple es* $O(n^2)$*, por lo cual el orden es* $O(1)+O(n^2)$*, pero como ya se ha dicho, solo se toma el mayor, quedando el orden del programa en* $O(n^2)$.

A los algoritmos de orden n se les llama de orden lineal, los de n^2 de orden cuadrático, etc.

Una alternativa a la complejidad, (y se suele hacer paralelamente a ésta), es el denominado *análisis medio* que consiste en evaluar el comportamiento real observado a partir de una serie de ejemplos de prueba de distintas dimensiones.

Con este criterio se compara el tiempo que tarda el algoritmo en cuestión en resolver diversos ejemplos de variadas dimensiones. Al trabajar con tiempos, hay que especificar el tipo de ordenador que realiza los cálculos y el procedimiento de generación de los ejemplos analizados.

No siempre son coincidentes ambos análisis; así, por ejemplo, el algoritmo del simplex tiene una complejidad muy mala, (es una función exponencial de la dimensión del ejemplo), mientras que su comportamiento medio es muy bueno.

Comparando ambos criterios, que son complementarios, el estudio de la complejidad algorítmica se justifica por tres razones:

1. Representa una cota superior del número de operaciones para cualquier ejemplo del problema.

2. No depende de una distribución de probabilidad sobre los ejemplos de un problema.

3. Se puede evaluar directamente.

1.2.2 Clasificación de los algoritmos por su complejidad. Clase $\mathcal{P}$

Según el tipo de la función de complejidad, los algoritmos pueden ser clasificados:

Definición 1.2.7 *Un algoritmo se dice polinomial si la complejidad es una función polinomial en la dimensión del problema.*

Se puede decir que un algoritmo tiene complejidad polinomial , o se ejecuta en *tiempo polinomial*, si tiene un orden $O(n^x)$. Estos algoritmos se dice que son algoritmos eficientes y los problemas que se resuelven con estos algoritmos se dice que son problemas tratables. Ejemplos de algoritmos polinomiales son el algoritmo de multiplicación de matrices o el método de la burbuja para ordenar números.

El tiempo de computación de otros algoritmos, sin embargo, crece exponencialmente respecto al tamaño de los jemplos. Es el caso de algunos algoritmos que tienen una complejidad exponencial.

Definición 1.2.8 *Un algoritmo se dice exponencial si la complejidad es una función exponencial en la dimensión del problema, es decir, existen constantes c_1, $c_2 > 0; d_1$, $d_2 > 1$ tales que a partir de un $n \geq n_0$:*

$$c_1 d_1^n \leq f(n) \leq c_2 d_2^n \quad \forall n \geq n_0 \tag{1.2.2}$$

Los problemas que se resuelven usando algoritmos de *tiempo exponencial* se conocen como problemas intratables y a los algoritmos como algoritmos ineficientes.

Ejemplo 1.2.9 *El problema de las Torres de Hanoi tiene complejidad de orden 2^n. El algoritmo de las Torres de Hanoi es el siguiente:*

```
[1]  void Hanoi(int n, int A, int B, int C)
[2]    {if(n == 1)
[3]       MueveAnillo(A, B);
[4]     else{
[5]    Hanoi(n − 1, A, C, B);
[6]    MueveAnillo(A, B);
[7]    Hanoi(n − 1, C, B, A); }
[8]    }
```

Debido a que el algoritmo es recursivo, el tiempo de ejecución puede ser descrito por la función $f(n) = 2f(n-1)+$constante. De esta forma se observa que al aumentar el tamaño de la entrada, el tiempo se duplica, (factor de dos), por lo que es fácil deducir que el orden del algoritmo es $O(2^n)$.

Aunque la mayor parte de los algoritmos son polinomiales o exponenciales hay, sin embargo, otras posibilidades; por ejemplo, cuando la función de complejidad es del tipo

$$g(n) = n^{\log(n)} \tag{1.2.3}$$

que crece más rápidamente que cualquier función polinomial y más lentamente que una exponencial. Por otra parte, hay funciones con crecimiento mayor que el exponencial; sería el caso de la función de complejidad.

$$f(n) = n^{n^n} \tag{1.2.4}$$

Para comprender la importancia que tiene la condición de polinomial para un algoritmo, se detalla a continuación en la Tabla 1.1, para distintos valores de n, la dimensión del ejemplo, el tiempo en ejecutar los algoritmos a partir de una función de complejidad polinomial (n, n^2, n^3 y n^2) o no polinomial (2^n y 3^n). Se ha supuesto que el ordenador es capaz de realizar 10^6 operaciones por segundo.

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	.00001	.00002	.00003	.00004	.00005	.00006
n^2	.0001	.0004	.0009	.0016	.0025	.0036
n^3	.001	.008	.027	.064	.125	.216
n^5	.3	3.2	24.3	1.7 min	5.2 min	13 min
2^n	.001	1.0	17.9 min	12.7 dias	35.7 años	366 sig.
3^n	.059	58 min	6.5 años	3855 sig	2×10^8 sig.	1.3×10^{13} sig.

Tabla 1.1: Complejidad de un algoritmo

Por tanto, podemos definir formalmente los problemas de decisión clase $\mathcal{P}$

Definición 1.2.10 *Un problema de decisión es de la clase $\mathcal{P}$ cuando existe un algoritmo polinomial que lo resuelva.*

El hecho de que haya problemas para los que no se ha encontrado un algoritmo polinomial induce a pensar que hay problemas en el complementario de $\mathcal{P}$. La cuestión fundamental es si esta última afirmación es cierta; parece razonable pensar que sí.

Ejemplo 1.2.11 *El problema de encontrar un camino en un digrafo es de clase $\mathcal{P}$. La entrada del algoritmo es un digrafo $G = (V, E)$ y dos subconjuntos S y T de G. La salida del algoritmo es un camino de G de un nodo en S a un nodo en T, si el camino existe.*

1.3 Algoritmo no determinístico polinomial

Para ilustrar el concepto de algoritmo no determinístico, nos centraremos en el problema de decisión del viajante: Nadie ha encontrado un algoritmo polinomial que lo resuelva, pero lo que sí es cierto es que, dado un recorrido, en tiempo polinomial se puede decidir si la respuesta al problema es afirmativa o negativa.

Definición 1.3.1 *Dado un problema de decisión π, se define un algoritmo no determinístico como aquel procedimiento definido por dos etapas:*

$\triangledown$ **Etapa de Conjetura***: En esta etapa se propone para cada ejemplo $I \in D_\pi$ una estructura S, (solución factible del problema).*

$\triangledown$ **Etapa de Comprobación***: A partir del ejemplo I y de la estructura S propuesta en la etapa anterior, en la etapa de comprobación se determina si la respuesta es SI, NO, o no termina el procedimiento.*

Se dice que un algoritmo no determinístico resuelve un problema de decisión π si tiene las dos propiedades siguientes:

- Si $I \in Y_\pi$, entonces existe una estructura S tal que en la etapa de comprobación la respuesta es SI.
- Si $I \notin Y_\pi$, entonces no existe una estructura S con la propiedad anterior.

Nota 1.3.2 *Cualquier algoritmo tradicional, (determinístico), se puede interpretar como un algoritmo no determinístico sin más que comprimir en una las fases de conjetura y comprobación.*

1.3.1 Clases $\mathcal{NP}$ y $\mathcal{NP}$-completa

La clase de problemas $\mathcal{NP}$ está formada por todos aquellos problemas de decisión para los cuales existe un algoritmo de solución que se ejecuta en tiempo polinomial dentro de una máquina no determinista. Dicho de otro modo, no se ha encontrado un algoritmo determinista que lo resuelva en tiempo polinomial.

Definición 1.3.3 *Un algoritmo no determinístico que resuelve un problema de decisión π, se dice que es polinomial si $\forall I \in Y_\pi$ existe alguna estructura S de forma que la etapa de comprobación se realiza en tiempo polinomial.*

Obsérvese que la propia definición asegura que la dimensión de la estructura S está acotada polinomial.

Definición 1.3.4 *Un problema de decisión es de la clase $\mathcal{NP}$ cuando existe un algoritmo no determinístico polinomial que lo resuelva.*

Un ejemplo de problema de decisión de la clase $\mathcal{NP}$ es el del problema del viajante.

Proposición 1.3.5 *Cualquier problema de decisión de clase $\mathcal{P}$ está incluido en la clase $\mathcal{NP}$, es decir, $\mathcal{P} \subset \mathcal{NP}$.*

Demostración. La existencia de un algoritmo, (determinístico), polinomial asegura la existencia de un algoritmo no determinístico polinomial, según se vio en la Nota 1.3.2. ■

Proposición 1.3.6 *Si $\pi \in \mathcal{NP}$, entonces existe un algoritmo exponencial que lo resuelve.*

Demostración. El número de estructuras propuestas en la fase de conjetura es exponencial en el peor de los casos respecto de la dimensión del ejemplo en cuestión. ■

La cuestión más importante, y que sigue sin ser respondida en la actualidad, es si las clases $\mathcal{P}$ y $\mathcal{NP}$ coinciden. Dado que el problema sigue abierto, y considerando la gran cantidad de recursos movilizados sin encontrar una respuesta afirmativa, en lo que sigue se supondrá que:

$$\mathcal{P} \subset \mathcal{NP} \qquad \mathcal{P} \neq \mathcal{NP}$$

o gráficamente lo podemos ver:

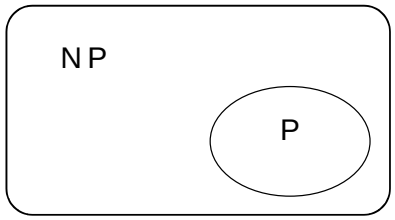


Figura 1.1: Clases $\mathcal{P}$ y $\mathcal{NP}$

Nota 1.3.7 *Suponiendo cierta la conjetura $\mathcal{P} \neq \mathcal{NP}$, el objetivo de la teoría que se está desarrollando es el de clasificar cada problema de decisión planteado. Si se demuestra que el problema es de la clase $\mathcal{NP}$ estrictamente, es decir, no incluido en $\mathcal{P}$, no merece la pena buscar un algoritmo exacto que lo resuelva si la dimensión es suficientemente grande. Habrá, pues, que centrar los esfuerzos en el diseño de algoritmos* **aproximados** *o* **heurísticos**.

La definición de la clase $\mathcal{NP}$ puede parecer tan poco restrictiva que induzca a pensar que cualquier problema de decisión deba pertenecer a dicha clase. Para ver que esto no es cierto, se introduce el concepto de problema de decisión complementario.

Definición 1.3.8 *Dado un problema de decisión π, se define su complementario π^c como el problema de decisión que verifica que $D_\pi = D_{\pi^c}$ y $\forall I \in D_\pi$:*

$$I \in Y_\pi \Longleftrightarrow I \notin Y_{\pi^c}$$

Para problemas polinomiales parece natural pensar que tan fácil es resolver el problema como su complementario. Esta idea se concreta en la siguiente proposición.

Proposición 1.3.9 *Los problemas de la clase $\mathcal{P}$ son simétricos, en el sentido de que*

$$\pi \in \mathcal{P} \Longleftrightarrow \pi^c \in \mathcal{P}.$$

La proposición anterior no es cierta para la clase $\mathcal{NP}$. Basta observar que el complementario del problema del viajante no permite comprobar en tiempo polinomial todas las permutaciones posibles. Se demuestra así que fuera de la clase $\mathcal{NP}$ hay más clases de problemas.

Cuando se trata de clasificar los problemas de decisión por su complejidad o dificultad para ser resueltos, es fundamental el definir unas reglas precisas que permitan relacionar unos problemas con otros. En complejidad algorítmica, casi todos los resultados se obtienen así por comparación.

Definición 1.3.10 *Dados dos problemas de decisión π_1 y π_2, y supuesto que el método de codificación de los ejemplos es razonable, se dice que existe una transformación polinomial de π_1 a π_2, y se denota por $\pi_1 \propto \pi_2$ cuando existe una función*

$$f : D_{\pi_1} \to D_{\pi_2}$$

verificando

1. *f es computable en tiempo polinomial.*
2. *$\forall I \in D_\pi,\ I \in Y_{\pi_1} \iff f(I) \in Y_{\pi_2}$*

Además se verfican las siguientes propiedades:

Proposición 1.3.11 *Dados dos problemas π_1 y π_2, se verifica:*

$$\pi_1 \propto \pi_2 \Longrightarrow \begin{cases} \pi_2 \in \mathcal{P} \Longrightarrow \pi_1 \in \mathcal{P} \\ \pi_1 \notin \mathcal{P} \Longrightarrow \pi_2 \notin \mathcal{P} \end{cases} \tag{1.3.5}$$

Proposición 1.3.12 *Dados tres problemas π_1, π_2 y π_3, se verifica*

$$\begin{cases} \pi_1 \propto \pi_2 \\ \pi_2 \propto \pi_3 \end{cases} \Longrightarrow \pi_1 \propto \pi_3 \tag{1.3.6}$$

Definición 1.3.13 *Dos problemas π_1 y π_2 son polinomialmente equivalentes cuando $\pi_1 \propto \pi_2$ y $\pi_2 \propto \pi_1$.*

Al ser la relación $\propto$ reflexiva y transitiva, induce un *orden parcial* en el conjunto de los problemas de decisión de la clase $\mathcal{NP}$. Un conjunto con un orden parcial no asegura que cualquier par de elementos puedan ser ordenados, en este caso sería un *orden total*. Un ejemplo de conjunto ordenado parcialmente es el cuadrado unidad $[0,1]^2 \subset \mathbb{R}^2$ con el orden parcial definido por:

$$x \leq y \iff x_j \leq y_j \quad j = 1, 2$$

En $[0,1]^2$ hay un elemento mayor $\mathbf{1} = (1,1)^t$ de forma que se verifica

$$x \leq \mathbf{1} \quad \forall x \in [0,1]^2$$

Dado el conjunto $\mathcal{NP}$ ordenado parcialmente con la relación $\propto$ si existiera un elemento maximal, cualquier problema de decisión $\pi \in \mathcal{NP}$ sería transformable polinomialmente a dicho elemento maximal. Dicho problema maximal sería el más general de todos los problemas de la clase $\mathcal{NP}$, puesto que un algoritmo que lo resolviera, resolvería cualquier problema de la clase $\mathcal{NP}$ con la misma dificultad. Si hubiese más de un elemento maximal, serían todos ellos polinomialmente equivalentes, ya que estarían todos ellos incluidos en la clase $\mathcal{NP}$. Realmente, se tendría una clase de equivalencia definida a partir de la relación $\propto$.

Definición 1.3.14 *La clase $\mathcal{NP}$-completa es la clase de equivalencia según la relación $\propto$ formada por los problemas maximales. Dicho de otro modo:*

$$\pi \in \mathcal{NP}\text{-}completa \Longleftrightarrow \left\{ \begin{array}{l} \pi \in \mathcal{NP} \\ \forall \pi' \in \mathcal{NP} \quad \pi' \propto \pi \end{array} \right.$$

En consecuencia, si un problema $\mathcal{NP}$-completo pudiera ser resuelto en tiempo polinomial entonces, los problemas de la clase $\mathcal{NP}$ podrían ser resueltos en tiempo polinomial. Análogamente, si un problema de la clase $\mathcal{NP}$ es intratable entonces, todos los problemas $\mathcal{NP}$-completos son intratables.

La clase de $\mathcal{NP}$-completos está en el complementario de la clase $\mathcal{P}$ dentro de $\mathcal{NP}$.

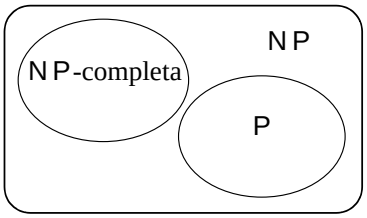


Figura 1.2: Clases de problemas de decisión

Proposición 1.3.15 *Dados dos problemas de decisión π_1, $\pi_2 \in \mathcal{NP}$, entonces:*

$$\left. \begin{array}{c} \pi_1 \in \mathcal{NP}\text{-}completa \\ \pi_1 \propto \pi_2 \end{array} \right\} \Longrightarrow \pi_2 \in \mathcal{NP}\text{-}completa$$

Esta proposición permite demostrar fácilmente cuando un problema de decisión es $\mathcal{NP}$-completo. El procedimiento para demostrar que π es $\mathcal{NP}$-completo sería el siguiente:

1. $\pi \in \mathcal{NP}$
2. $\exists \pi' \in \mathcal{NP}$-completa tal que $\pi' \propto \pi$

La cuestión que queda pendiente es si existe al menos un problema de decisión que sea $\mathcal{NP}$-completo. La respuesta a esta pregunta es afirmativa y el resultado que se conoce como *Teorema de Cook.*

A parte de esta clase de problemas de decisión existen otros problemas que son tan "duros" como los $\mathcal{NP}$-completos. Aparece el concepto de problema $\mathcal{NP}$-duro para problemas generales.

Un *problema de optimización ó búsqueda* está formado por un conjunto de entradas y para cada entrada un conjunto de *soluciones*. Diremos que un algoritmo resuelve un problema de búsqueda si dada una entrada devuelve respuesta negativa cuando no existe ninguna solución y en otro caso proporciona una de sus soluciones.

Es posible generalizar el concepto de transformación polinomial para problemas de búsqueda. Esta generalización está motivada por la observación de que cualquier transformación polinomial entre dos problemas de decisión proporciona un algoritmo polinomial que resuelve el primero a partir de uno, también polinomial, que resuelve el segundo.

Dados dos problemas de optimización o búsqueda π y π', una *reducción de Turing polinomial* de π a π' es un algoritmo A que resuelve π utilizando una hipotética subrutina S para resolver π' de forma que, si S fuera un algoritmo polinomial para π', entonces A sería un algoritmo polinomial para π. Si existe una reducción de Turing polinomial de π en π' escribiremos $\pi \propto_T \pi'$.

Un problema de búsqueda π es $\mathcal{NP}$-duro si existe un problema $\mathcal{NP}$-completo L tal que $L \propto_T \pi$. Evidentemente se puede observar que un problema $\mathcal{NP}$-duro no puede resolverse utilizando un algoritmo polinomial, a menos que $\mathcal{P} = \mathcal{NP}$.

Introducimos a continuación los conceptos fundamentales sobre heurísticas y metaheurísticas. Debe hacerse notar que en la segunda parte de esta memoria se utilizarán fundamentalmente las metaheurísticas *simulated annealing* y los *algoritmos genéticos*.

1.4 Heurísticas

El conocimiento de que un problema es difícil de resolver es instructivo. Puede incluso ser útil si evita que desperdiciemos tiempo buscando un algoritmo que probablemente no exista. Sin embargo, esto no hace que el problema desaparezca. En ocasiones, se debe hallar algún tipo de solución para el problema, tanto si es difícil como si no. Estos son los dominios de la heurística y de los algoritmos aproximados, (ver [85]).

Al hablar de *algoritmos heurísticos* o simplemente *heurísticas*, nos referimos a un procedimiento que puede producir una buena solución para nuestro problema, incluso una solución óptima si somos afortunados, pero que por otra parte puede no producir una solución, o dar lugar a una que no sea precisamente óptima, si no lo somos. La heurística puede ser probabilística o determinista. Por otra parte, puede haber casos para los cuales la heurística, tanto si es probabilísitca como si no, nunca producirá una solución.

Reservamos el término *algoritmo aproximado* para un procedimiento que siempre proprocione algún tipo de solución para el problema, aun cuando quizá no llegue a encontrar la solución óptima. Para que sea útil, también debe ser posible calcular una cota bien de la diferencia, bien de la razón entre la solución óptima y la producida por el algoritmo aproximado.

1.4.1 Algoritmos exactos y heurísticas

Sea el problema de optimización combinatoria π y sea $S_\pi(I)$ el conjunto de soluciones factibles asociado a un ejemplo $I \in D_\pi$. Se introduce el valor de la función objetivo $m_\pi(I, \sigma)$ asociado a la solución $\sigma \in S_\pi(I)$. La solución óptima σ^* debe verificar, (en un problema de mínimo):

$$m_\pi(I, \sigma^*) \leq m_\pi(I, \sigma) \quad \forall \sigma \in S_\pi(I) \tag{1.4.7}$$

Al valor $m_\pi(I, \sigma^*)$ se le denomina $OPT_\pi(I)$.

Definición 1.4.1 *Un algoritmo A se dice que es una heurística o algoritmo aproximado para el problema de optimización combinatoria π cuando $\forall I \in D_\pi$ encuentra en tiempo razonable una solución factible $\sigma \in S_\pi(I)$. Obviamente, se verifica $A(I) = m_\pi(I, \sigma) \geq OPT_\pi(I)$.*

La cuestión es cómo se aproxima $A(I)$ al valor $OPT_\pi(I)$. Evidentemente, si el problema de decisión asociado a π es de la clase $\mathcal{P}$, el algoritmo polinomial correspondiente sería una heurística verificando $A(I) = OPT_\pi(I)$ para cualquier ejemplo $I \in D_\pi$. Sin embargo, cuando el problema sea de la clase $\mathcal{NP}$, el algoritmo no determinístico polinomial no puede ser utilizado como algoritmo; en este caso hay dos opciones básicas:

- **Algoritmos exactos**: Diseñar un algoritmo de búsqueda exacto que recorra todas las soluciones, (enumeración explícita), o parte de ellas, (enumeración implícita). Este enfoque sólo vale para problemas de tamaño reducido.
- **Algoritmos aproximados y/o heurísticas**: Buscan una buena solución aunque no sea la óptima en un tiempo reducido y no pueden asegurar la optimalidad al no recorrer explícitamente todas las soluciones.

Sería deseable poder medir el grado de aproximación de este tipo de algoritmos. Se distinguen así las medidas a *priori* y las medidas a *posteriori.*

Medidas de aproximación a priori

Cociente $R_A(I)$: Se define este cociente a partir del problema de minimización π y de un algoritmo A :

$$R_A(I) = \frac{A(I)}{OPT_\pi(I)} \tag{1.4.8}$$

En el caso de maximización se define de forma inversa de forma que, en cualquier caso, se verifique $R_A(I) \geq 1$. Al depender del ejemplo $I \in D_\pi$, sólo es válido cuando se conozca el óptimo, por lo que limita fuertemente su validez. Sería más interesante que no dependiera del ejemplo para que tuviera una validez general. En esa línea se define el siguiente índice.

Cociente de comportamiento absoluto R_A: Se define según:

$$R_A = Inf\,\{r \geq 1 \ / \ R_A(I) \leq r \quad \forall I \in D_\pi\} \tag{1.4.9}$$

Es muy difícil, sin embargo, identificar este índice para una heurística concreta A que resuelve un problema de optimización π, puesto que debe acotar la función objetivo de cualquier ejemplo $I \in D_\pi$.

Ejemplo 1.4.2 *El problema a resolver es el del viajante en el plano euclídeo TSP_E. Se considera un grafo completo. La heurística se basa en el árbol soporte de mínimo peso y se detalla a continuación.*

1. *Determinar el árbol soporte de mínimo peso. Al representarse en el plano los vértices y al ser la distancia euclídea el grafo es planar.*
2. *Duplicar las aristas del árbol generador para garantizar la existencia de un circuito eule-*

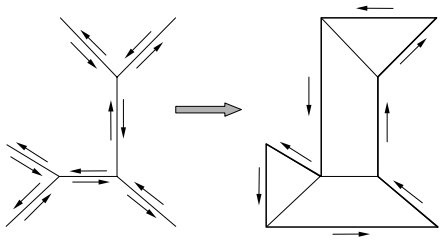


Figura 1.3: Árbol de soporte mínimo y recorrido hamiltoniano

riano que visite todas las aristas y en consecuencia, todos los vértices.

3. Determinar el recorrido hamiltoniano a partir del circuito euleriano:

(a) Determinar un vértice de partida.

(b) Recorrer en el sentido contrario de las agujas del reloj las aristas del circuito euleriano mientras no se repitan vértices hasta llegar al vértice inicial.

(c) Cuando se llegue a un vértice visitado ir directamente, (sin utilizar aristas del árbol soporte), al siguiente vértice no visitado siguiendo la ordenación del circuito euleriano indicada anteriormente.

Formalmente, sea I un ejemplo del problema citado, sea $MST(I)$ el peso del árbol generador de peso mínimo, sea $A(I)$ la longitud del recorrido determinado por la heurística anterior y, por último, sea $OPT(I)$ la longitud del recorrido mínimo del problema del viajante para el ejemplo I. Por construcción del árbol generador, se verifica:

$$MST(I) \leq OPT(I) \quad \forall I \in D_{TSP_E}$$

A continuación se demuestra que

$$OPT(I) \leq 2 \cdot MST(I) \quad \forall I \in D_{TSP_E}$$

de forma que $R_A = 2$.

El caso límite sería el de un grafo con tres vértices alineados y a una distancia unidad entre dos consecutivos. En este caso $MST(I) = 2$ y $A(I) = 4$.

En general, sin embargo, la determinación de este cociente puede ser muy difícil, puesto que se calcula para todos los ejemplos del problema. Se utilizan otras medidas que dependen del ejemplo en cuestión; son las medidas a *posteriori*.

Medidas a posteriori

Suelen ser más ajustadas que las anteriores y están basados en los resultados obtenidos para distintos ejemplos del problema en cuestión. La dificultad estriba en el conocimiento de la solución exacta del problema. Esta será la medida utilizada para el estudio de los mecanimos heurísticos que proponemos en esta memoria.

Basándose en determinados ejemplos aportados por la comunidad científica, se pueden evaluar empíricamente las heurísticas. Es lo que se denomina *análisis medio* en contraposición al análisis del *peor de los casos* que caracteriza los índices a *priori*. A continuación y para terminar esta sección se estudian los algoritmos iterativos, voraces, probabilistas y paralelos. Posteriormente, se estudiarán las meta-heurísticas que permiten construir diferentes algoritmos aproximados: Temple simulado, Algoritmos genéticos o evolutivos y con Hormigas, Búsqueda Tabú, Grasp.

El término metaheurística, fue acuñado en 1986 por Glover, (ver [52]), al introducir la búsqueda tabú. Una metaheurística es una estrategia que rige y controla un conjunto de heurísticas y/o algoritmos aproximados. Cuando se diseña una metaheurística hay un conjunto de parámetros que, al variar sus valores, determinan el conjunto de heurísticas asociadas.

Desde el punto de vista operativo, una metaheurística se puede entender como un procedimiento parametrizado. La codificación de una metaheurística permite aplicar sin ningún esfuerzo computacional gran cantidad de heurísticas.

1.4.2 Algoritmos Iterativos

El esquema general de los algoritmos iterativos es muy simple: a partir de una solución factible, construir otra mejor e iterar este proceso de mejora hasta verificar cierto criterio de parada.

Sea $x^i \in X$ la solución factible inicial. A partir de $x^k \in X$, se construye otra $x^{k+1} = h(x^k) \in X$ verificando $f(x^{k+1}) \leq f(x^k)$ donde $f()$ es la función objetivo del problema de minimización. La elección de la función $h()$ y el valor inicial x^1 caracterizan el método iterativo.

Un caso particular de métodos iterativos son los *algoritmos vecinales*. Para cada solución factible $x \in X$ se define el conjunto de soluciones vecinas $V(x)$ y a partir de una distancia λ se limita este conjunto

$$V_\lambda(x) = \{y \in X \ / \ d(x,y) \leq \lambda\} \tag{1.4.10}$$

Los algoritmos vecinales eligen $x^{k+1} \in V_\lambda(x^k)$.

1.4.3 Algoritmos Voraces (greedy)

A diferencia de los métodos iterativos, que parten de una solución factible y la mejoran en cada iteración, los métodos voraces van construyendo paso a paso la solución de forma que hasta el final no se tiene una solución factible. El nombre se justifica al tratar en cada paso de aportar lo mejor para la solución óptima.

Generalmente, los algoritmos voraces y los problemas que éstos resuelven se caracterizan por la mayoría de las propiedades siguientes:

Tenemos que resolver algún problema de forma óptima. Para construir la solución de nuestro problema, disponemos de un conjunto de candidatos. A medida que avanza el algoritmo, vamos acumulando dos conjuntos. Uno contiene candidatos que ya han sido considerados y seleccionados, mientras que el otro contiene candidatos que han sido considerados y rechazados. Existe una función que comprueba si un cierto conjunto de candidatos constituyen una *solución* de nuestro problema, ignorando si es o no óptima por el momento. Hay una segunda función que

comprueba si un cierto conjunto de candidatos es *factible* , esto es, si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución de nuestro problema. Hay otra función más, la *función de selección*, que indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados. Por último, existe una *función objetivo* que da el valor de la solución que hemos hallado.

Para resolver nuestro problema, buscamos un conjunto de candidatos que contituya una solución, y que optimice el valor de la *función objetivo*. Los algoritmos voraces avanzan paso a paso. Inicialmente, el conjunto de elementos seleccionados está vacio. Entonces, en cada paso se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra elección por la función de selección. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible, rechazamos el candidato que estamos considerando en ese momento. Sin embargo, si el conjunto aumentado sigue siendo factible, entonces añadimos el candidato actual al conjunto de candidatos seleccionados. Cada vez que se amplía el conjunto de candidatos seleccionados, comprobamos si éste constituye ahora una solución para nuestro problema. En forma de pseudocódigo una algoritmos voraz lo podemos expresar de la siguiente manera:

```
[1]  función voraz (C : conjunto) : conjunto
[2]    {S ← ∅;
[3]    mientras C ≠ ∅ y no solución(S) hacer
[4]      x ← seleccionar(C);
[5]      C ← C \ {x};
[6]      si factible (S ∪ {x}) entonces S ← S ∪ {x};
[7]    si solución(S) entonces return(S)
[8]    si_no return("no hay soluciones");
[9]    }
```

Está clara la razón por la cual tales algoritmos se denominan *voraces*: en cada paso, el procedimiento selecciona el mejor bocado que puede tragar, sin preocuparse por el futuro. Nunca cambia de opinión: una vez que un candidato se ha incluido en la solución, queda allí para siempre; una vez que se excluye un candidato de la solución, nunca vuelve a ser candidato.

En el algoritmo de Kruskal para el problema del árbol generador de mínimo peso, en cada paso se añade una arista que no forme ciclo con las anteriormente seleccionadas y que sea la de menor peso no considerada hasta ese instante.

Desafortunadamente, no siempre se puede garantizar la optimalidad de la solución obtenida siguiendo este método. A continuación se detalla un algoritmo de este tipo para el problema del viajante.

Ejemplo 1.4.3 ALGORITMO DE LA CIUDAD MÁS PRÓXIMA*: Se supone la desigualdad triangular, es decir, para cualquier grupo de tres ciudades a, b y c, se verifica siempre $d(a,c) \leq d(a,b) + d(b,c)$. El algoritmo se basa en escoger, a partir de una ciudad inicial, la ciudad más próxima a ella que no haya sido visitada y construir así el recorrido hasta unir la última con la primera.*

1.4.4 Algoritmos Probabilístas

Cuando un algoritmo se enfrenta a una decisión, a veces es preferible seguir un curso de acción aleatorio, en lugar de invertir tiempo en determinar cuál de las alternativas es la mejor. Esta situación se produce cuando el tiempo necesario para determinar la opción óptima es excesivo en comparación con el tiempo que se ahorra de análisis en el caso medio al tomar esta decisión óptima.

La característica fundamental de los algoritmos probabilistas es que un mismo algoritmo se puede comportar de forma distinta cuando se aplica dos veces a un mismo caso. Su tiempo de ejecución, e incluso el resultado obtenido, puede variar considerablemente entre usos consecutivos. Esto se puede explotar de muchas maneras. Por ejemplo, no se permite que un algoritmo determinista pierda el control, (bucle infinito, división por cero) porque si hace esto en un caso dado, entonces nunca se podrá resolver ese caso con ese algoritmo. Por contraste, este comportamiento es admisible para un algoritmo probabilista siempre y cuando se produzca con una probabilidad razonablemente pequeña en cualquier caso dado: si el algoritmo se atasca, basta reiniciarlo para ese mismo caso y dispondremos de una nueva oportunidad de éxito. Otra ventaja de este enfoque es que si hay más de una respuesta correcta, se pueden obtener varias diferentes ejecutando el algoritmo probabilista más de una vez; un algoritmo determinista siempre llega a la misma respuesta, aunque por supuesto se puede programar para buscar varias.

Otra consecuencia del hecho de que los algoritmos probabilistas se puedan comportan de forma distinta cuando se ejecutan dos veces con las mismas entradas es que algunas veces les permitiremos que produzcan resultados erróneos. Supuesto que esto suceda con probabilidad suficientemente pequeña en cualquier caso dado, basta invocar el algoritmo varias veces sobre el caso deseado para llegar a una confianza arbitrariamente grande de que se ha obtenido la respuesta correcta. Por constraste, un algoritmo determinista que da respuestas erróneas con algunas entradas es inadmisible para la mayoría de las aplicaciones, porque siempre fallará en esas entradas.

Hay tres categorías de algoritmos probabilistas; los dos primeros no garantiza la corrección del resultados y sin embargo el tercero sí:

Algoritmos numéricos

Estos algoritmos producen un *intervalo de confianza* de la forma "con una probabilidad del 90%, la respuesta correcta es 59 ± 3". Cuanto más tiempo se conceda a estos algoritmos numéricos, más preciso es el intervalo. Las respuestas de opinión ofrecen un ejemplo familiar de este tipo de respuestas.

Algoritmos de Monte Carlo

Los denominados algoritmos de *Monte Carlo* dan la respuesta exacta con una elevada probabilidad, aunque a veces proporcionan una respuesta incorrecta. En general, no se puede saber si la respuesta proporcionada por un algoritmo de *Monte Carlo* es correcta, pero se puede reducir arbitrariamente la probabilidad de error dando más tiempo al algoritmo. Como ejemplo de este

tipo de algoritmos damos a continuación un algoritmo para la comprobación de primalidad.

Comprobación de primalidad. Es posible que el algoritmo de *Monte Carlo* más famoso sea el que decide si un entero impar dado es o no primo. La comprobación de primalidad es un tema crucial en la criptografía moderna.

La historia de la comprobación probabilista de *primalidad* tiene sus raíces en Pierre de Fermat, el padre de la Teoría de los Números moderna. En 1640 enunció el teorema siguiente, que a veces se denomina teorema menor de Fermat:

Teorema 1.4.4 (Fermat). *Sea n un primo. Entonces* $a^{n-1} \bmod n = 1$ *para cualquier entero a tal que* $1 \leq a \leq n-1$.

Considérese ahora la versión contrapositiva del teorema de Fermat: si n y a son enteros tales que $1 \leq a \leq n-1$ y si $a^{n-1} \bmod n \neq 1$ entonces n no es primo. Esto sugiere el siguiente algoritmo probabilista para la comprobación de *primalidad*. Suponemos que $n \geq 2$:

```
[1]  función MonteCarlo (n : entero) : booleano
[2]    {a ← uniforme(1..n − 1);
[3]     si (a^(n-1) mod n = 1) entonces return(true);
[4]     si_no return(false);
[5]    }
```

¿Qué se puede decir, sin embargo, si `MonteCarlo(n)` devuelve el valor `true`? Para concluir que n es primo, necesitaríamos el recíproco y el contrapositivo del teorema de Fermat. Este resultado diría que $a^{n-1} \bmod n$ nunca es igual a 1 cuando n es compuesto y $1 \leq a \leq n-1$. Desafortunadamente, esto no es cierto, porque $1^{n-1} \bmod n = 1$ para todo $n \geq 2$. Un entero a tal que $2 \leq a \leq n-2$ y que $a^{n-1} \bmod n = 1$ se denomina *testigo falso de primalidad* para n si de hecho n es compuesto. Sin embargo, los testigos falsos son bastantes escasos, aunque existen números compuestos que admiten una proporción significativa de falsos testigos. Afortunadamente, una pequeña modificación de la comprobación de `MonteCarlo` resuelve esta dificultad, (ver [17]).

Algoritmos de Las Vegas

Los algoritmos de *Las Vegas* toman decisiones probabilistas como ayuda para guiarse más rápidamente hacia una solución correcta. A diferencia de los algoritmos de *Monte Carlo*, nunca proporcionan una respuesta incorrecta. Hay dos categorias principales de algoritmos de *Las Vegas*:

1. Puede utilizarse la aleatoriedad para guiar su búsqueda de tal manera que se garantice una solución correcta aun cuando se tomen decisiones poco afortunadas: si esto ocurriera, tan sólo necesitarán más tiempo. Este tipo de algoritmos suele utilizarse cuando un algoritmo determinista conocido para resolver el problema en cuestión es mucho más rápido en el caso promedio que en caso peor. Al incorporar un elemento aleatorio, se permite a los algoritmos de *Las Vegas* reducir, y a veces eliminar, esta diferencia entre casos buenos

y malos. Un ejemplo prodría ser el algoritmo de ordenación *Quicksort*, tomando como elemento pivote un elemento aleatorio en el rango del problema.

2. La otra categoría de algoritmos de *Las Vegas* se caracteriza por tomar de vez en cuando decisiones que los llevan a un callejón sin salida. Se pide que estos algoritmos sean capaces de reconocer su situación, en cuyo caso se limitarán a admitir el fallo. Este comportamiento sería intolerable en un algoritmo determinista, por cuanto supondría que no sería capaz de resolver el caso considerado. Sin embargo, la naturaleza probabilista de los algoritmos de *Las Vegas* hace que esta admisión de fallo sea aceptable siempre y cuando no se produzca con excesiva probabilidad: cuando se produzca un fracaso basta con volver a aplicar el mismo algoritmo al mismo caso para tener una nueva posibilidad de éxito.

   Cuando se permite que falle un algoritmo de *Las Vegas*, es más cómodo representarlo en la forma de un **procedimiento** que en forma de una **función**. Esto permite tener un parámetro de salida *éxito*, que recibe el valor *verdadero* si se obtiene una solución, o *falso* en caso contrario. La llamada típica para resolver el caso `x` es `LV(x,y,éxito)`, donde el parámetro de salida `y` recibe la solución siempre que `éxito` reciba el valor verdadero.

   Sea $p(x)$ la probabilidad de éxito del algoritmo cada vez que se le pide que resuelva el caso x. Para que un algoritmo merezca el nombre "*Las Vegas*", exigimos que $p(x) > 0$ para todo caso x. Es aún mejor si existe una constante $\delta > 0$ tal que $p(x) \geq \delta$ para todo caso x, puesto que en caso contrario el número esperado de repeticiones antes del éxito podría crecer arbitrariamente con el tamaño del caso.

   Consideremos el algoritmo siguiente:

```
[1]  función RepetirLV(x);
[2]  repetir
[3]    LV(x,y,éxito);
[4]  hasta que (éxito==true);
[5]  return(y);
```

   Dado que cada llamada `LV(x)` tiene una probabilidad de éxito $p(x)$, el número esperado de pasadas por el bucle es $1/p(x)$. Un parámetro más interesante es el tiempo esperado $t(x)$ antes de que `RepetirLV(x)` tenga éxito. Se puede pensar a primera vista que $t(x)$ es simplemente $1/p(x)$ multiplicado por el tiempo esperado que requiera cada llamada a `LV(x)`. Sin embargo, un análisis correcto debe considerar por separado el tiempo esperado tomado por `LV(x)` en caso de éxito y en caso de fracaso. Denotemos estos tiempos esperados mediante $s(x)$ y $f(x)$.

   Con probabilidad $p(x)$, la primera llamada a `LV(x)` tiene éxito al cabo de un tiempo esperado $s(x)$. Con probabilidad $1 - p(x)$, la primera llamada a `LV(x)` fracasa al cabo de un tiempo esperado $f(x)$. Después, volvemos al punto de partida, y seguimos estando a un tiempo esperado $t(x)$ del éxito. El tiempo esperado total en este caso es por tanto $f(x) + t(x)$. Por tanto tenenos:

$$t(x) = p(x)s(x) + (1 - p(x))(f(x) + t(x)) \tag{1.4.11}$$

que se resuelve fácilmente

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} f(x) \tag{1.4.12}$$

1.4.5 Algoritmos Paralelos

En otras partes de esta memoria suponemos implícitamente que nuestros algoritmos se ejecutan en una máquina que sólo puede ejecutar una instrucción a la vez. Por supuesto, cualquier máquina moderna simultanea el cálculo con las operaciones de entrada/salida. Muchas de ellas simultanean también distintas operaciones aritméticas cuando se calcula una expresión, de tal manera que las sumas, por ejemplo, se pueden efectuar en paralelo con las multiplicaciones. Si consideramos esta posibilidad, entonces podemos tener la esperanza de acelerar algunos de nuestros algoritmos.

Las computadoras capaces de efectuar estos cálculos en paralelo no están todavía muy extendidas. Sin embargo, su número crece, y el interés por los *algoritmos paralelos*, que aprovechan esta capacidad, también.

1.5 Metaheurísticas

El auge que experimentan los procedimientos heurísticos se debe sin duda a la necesidad de disponer de herramientas que permitan ofrecer soluciones rápidas a problemas reales. Es importante destacar el hecho de que los algoritmos heurísticos, (por sí solos), no garantiza la optimalidad de la solución encontrada, aunque su propósito es encontrar una solución cercana al óptimo en un tiempo razonable. Sin embargo, la gran cantidad de publicaciones donde problemas de gran dificultad son resueltos con gran rapidez, (en muchos casos óptimamente), avalan estos métodos.

Dentro de las técnicas heurísticas podemos encontrar diversos métodos, tales como: Métodos constructivos, de descomposición, de reducción, de manipulación del modelo y de búsqueda local. Tradicionalmente para resolver un problema dado se diseñaba un algoritmo específico que pertenecía a algunos de los métodos enumerados. Hoy día, el interés primordial de los investigadores del área es el de diseñar métodos generales que sirvan para resolver clases o categorías de problemas. Dado que estos métodos generales sirven para construir o guiar el diseño de métodos que resuelvan problemas específicos se les ha dado el nombre de Metaheurísticos. Los pofesores Osman y Kelly (1995) introducen la siguiente definición:

"Los procedimientos Metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que las heurísticas clásicas no son ni efectivos ni eficientes. Las Metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de: inteligencia artificial, evolución biológica y mecanismos estadísticos."

En estas páginas vamos a abordar los procedimientos Metaheurísticos más utilizados y reconocidos en Optimización Combinatoria: *Recocido Simulado, (simulated annealing), Algoritmos*

Genéticos, Búsqueda Tabú, Ant System, (Algoritmos con Hormigas), GRASP. Los dos primeros serán utilizados en las metaheurísticas propuestas en esta memoria.

1.5.1 Simulated Annealing

Considerando los problemas de optimización combinatoria como problemas que buscan el óptimo global, se pueden incluir procedimientos de búsqueda estocástica como alternativas heurísticas. El temple simulado traducción del término inglés *simulated annealing*, se basa en determinados principios de termodinámica.

Esquema general del método

Un problema de optimización combinatoria puede plantearse de la siguiente forma:

Dado un espacio finito de *configuraciones* o soluciones $S = \{x_1, ..., x_m\}$, donde m es la dimensión de dicho espacio, dada una función de costes $C : S \to \mathbb{R}$, determinar $x^* \in S$ tal que $C(x^*) \leq C(x) \quad \forall x \in S$.

Para entender mejor el temple simulado, se recuerda el esquema general de un algoritmo de minimización local.

Algoritmo de minimización local

Sea $x \in S$

[1] `Repetir`
[2] {`Genera` $y \in V(x) \subset S$;
[3] `Evalúa` $\delta = C(y) - C(x)$;
[4] `si` $(\delta < 0)$ `entonces` $x \leftarrow y$;
[5] }`Hasta que` $C(y) \geq C(x) \quad \forall y \in V(x)$;

El problema que tiene este esquema es que depende fuertemente de la solución inicial elegida. Por tanto una solución puede quedar atrapada en un mínimo local y no llegar nunca al mínimo global.

El temple simulado introduce una variable de control T, (denominada temperatura), que permite, con una cierta probabilidad, empeorar la función objetivo para así salir de una hondonada donde ha podido quedar atrapado un mínimo local. Esta probabilidad se denomina *función de aceptación* y normalmente se evalúa según:

$$e^{\left(\frac{-\delta}{T}\right)} \tag{1.5.13}$$

siendo δ el incremento o empeoramiento de la función objetivo.

La estrategia del *temple simulado* es comenzar con una temperatura inicial alta, lo cual proporciona una probalidad también alta de no mejora. En cada iteración se va reduciendo la temperatura y por tanto las probabilidades son cada vez más pequeñas conforme avanza el procedimiento y nos acercamos a la solución óptima. De este modo, inicialmente se realiza una diversificación de la búsqueda sin controlar demasiado el coste de las soluciones visitadas. En

iteraciones posteriores resulta cada vez más difícil aceptar malos movimientos, y por tanto se produce un descenso en el coste. A igualdad de incremento, la probabilidad de aceptarlo es mayor para valores de T altos; en el caso límite $T = \infty$ se acepta cualquier empeoramiento del coste; en el valor límite $T = 0$ no acepta ningún incremento de la función objetivo y se tendría el esquema normal de minimización local.

La probabilidad se introduce a partir de números aleatorios u siguiendo la distribución $U(0, 1)$. El esquema básico del *temple simulado* sería el que resulta de incluir esta modificación, (variando el parámetro T desde un valor positivo hasta el valor nulo), en el algoritmo anterior.

Algoritmo de recocido simulado homogéneo

Si $x \in S$ es la configuración inicial y $T > 0$ la temperatura inicial, el esquema general del *temple simulado* es el siguiente:

```
[1]  Repetir
[2]    {Repetir
[3]      {Genera solución y ∈ V(x) ⊂ S;
[4]       Evalúa δ ← C(y) − C(x);
[5]       si (δ < 0) entonces x ← y
[6]       si_no
[7]          si ((δ ≥ 0) ∧ (U(0,1) < e^(−δ/T))) entonces x ← y;
[8]       n ← n + 1;
[9]      }mientras (n ≤ N(T))
[10]   Disminuir T;
[11]   }mientras (parada==false);
```

Analogía física del recocido simulado

El nombre se justifica por el templado o enfriado controlado con el que se producen determinadas sustancias; es el caso, por ejemplo, del proceso de cristalización del vidrio. Inicialmente, a temperaturas muy elevadas, se produce una amalgama líquida. En este líquido las partículas se configuran aleatoriamente. El estado sólido se caracteriza por tener una configuración de mínima energía y es una configuración concreta, (el mínimo global). Para alcanzar esta configuración, es preciso templar la amalgama lentamente, puesto que un enfriamento súbito obstaculizaría el proceso y se llegaría a una configuración distinta a la esperada, (un mínimo local distinto del mínimo global).

El *temple simulado* se puede considerar como una variación de la simulación de *Monte Carlo*: en 1953, Metropolis simuló el comportamiento de una colección de átomos a una cierta temperatura. En cada iteración, cada átomo es sometido a un desplazamiento aleatorio que provoca un cambio global en la energía del sistema (δ). Si $\delta < 0$, se acepta el cambio; en caso contrario, el cambio se acepta con una probabilidad $e^{(-\delta/K_B T)}$, siendo K_B la constante de Boltzman y T la temperatura absoluta. Para un número grande de iteraciones el sistema alcanza el equilibrio en cada temperatura, y la distribución de probabilidad del sistema sigue la distribución de

Boltzman:

$$P\{X_T = i\} = \frac{1}{Z(T)} e^{\frac{-E_i}{K_B T}}$$

siendo E_i la energía del estado i y $Z(t)$ la constante de normalización $Z(t) = \Sigma_i e^{\frac{-E_i}{K_B T}}$. A la función $e^{\frac{-\delta}{K_B T}}$ se la denomina función de aceptación y asegura que el sistema converja a la distribución de Boltzman.

Implementación del método

Dado un problema de optimización, es preciso adaptarlo al esquema descrito anteriormente, lo que se consigue concretando los siguientes aspectos:

- **Adaptación del problema**:
  - Conjunto S de configuraciones o soluciones factibles del problema.
  - Función de coste C.
  - Vecindad de cada configuración.
  - Configuración inicial.
- **Estrategia de templado**:
  - Temperatura inicial.
  - Disminución de la temperatura en cada iteración, (por ejemplo $T = \alpha T$ $(0 < \alpha < 1)$).
  - Número de iteraciones de cada temperatura $N(T)$.
  - Criterio de parada.

A continuación exponemos a modo de ejemplo un algoritmo que utiliza la metaheurística de *temple simulado* para el problema del viajante.

Ejemplo 1.5.1 ALGORITMO DE CERNY: *Sea n el número de ciudades y sea d_{ij} la matriz de distancia $n \times n$ no necesariamente simétrica. El conjunto de configuraciones es el conjunto de permutaciones de n elementos:*

$$S = \{(s_1, ..., s_n)\}$$

La función de coste es

$$C(s_1, ..., s_n) = d_{s_n, s_1} + \sum_{k=1}^{n} d_{s_k, s_{k+1}}$$

El algoritmo quedaría de la siguiente forma:

▽ **Configuración inicial**: $(s_1, ..., s_n)$.

▽ **Temperatura inicial**: T.

```
[1]  Inicio
[2]    (c_1, ..., c_n) ←(s_1, ..., s_n);
[3]     d ← C(c_1, ..., c_n);
[4]  Repetir disminuyendo T hasta parada
[6]     {Desde i = 1 hasta n
[7]       {Generar  j ∈ {1, .., n}   j ≠ i;
[8]            /* Generar configuración vecina j − ésima (t_1, ..., t_N) */
[9]        i_0← Min{i, j};
[10]       j_0← Max{i, j};
[11]       t_k← c_k   k = 1, .., i_0−1;
[12]       t_{i_0+k}← c_{j_0−k}   k = 0, 1, .., j_0−i_0;
[13]       t_k← c_k   k = j_0+1, .., n;
[14]       d'← C(t_1, ..., t_n);
[15]       si ((d'< d) ∨ (d'≥ d ∧ u < e^((d−d')/T))) entonces
[16]         {d ← d';
[17]          (c_1, ..., c_n) ← (t_1, ..., t_n);
[16]         }
[17]       }
[18]     }
[19]  fin
```

1.5.2 Algoritmos Genéticos

Los algoritmos genéticos están basados en los mecanismos de la genética y la selección natural. Como se verá más adelante, es la única metaheurística que trabaja simultáneamente con conjuntos de soluciones factibles, que las considerará como individuos de una población que se cruza, reproduce y puede incluso mutar para sobrevivir.

Fueron introducidos por Holland en 1975 en su trabajo *"Adaptation in Natural and Artificial Systems"* [60]. El objetivo de su investigación era doble: por un lado, explicar de forma rigurosa los procesos de adaptación natural en los seres vivos; y, por otro lado, diseñar programas de ordenador basados en dichos mecanismos naturales.

Introducción

Durante las últimas décadas ha habido un creciente interés en algoritmos basados en el principio de la evolución, (supervivencia del más apto). Entre los algoritmos evolutivos más conocidos se incluyen los algoritmos genéticos, ([29, 55, 40]), programación evolucionaria, ([43]), estrategias evolutivas, ([7, 74]) y programación genética, ([69]). El conjunto de estas técnicas se agrupa bajo el nombre de *computación evolucionaria*.

Los algoritmos evolutivos son métodos robustos de búsqueda, que permiten tratar problemas de optimización donde el objetivo es encontrar un conjunto de parámetros que minimizan o maximizan una función de adaptación. Estos algoritmos operan con una población de individuos $A(t) = \{x_1^t, ..., x_n^t\}$, para la iteración t, donde cada individuo x_i^t representa un punto de búsqueda en el espacio de las soluciones potenciales a un problema dado. El objetivo de un individuo x_i se evalúa según una función de aptitud $f(x_i)$. Esta función permite ordenar del mejor al peor los individuos de la población.

La población inicial evoluciona sucesivamente hacia mejores regiones del espacio de búsqueda mediante procesos probabilísticos de:

1. Selección de los individuos más adaptados en la población, (a mayor grado de adaptación mayor probabilidad de dejar descendencia).

2. Modificación por recombinación y/o mutación de los individuos seleccionados.

La estructura del algoritmo evolutivo básico es la siguiente:

```
[1]  Inicio
[2]    {t ← 0;
[3]     Inicializar(A(t));
[4]     Evaluar(A(t));
[5]     Mientras(no parada)
[6]       {t← t + 1;
[7]        Seleccionar A(t) a partir de A(t-1);
[8]        Recombinar y/o mutar A(t);
[9]        Evaluar A(t);
[10]      }
[11]   }
```

Veamos los elementos que determinan la selección y modificación de los individuos, así como las diferentes variantes de éstos.

Algoritmo genético simple

El algoritmo genético simple se aplica a problemas de optimización de parámetros continuos de la forma:

$$\min \ f(x_{k1}, x_{k2}, ..., x_{kn}), \qquad x_{ki} \in [l_i, u_i] \in \mathbb{R}, \quad l_i < u_i, \quad \forall i = 1, ..., n \tag{1.5.14}$$

donde cada componente x_{ki} tiene un dominio definido por una cota inferior l_i y una cota superior u_i, y por lo tanto el espacio de búsqueda es un subconjunto de $\mathbb{R}^n$.

A continuación se especifica un algoritmo genético simple y se explican brevemente sus fundamentos.

1. **Representación de la Función de Adaptación**

   Un punto de búsqueda que se denota como $\vec{x}_k = (x_{k1}, x_{k2}, ..., x_{kn})$, se representa mediante una tira binaria, (*binary string*). Cada una de las n componentes del vector $\vec{x}_k$ se codifica en binario usando b bits. Luego, las representaciones binarias de cada parámetro se concatenan en una sola tira, obteniéndose individuos de largo $l = nb$ bits.

   El algoritmo genético considera una población inicial de M tiras binarias de largo l, generadas aleatoriamente. Para evaluar estos individuos se requiere decodificar cada componente representada en binario al entero correspondiente entre 0 y 2^b-1 y luego reescalarlo en el intervalo real correspondiente al dominio de esa componente según la fórmula 1.5.14. Durante el proceso evolutivo, el algoritmo genético genera una nueva población de tamaño M a partir de la población actual y evalúa las aptitudes de los nuevos individuos.

2. **Operador de de Selección**

   El mecanismo de selección permite orientar la búsqueda a aquellos puntos más destacados, es decir, con la mayor adapatación abservada hasta el momento. El operador de selección genera a partir de la población actual una población intermedia del mismo tamaño, reproduciendo con un mayor número de copias a los individuos más aptos y eliminando o asignando un menor número de copias a los individuos menos aptos. El operador de selección no produce puntos nuevos en el espacio de búsqueda, sino que determina qué individuos dejarán descendencia y en que cantidad en la próxima generación.

   El algoritmo genético simple utiliza una regla se supervivencia probabilista. En analogía con un problema de teoría de juegos, (*multi-armed bandit problem)*, John Holland postuló que la estrategia óptima de selección consiste en aumentar exponencialmente el número de copias del mejor individuo observado respecto al peor. Este método se conoce como selección proporcional.

   - SELECCIÓN PROPORCIONAL: La probabilidad de selección p_{it} del i-ésimo individuo en la población $A(t)$ depende de la adaptación relativa de éste con respecto a la población:

   $$p_{it} = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{1.5.15}$$

   donde f_j es la adaptación del j-ésimo individuo. El número esperado de copias N_e del i-ésimo individuo en la próxima generación es:

   $$N_e\,[i] = Mp_{it} = \frac{f_i}{\bar{f}_t} \tag{1.5.16}$$

   donde $\bar{f}_t$ es la adapatación promedio de la población $A(t)$ y M es el tamaño de ésta.

   La fase de selección de un algoritmo genético basado en valores esperados se compone de dos partes: Determinación de los valores esperados N_e y conversión de los valores esperados a números discretos de descendencia, (muestreo). El algoritmo

de muestreo debe mantener una población constante y al mismo tiempo proveer de un muestreo exacto, consistente y eficiente. El algoritmo de muestreo original propuesto por Holland se conoce como método de la ruleta.

Método de la ruleta:

(a) Determinar la suma S de las adaptaciones de todas la población.
(b) Relacionar uno a uno todos los individuos con segmentos contiguos de la recta real $[0, S)$, tal que cada segmento individual sea igual en su tamaño a su grado de adaptación.
(c) Generar un número aleatorio en $[0, S)$.
(d) Seleccionar el individuo cuyo segmento cubre el número aleatorio.
(e) Repetir el proceso hasta obtener el número deseado de muestras.

El método de la ruleta sufre de una dispersión ilimitada, es decir la discrepancia entre el número esperado de copias y el número real obtenido por el método de la ruleta puede ser la máxima posible. El Muestreo estocástico universal corrige esta situación [7].

Muestreo estocástico universal: Es análogo a una ruleta con M punteros igualmente espaciados entre sí, de modo que con un solo lanzamiento se obtienen M ganadores. El método no tiene sesgo y su dispersión es la mínima posible. El algoritmo es como sigue:

```
[1]   sum←0;
[2]   ptr←rand()∈ N_e[i];
[3]   For k = 1 to M do
[4]     Inicio
[5]       sum←sum+N_e[i];
[6]       while(sum > ptr) do
[7]         Inicio
[8]           selectind[i];
[9]           ptr ← ptr + 1;
[10]        Fin
[11]    Fin
```

donde $N_e[i]$ es el número esperado de copias para el individuo i-ésimo según la fórmula 1.5.16. Debe tomarse en cuenta que éste método puede sólo reducir el error de muestreo pero no eliminarlo completamente.

Por otra parte la determinación del valor esperado mediante la fórmula 1.5.16 es muy sensible a la presencia de un individuo super adaptado en la población actual, pudiendo éste llevarse un elevado número de copias generación tras generación y causar una convergencia prematura del algoritmo a un óptimo local, especialmente para poblaciones pequeñas. Una forma de resolver la convergencia por presencia de superindividuos es usar selección por ranking.

- SELECCIÓN POR RANKING: El algoritmo de selección por ranking es como sigue:
  (a) Ordenar la población del mejor individuo, $(x = 1)$, al peor, $(x = M)$.
  (b) Asignar un número de copias esperadas según:

$$\alpha(x) = \eta^+ - \left(\eta^+ - \eta^-\right) \frac{(x-1)}{(M-1)} \tag{1.5.17}$$

  donde

$$\sum_x \alpha(x) = M, \quad 1 \leq \eta^+ \leq 2, \quad \eta^- = 2 - \eta^+ \tag{1.5.18}$$

  siendo η^+ el máximo valor esperado, $(1.1 - 1.2$ recomendado), y η^- mínimo valor esperado.
  (c) Usar muestreo estocástico universal para llenar la población.

- SELECCIÓN POR TORNEO: Este método de selección no se basa en valores esperados y no requiere por lo tanto de un algoritmo de muestreo. El algoritmo es como sigue:
  (a) Escoger tamaño de torneo q, (típicamente $q = 2$).
  (b) Crear una permutación aleatoria de M enteros.
  (c) Comparar la adaptación de los próximos q-miembros de la población y seleccionar el mejor.
  (d) Si se acaba la permutación, generar una una nueva permutación.
  (e) Repetir hasta llenar la población.

  Selección por torneos de tamaño $q = 2$ es análogo a la selección por ranking con $\eta^+ = 2$, ya que ambos métodos asignan dos copias al mejor, cero copias al peor y una al promedio.

Los distintos métodos de selección pueden ser analizados en términos de su presión selectiva. Ésta se mide como el inverso del tiempo requerido por el mejor individuo para llenar la población con copias de si mismo, cuando no actúa otro operador genético.

Los métodos de selección se ordenan en orden creciente de presión selectiva, (para valores estándares de sus parámetros), del modo siguiente: selección proporcional, ranking y torneo [7].

3. Operador de Cruce

El operador de cruce, (*crossover*), es el operador de búsqueda más importante en los algoritmos genéticos. Este es un operador que intercambia el material genético de un par de padres produciendo descendientes que normalmente difieren de sus padres. La idea central es que segmentos distintos de padres diferentes con alta adaptación deberían combinarse en nuevos individuos que tomen ventaja de esta combinación.

El algoritmo genético explota las regiones con mayor adaptación, ya que generaciones

sucesivas de selección y cruce producen un número creciente de puntos en estas regiones.

El operador de cruce opera con probabilidad p_c, (esto permite que en algunos casos no haya cruce y se mantegan los padres). Dados $\vec{p}$ y $\vec{q}$ un par de padres, de largo l bits, se escoge aleatoriamente un punto $k \in \{1, ..., l-1\}$ y se intercambian los bits a la derecha de esa posición entre ambos individuos, obteniéndose los descendientes $\vec{s}$ y $\vec{v}$, como se indica a continuación:

$$\left.\begin{array}{l} \vec{p} = (p_1, .., p_{k-1}, p_k, .., p_t) \\ \vec{q} = (q_1, .., q_{k-1}, q_k, .., q_t) \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \vec{s} = (p_1, .., p_{k-1}, q_k, .., q_t) \\ \vec{v} = (q_1, .., q_{k-1}, p_k, .., p_t) \end{array}\right. \tag{1.5.19}$$

El operador *crossover* de un punto, descrito más arriba, sufre de un sesgo posicional ya que un bit cercano al extremo derecho de la tira tiene una alta probabilidad de intercambio, mientras que un bit en el extremo izquierdo tiene una baja probabilidad de intercambio. El operador *crossover binomial* corrige este sesgo, intercambiando bits entre padres sobre una base bit a bit, con probabilidad $p \in [0, 1]$ aleatoria, distinta para cada posición.

4. **Operador de Mutación**

En el algoritmo genético simple, el operador de mutación juega un papel secundario, invirtiendo ocasionalmente un bit. Tasas de mutación pequeñas garantizan que un individuo no difiera mucho de sus padres en el genotipo, (tira binaria). La mutación sirve para evitar la perdida de diversidad producto de bits que han convergido a un cierto valor para toda la población, y que por tanto no pueden ser recuperados por el operador de recombinación. El operador de mutación invierte cada bit de la tira binaria sobre una base bit a bit con probabilidad p_m:

$$m'_{\{p_m\}}(q_1, q_2, ..., q_l) = \left(q'_1, q'_2, ..., q'_l\right), \quad \forall i \in \{1, .., l\}$$

donde

$$q'_i = \left\{\begin{array}{ll} q_i & si\ r > p_m \\ 1 - q_i & si\ r \leq p_m \end{array}\right. \tag{1.5.20}$$

y $r \in [0, 1]$ uniformemente aleatorio, distinto para cada bit q_i.

Concepto de esquema

Considerando las distintas soluciones del problema de optimización combinatoria como cadenas de caracteres, existe un paralelismo con conceptos de genética: Cada carácter, (bit), es un gen; la cadena de caracteres, (conjunto de 5 bits por ejemplo), es un cromosoma. De esta forma, las modificaciones de las distintas soluciones se pueden interpretar como modificaciones genéticas producidas por la reproducción, cruce y mutación de los genes de los cromosomas.

Con el fin de distinguir las características de un individuo concreto de una generación y las de toda la generación en concreto, se introduce el concepto de esquema.

Si se incorpora el valor $*$ en los bits, identificando cualquiera de los valores 0 o 1, se representa el conjunto de las 8 soluciones con 1 en los 2 bits de la izquierda como $(11 * **)$.

Formalmente, se introduce la siguiente notación:

- $V = \{0, 1\}$ es el alfabeto binario.
- k es el número de caracteres de la cadena.
- n es el tamaño de cada generación de individuos.
- t es el índice de la generación.
- $A(t)$ son los individuos de la generación t-ésima.
- $V^+ = V \cup \{*\}$ es el alfabeto extendido.
- $H \in V^{+k}$ es un esquema.

Para comparar los esquemas se introducen los siguientes conceptos:

Definición 1.5.2 *El orden de un esquema es el número de caracteres fijos de dicho esquema. Se representa por* $o(H)$.

Definición 1.5.3 *La longitud de un esquema es la distancia entre las posiciones extremas de los caracteres fijos. El primer carácter es el de la izquierda y el último el de la derecha. Se representa por* $\delta(H)$.

Si tenemos los esquemas $H^1 = (0 * * * **)$ y $H^2 = (011 * 1*)$, entonces:

$$\begin{array}{ll} o(H^1) = 1 & o(H^2) = 4 \\ \delta(H^1) = 1 - 1 = 0 & \delta(H^2) = 5 - 1 = 4 \end{array}$$

Teorema fundamental de los algoritmos genéticos

Este teorema estudia la evolución temporal de los esquemas con el fin de analizar el comportamiento de los algoritmos genéticos. Estudia la evolución de un esquema genético H con el tiempo, con este objetivo introduce el número medio de individuos de la población $A(t)$ pertenecientes al esquema H como $m(H, t)$.

Según se describió en la subsección anterior si hay n individuos en la generación t, es decir, $|A(t)| = n$, la probabilidad de que sobreviva en la siguiente generación el individuo i-ésimo con aptitud f_i es

$$p_i = \frac{f_i}{\sum_j f_j}$$

Se denomina $f_T = \sum_{i=1}^{n} f_i$ la suma total de aptitudes de la generación t-ésima.

Considerando inicialmente sólo el operador de selección, que genera un número aleatorio para cada individuo de la población $A(t)$, se investiga cuántos de los $m(H, t)$ individuos del esquema H sobrevivirán en la siguiente generación:

$$m(H, t+1) = \sum_{i=1}^{n} \{Prob\{i - \acute{e}simo\ individuo\ generado \in H\}\} \tag{1.5.21}$$

La probabilidad de generar un individuo del esquema H es

$$\frac{\sum_{j\in H} f_j}{f_T} \tag{1.5.22}$$

y esta probabilidad es independiente del índice i, por lo que la expresión anterior es

$$m(H, t+1) = n\frac{\sum_{j\in H} f_j}{f_T} \tag{1.5.23}$$

Introduciendo la aptitud media de cada individuo de la población $\bar{f}$ y la aptitud media de cada individuo del esquema H como $f(H)$

$$\bar{f} = \frac{\sum_{i=1}^{n} f_i}{n} = \frac{f_T}{n} \tag{1.5.24}$$

$$f(H) = \frac{\sum_{j\in H} f_j}{m(H, t)} \tag{1.5.25}$$

se concluye

$$m(H, t+1) = n\frac{m(H, t)f(H)}{f_T} = m(H, t)\frac{f(H)}{\bar{f}} \tag{1.5.26}$$

Dicho de otra forma, un esquema cuya aptitud media supere a la media de la generación incrementará su presencia en las generaciones futuras, por lo que sobrevivirán los esquemas más aptos.

Para analizar el operador de cruce, se consideran los esquemas siguientes:

$$H = (*1 * | * * * 0) \qquad H' = (* * *|10 * *)$$

y se supone que el cruce se realiza con los 3 primeros bits y los 4 últimos.

Obsérvese que el esquema H no sobrevivirá a la siguiente generación cuando se cruce puesto que o mantiene los 3 primeros bits o los 4 últimos, pero nunca ambos. H' sí puede mantenerse en el cruce, pues basta considerar que los 3 primeros bits de H se pueden considerar incluidos en el esquema H'. El problema surge al separar los bits en esquemas con longitud δ muy grande, ($\delta(H) = 5$ y $\delta(H') = 1$).

En los dos esquemas anteriores, venía prefijada la posición del cruce: entre el tercer y cuarto bits; en general, sin embargo se puede seleccionar con probabilidad uniforme entre los $k-1$ caracteres de la cadena. En este caso, la probabilidad de destruir el esquema H será

$$P_D = \frac{\delta(H)}{k-1} = \frac{5}{6}$$

de forma que la probabilidad de superviviencia del esquema será

$$P_S = 1 - P_D = \frac{1}{6}$$

Sea P_c la probabilidad de operar con el operador de cruce. Con esta probabilidad se selecciona aleatoriamente un padre y una madre para ser sustituidos por sus dos hijos. La probabilidad de supervivencia de un esquema H es

$$P_S \geq 1 - P_c \frac{\delta(H)}{k-1} \tag{1.5.27}$$

La desigualdad anterior se justifica al no considerar en la población $A(t+1)$ aquellos hijos que pertenecen al esquema H aún en el caso en que el corte de los bits se produzca entre dos elementos fijos. Por ejemplo, al cruzar $(*11|010*)$ y $(*1*|101*)$. Combinando los operadores selección y cruce se tiene la siguiente expresión

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} (1 - P_c \frac{\delta(H)}{k-1}) \tag{1.5.28}$$

Si se incluye el operador de mutación, que altera cada uno de los caracteres fijos de la cadena con probabilidad P_m, y considerando que en el esquema H hay $o(H)$ caracteres fijos, la probabilidad de mantener este esquema será $(1 - P_m)^{o(H)}$.

Para simplificar, considerando a su vez que $P_m << 1$, se puede suponer la aproximación $(1 - P_m)^{o(H)} \equiv 1 - o(H)P_m$. La expresión anterior queda, por consiguiente

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} (1 - P_c \frac{\delta(H)}{k-1} - o(H)P_m) \tag{1.5.29}$$

que se conoce como *Teorema fundamental de los algoritmos genéticos.*

1.5.3 Búsqueda Tabú

Los orígenes de *Tabú Search* pueden situarse en diversos trabajos publicados hace alrededor de 20 años. Oficialmente, el nombre y la metodología fueron introducidos posteriormente por Fred Glover, (1989). Numerosas aplicaciones han aparecido en la literatura, así como artículos y libros para difundir el conocimiento teórico del procedimiento, (ver [54]).

Tabú Search es una técnica para resolver problemas combinatorios de gran dificultad que está basada en principios generales de Inteligencia Artificial. En esencia es un metaheurístico que puede ser utilizado para guiar cualquier procedimiento de búsqueda local en la búsqueda agresiva del óptimo local. Por agresiva nos referimos a la estrategia de evitar que la búsqueda quede atrapada en un óptimo local que no sea global. A tal efecto, la *búsqueda tabú* toma de la inteligencia artificial el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente.

Con el fin de introducir esta metaheurística, se estudia un ejemplo de optimización combinatoria que es resuelto con un algoritmo de búsqueda tabú.

Ejemplo 1.5.4 *El problema a resolver es el de determinar la permutación óptima de* 7 *módulos con el fin de conseguir un máximo aislamiento. Se supone que para cualquiera de las* 7! *posibles soluciones se conoce el valor del aislamiento, que es la función objetivo a maximizar.*

El esquema general de la búsqueda tabú es el siguiente: A partir de una solución inicial, se determina una próxima o vecina que la mejore hasta llegar a una solución aceptable. Se denota por $N(s) \subset X$ al conjunto de vecinos de la solución $s \in X$ del problema de optimización. En la búsqueda tabú conviene hablar más que de vecinos de una solución del conjunto de movimientos que permite construir una solución a partir de otra.

Definición 1.5.5 *Dada una solución $s \in X$, se define un movimiento como un elemento $m \in M(s)$ de forma que permite construir una solución vecina:*

$$s' = s \oplus m \in N(s) \quad \forall m \in M(s)$$

En el Ejemplo 1.5.4, dada la permutación inicial

$$s = (2, 5, 7, 3, 4, 6, 1)$$

se caracterizan los movimientos a partir de combinaciones de 2 elementos de 7 módulos, de forma que

$$M(s) = \{(i, j) \ / \ 1 \leq i < j \leq 7\} \quad \forall s \in X$$

Se consideran dos soluciones próximas o vecinas cuando intercambian dos módulos. Si por ejemplo, $m = (4, 5)$ entonces

$$s' = s \oplus m = (2, 4, 7, 3, 5, 6, 1)$$

Definición 1.5.6 *Dado un movimiento $m \in M(s)$, se define su valor y se denota por $v(m)$ a la mejora que se produce en la valoración de la solución vecina obtenida por dicho movimiento:*

$$f(s') = f(s \oplus m) = f(s) + v(m) \quad \forall m \in M(s) \quad \forall s \in X$$

A continuación se aplica al Ejemplo 1.5.4 un esquema de maximización local a partir de la solución inicial s.

- **Iteración 0**: *Sea la solución inicial $s = (2, 5, 7, 3, 4, 6, 1)$ con función objetivo 10. Los mejores movimientos, (hay un total de 21), ordenados por su valor son:*

m	$(4,5)$	$(4,7)$	$(3,6)$	$(2,3)$	$(1,4)$
$v(m)$	6	4	2	0	-1

  Se elige el mejor movimiento como el de máximo valor, en este caso, $m = (4, 5)$. Se construye la solución $s = s \oplus m = (2, 4, 7, 3, 5, 6, 1)$.

- **Iteración 1**: *Sea la solución $s = (2, 4, 7, 3, 5, 6, 1)$ con función objetivo 16 obtenida con el movimiento anterior a partir de la solución inicial. Los mejores movimientos, ordenados*

por su valor, son:

m	$(1,3)$	$(2,3)$	$(3,6)$	$(1,7)$	$(1,6)$
$v(m)$	2	-1	-1	-2	-4

Se elige el movimiento $m = (1,3)$. Se construye la solución $s = s \oplus m = (2,4,7,1,5,6,1)$.

- **Iteración 2**: *Sea la solución* $s = (2,4,7,1,5,6,3)$ *con función objetivo* 18. *Los mejores movimientos son:*

m	$(1,3)$	$(2,4)$	$(6,7)$	$(4,5)$	$(3,5)$
$v(m)$	-2	-4	-6	-7	-9

Obsérvese que la situación es de un mínimo local respecto a cualquier movimiento de los definidos anteriormente. Entre estos movimientos, el mejor es el primero, pero reproduce la solución anterior, por lo que se entraría en un ciclo. Además el movimiento $(4,5)$ *se utilizó en la iteración anterior.*

En un proceso de búsqueda constituido por una sucesión de movimientos, parece razonable el exigir que no se repitan los movimientos en un período suficientemente corto. Para ello hay que registrar los movimientos que han modificado las soluciones anteriores.

De ahí el término de *memoria a corto plazo* que identifica esta metaheurística que prohíbe determinados movimientos utilizados en las últimas iteraciones. La memoria de los últimos movimientos trata de evitar repeticiones sistemáticas en el recorrido por el espacio de soluciones para diversificar así el esfuerzo de búsqueda.

Se introduce así el siguiente concepto que da nombre a la metaheurística.

Definición 1.5.7 *Un movimiento* $m \in M(s)$ *se denomina tabú y se denota por* $m \in T(s)$, *cuando no interese utilizarlo en las próximas iteraciones. Se denomina duración al número de iteraciones que un movimiento tabú no puede ser utilizado.*

En el ejemplo se considerará una duración igual a 3. Con el fin de poder aplicar el algoritmo de búsqueda tabú se introduce una estructura de datos tabú para conocer en cada iteración el número de iteraciones tabúes que le quedan a cada movimiento. En el Ejemplo 1.5.4, se considera la matriz triangular superior, donde el valor de la fila i y la columna j, $(i < j)$ especifica el número de iteraciones que al movimiento (i,j) le quedan prohibidas. Obviamente, si no es un movimiento tabú, el valor del elemento (i,j) de la matriz es 0.

Si se considera una duración de 3 iteraciones, la secuencia de la iteraciones anteriores de la tablas tabúes es la siguiente:

- **Iteración 0**: *La tabla de datos tabú es nula indicando que no hay movimientos tabúes:*

	2	3	4	5	6	7
1	·	·	·	·	·	·
	2	·	·	·	·	·
		3	·	·	·	·
			4	·	·	·
				5	·	·
					6	·

- **Iteración 1**: *En la primera iteración la tabla de datos tabúes quedará:*

	2	3	4	5	6	7
1	·	·	·	·	·	·
	2	·	·	·	·	·
		3	·	·	·	·
			4	3	·	·
				5	·	·
					6	·

- **Iteración 2**:

	2	3	4	5	6	7
1	·	3	·	·	·	·
	2	·	·	·	·	·
		3	·	·	·	·
			4	2	·	·
				5	·	·
					6	·

Obsérvese que el movimiento $(4,5)$ *tiene ahora sólo* 2 *iteraciones de prohibición, mientras que el movimiento* $(1,3)$ *tiene* 3 *iteraciones. Siguiendo con el desarrollo del ejemplo, en la iteración* 2*, los mejores movimientos, ordenados por su valor, y considerando su cualidad de tabúes o no, son:*

m	$(1,3)$	$(2,4)$	$(6,7)$	$(4,5)$	$(3,5)$
$v(m)$	-2	-4	-6	-7	-9
$\in T(s)$	T			T	

Se escoge entonces el movimiento $m = (2,4)$ *por ser el mejor no prohibido. Se construye la solución* $s = s \oplus m = (4,2,7,1,5,6,3)$.

- **Iteración 3**: *Sea la solución* $(4, 2, 7, 1, 5, 6, 3)$ *con función objetivo* 14. *La tabla de datos tabú es ahora:*

	2	3	4	5	6	7
1	·	2	·	·	·	·
	2	·	3	·	·	·
		3	·	·	·	·
			4	1	·	·
				5	·	·
					6	·

Los mejores movimientos, ordenados por su valor, son:

m	$(4,5)$	$(3,5)$	$(1,7)$	$(1,3)$	$(2,6)$
$v(m)$	6	2	0	-3	-6
$\in T(s)$	T			T	

Obsérvese que ahora el mejor movimiento es uno tabú. No obstante, este movimiento conseguirá una función objetivo superior a la obtenida hasta este momento. Es conveniente, pues, relajar la prohibición y elegir este movimiento tabú.

Con el fin de permitir movimientos prohibidos para construir soluciones mejores, se introduce el siguiente concepto.

Definición 1.5.8 *Un movimiento tabú* $m \in T(s)$*, puede ser aceptado si verifica ciertas restricciones impuestas por la función de aspiración, que depende de la valoración de la solución actual* $f(s)$ *y se denota por* $A(f(s))$*:*

$$f(s \oplus m) > A(f(s))$$

Ejemplos de funciones de aspiración son las siguientes:

1. $A(z) = z$: Con esta función de aspiración se aceptan movimientos tabúes cuando la solución obtenida por el movimiento sea mejor a la anterior.

2. $A(z) = f(s^*)$: Con esta función de aspiración se aceptan movimientos tabúes que mejoren la mejor solución obtenida hasta el momento $s^* \in X$.

En algunos casos, y con el fin de incorporar movimientos que consigan soluciones sustancialmente diferentes a las analizadas hasta el momento, conviene registrar de alguna forma aquellos movimientos realizados hasta la iteración presente.

Al registrar los movimientos anteriores y no sólo los últimos, idea que recogían los movimientos tabúes, se denomina *memoria a largo plazo* la propiedad que recoge el siguiente concepto.

Definición 1.5.9 *Se llama frecuencia de un movimiento m al número de veces que se ha aplicado hasta la iteración actual.*

Se incorpora así en la búsqueda tabú una estrategia de diversificación. Con el fin de utilizar la frecuencia, se amplía la matriz de datos tabúes incorporando en su triángulo inferior la frecuencia absoluta de los movimientos hasta la iteración actual.

En la iteración última del Ejemplo 1.5.4, sería la matriz:

	1	2	3	4	5	6	7
1	◇	·	·	3	·	·	·
2		◇	·	·	·	·	·
3	3		◇	·	·	2	·
4	1	5		◇	·	·	1
5		4			◇	·	·
6			1			◇	·
7	2			3			◇

Hay varias formas de penalizar aquellos movimientos que han sido utilizados más a menudo en iteraciones anteriores. Se denotará por $p(m)$ la penalización de cada movimiento $m \in M(s)$.

En el Ejemplo 1.5.4, simplemente se ha restado al valor del movimiento la frecuencia absoluta observada para obtener así un valor penalizado que servirá para elegir el movimiento:

m	$(1,4)$	$(2,4)$	$(3,7)$	$(1,6)$	$(6,5)$
$v(m)$	3	-1	-3	-5	-4
$p(m)$	1	5	0	0	2
$v(m)-p(m)$	2	-6	-3	-5	-6
$\in T(s)$	T				

La estrategia de diversificación considera la evolución del proceso de búsqueda con el fin de evitar aquellos movimientos más utilizados. De forma análoga, se puede introducir una estrategia de intensificación que favorezca la elección de movimientos con mejores valores.

Implementación del método

El esquema general de un algoritmo de búsqueda se podría ver de la siguiente manera:

Inicialización: $\mathtt{s} \in \mathtt{X}$; $\mathtt{s}^* \leftarrow \mathtt{s}$; $\mathtt{k} \leftarrow \mathtt{0}$;
Proceso:

```
[1]   do
[2]     {k ← k + 1;
[3]      Generar V* ⊂ N(s, k);/ * subconjunto de soluciones vecinas * /
[4]      Elegir la mejor s' ∈ V*;
```

```
[5]          s ← s′;
[6]          if (f(s′) <f(s*))  s*← s′;
[7]        }while (Condición parada);
```

El esquema de un algoritmo de búsqueda tabú generaliza el algoritmo anterior incorporando las características introducidas previamente. Concretamente, se tienen las siguientes características:

1. La definición del conjunto $N(s,k)$ identifica el proceso de búsqueda tabú:

$$N(s,k) = N(s) - T$$

donde T representa el conjunto de las últimas soluciones obtenidas y a las que se llega con un movimiento tabú.

2. Formulación de los movimientos. En lugar de trabajar con el conjunto de vecinos de una solución concreta s, es más eficaz trabajar con movimientos válidos, de forma que una iteración se representa por:

$$s' = s \oplus m \quad m \in M(s)$$

y el conjunto de vecinos se define según:

$$N(s) = \left\{ s' \in X \ / \ \exists m \in M(s) \wedge s' = s \oplus m \right\}$$

3. Función de aspiración. Para poder aceptar movimientos tabúes cuando, por ejemplo, permitan alcanzar valores de la función objetivo mejores que los obtenidos hasta el momento. La actualización de la función de aspiración permite flexibilizar la búsqueda variando el nivel de prohibición según evoluciona el algoritmo. Una función de aspiración $A(z) = \infty$ no relajaría nunca la prohibición de los movimientos tabúes.

Así, el esquema general sería el siguiente:

Inicialización: s ∈ X; s*← s; k ← 0; T ← ∅; A(z) ← z;
Proceso:

```
[1]      do                              /* Movimientos válidos */
[2]        {M(s,k) ← {m ∈ M / m ∉T ∨ f(s ⊕ m) <A(f(s));
[3]         V*← {s′∈ X / s′= s ⊕ m,   m ∈ M(s,k)}
[4]         Elegir la mejor s′∈ V*;
[5]         if (f(s′) <f(s*))  s*← s′;
[6]         Actualizar T y A;
[7]         k ← k + 1;
[8]         s ← s′;
[9]       }while (Condición parada);
```

1.5.4 El Ant System

Según se indica en [9], *Ant System* es el primer algoritmo desarrollado en el área *ACO: Ant Colony Optimizacion.* En ésta se estudian sistemas artificiales que simulan colonias de hormigas reales, de donde toman su inspiración. Estos sistemas son utilizados para resolver problemas de optimización combinatoria, los cuales pueden ser descritos como problemas cuyo objetivo es encontrar la secuencia óptima de sus elementos componentes, como por ejemplo el problema *TSP*. El algoritmo *Ant System*, desarrollado por Marco Dorigo, (ver [34] y [33]), es utilizado para resolver este tipo de problemas.

Inspiración biológica

Las hormigas son capaces de encontrar el camino más corto desde el hormiguero a una fuente de comida y viceversa sin usar pistas visuales. Asimismo, son capaces de adaptarse a cambios en el ambiente. Por ejemplo, que un obstáculo sea colocado en la ruta que están utilizando como la más corta, como se ilustra en la Figura 1.4. El medio por el que las hormigas logran esto es por rastreo de la feronoma que ellas mismas depositan mientras caminan.

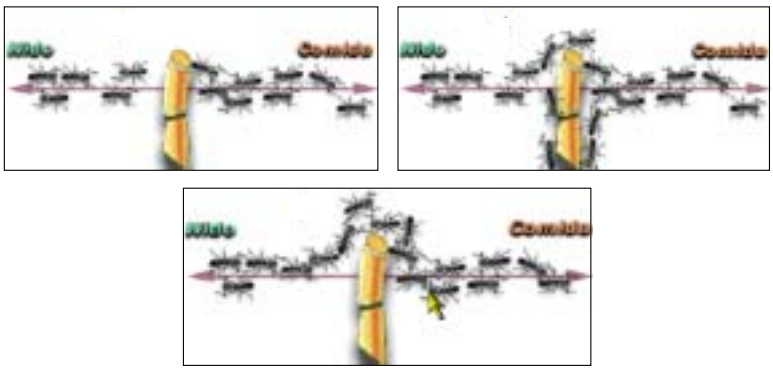


Figura 1.4: Comportamiento Adaptativo de las Hormigas

Todas las hormigas depositan una cierta cantidad de una sustancia llamada feromona mientras caminan y a su vez, cada hormiga prefiere caminar en una dirección rica en feronoma. Esta simple conducta de las hormigas explica porqué son capaces de ajustarse a cambios en el ambiente. Cuando un obstáculo inesperado es colocado en el camino que las hormigas están utilizando, las hormigas que están justo enfrente del obstáculo no pueden continuar siguiendo el rastro de feromona y por tanto, deben elegir sobre irse hacia la izquierda o hacia la derecha. La elección es aleatoria, es decir, cada hormiga decide al azar hacia donde irse, pero se espera que aproximadamente la mitad de las hormigas intente evadir el obstáculo por un lado y la otra mitad lo haga por el otro, (ver Figura 1.4). De esta forma, las hormigas que eligieron, (aleatoriamente), el camino corto, crearán en un cierto tiempo un depósito de feromona más fuerte que el de las hormigas que eligieron el camino largo. De tal forma, pasarán más hormigas por el camino corto, (debido a que llegan al otro lado más rápido que las otras), quedando depositada, por lo tanto, más feromona en esa ruta, lo que origina que las hormigas que vienen atras, prefieran

caminar por ella, reestableciéndose así, el camino más corto, (ver [32]).

Resulta obvio que encontrar la ruta más corta es una conducta que parece ser emergente de la interacción entre el obstáculo y la conducta distribuida de las hormigas, aún cuando todas las hormigas caminan aproximadamente a la misma velocidad y depositan también, aproximadamente, la misma cantidad de feromona.

El algoritmo

La conducta de las colonias de hormigas es imitada por el algoritmo usando agentes sencillos, llamados *hormigas (ants)*, que se comunican indirectamente por medio de un mecanismo inspirado en el rastro de feromona. Los rastros de feromona artifical, son un tipo de información numérica distribuida que es modificada por las *hormigas* y refleja su experiencia en la solución de un problema en particular.

Hay tres ideas que el algoritmo de la colonia de hormigas ha adoptado de las colonias reales de hormigas:

- Se utiliza comunicación indirecta a través de la feromona.
- Las rutas más cortas tienden a tener una razón más alta de crecimiento del valor de la feromona.
- Las hormigas tienen preferencia probabilísitca por las rutas con valores altos de feromona.

Además de estas características, se les ha dado a los agentes, (*hormigas)*, capacidades que no tienen las hormigas reales, pero que ayudan a resolver los problemas. Por ejemplo:

- Cada hormiga es capaz de determinar lo lejos que está de un estado.
- Poseen información acerca de su ambiente y la utilizan al tomar decisiones. Así, su comportamiento no sólo es adaptativo sino también *"voraz"*, (GRASP[1])
- Tienen memoria, la cual es necesaria para asegurar que se generen sólo soluciones factibles, (por ejemplo, en el *TSP*, necesitan saber las ciudades que han visitado pues una ciudad no se puede visitar más de una vez).

Descripción

Veamos un algoritmo *Ant System* para resolver el problema del viajante, (*TSP)*, con n ciudades y distancias d_{ij} entre cada par de ciudades i y j:

1. Las hormigas artificiales se distribuyen inicialmente en las n ciudades de acuerdo a algún criterio, (aleatoriamente, por ejemplo).

[1] Los métodos GRASP son algoritmos por reforzamiento, en los cuales siempre se escoge la acción cuya recompensa estimada se la más alta, (ver subsección 1.5.5 y [64]).

2. Posteriormente cada hormiga decide la ciudad a visitar en cada paso de un ciclo que se repite hasta que todas las ciudades son visitadas exactamente una vez por cada hormiga. La decisión de qué ciudad visitar, se toma con base a la siguiente expresión, donde p_{ij} denota la probabilidad de que una ciudad j sea seleccionada para visitarse después de la ciudad i.

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}\ \eta_{ij}^{\beta}}{\sum\limits_{h\in\Omega} \tau_{ih}^{\alpha}\ \eta_{ih}^{\beta}} & si\ j \in \Omega \\ 0 & en\ otro\ caso \end{cases} \tag{1.5.30}$$

Siendo $\eta_{ij} = \dfrac{1}{d_{ij}}$, y:

- τ_{ij} : cantidad de feronoma entre las ciudades i y j.
- α : parámetro que regula la influencia de τ_{ij}.
- β : parámetro para regular la influencia de η_{ij}.
- Ω : conjunto de ciudades que aún no han sido visitadas.

3. Cuando se termina el ciclo, (cada hormiga ha concluido su recorrido), se calcula la longitud del recorrido generado por cada hormiga y se actualiza el mejor recorrido encontrado hasta el momento.

4. Finalmente, se actualizan las cantidades de feromona. Esto se hace con la siguiente fórmula:

$$\tau_{ij} = p\tau_{ij} + \Delta\tau_{ij} \tag{1.5.31}$$

siendo $\Delta\tau_{ij}$ el incremento total de feromonas en esta iteración, que se calculará según:

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{1.5.32}$$

tomando $\Delta\tau_{ij}^{k}$ como el incremento de feromona realizado por la hormiga k en (i,j).

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{L_k} & si\ la\ hormiga\ viajó\ por\ el\ eje\ (i,j) \\ 0 & en\ otro\ caso \end{cases} \tag{1.5.33}$$

donde:

- $p \in [0,1]$: es un parámetro para regular la reducción de τ_{ij}.
- $\Delta\tau_{ij}$: es el incremento en la cantidad de feromona en la arista (i,j).
- m : es el número de hormigas.
- $\Delta\tau_{ij}^{k}$: es el incremento en la cantidad de feromona en la arista (i,j) realizado por la hormiga k.

- Q : es un valor constante que representa la cantidad de feromona depositada por una hormiga en cada recorrido.
- L_k : es la longitud del recorrido de la hormiga k.

En cada recorrido, cada hormiga deja una cantidad de feromona dada por $\frac{Q}{L_k}$, donde Q es una constante y L_k la longitud de su recorrido. Por lo tanto, en recorridos más cortos se deposita más feromona. En el algoritmo se simula la evaporación de feromona que sucede en la realidad. Las cantidades de feromona se reducen con un factor p antes de que se deposite nueva feromona. Esto se hace para evitar convergencia prematura.

Los pasos anteriores se repiten t veces, donde t es el número de iteraciones.

Pseudocódigo

El pseudocódigo del algoritmo *Ant System* original es el siguiente, ([19]):

```
[1]  Inicio
[2]    For t = 1 to Max_Iter do  //Max_Iter es el número de iteraciones
[3]     For k = 1 to m do  //m es el número de hormigas(agentes)
[4]        Repetir hasta que la hormiga k complete su recorrido
[5]          Seleccionar la siguiente ciudad que va a visitar;
[6]        Calcular la longitud del recorrido de la hormiga k;
[7]     Actualizar los niveles de feromona;
[8]  Fin.
```

1.5.5 Procedimientos GRASP

Los métodos *GRASP* fueron desarrollados al final de la década de los 80 con el objetivo inicial de resolver problemas de cubrimientos de conjuntos, (Feo y Resende, 1989). El término GRASP fue introducido por Feo y Resende en 1995, (ver [86]), como una nueva técnica metaheurística de propósito general.

GRASP es un procedimiento iterativo en donde cada paso consiste en una fase de *construcción* y una de *mejora*. En la fase de construcción se aplica un procedimiento heurístico constructivo para obtener una buena solución inicial. Esta solución se mejora en la segunda fase mediante un algoritmo de búsqueda local. La mejor de todas las soluciones examinadas se guarda como resultado final.

La palabra *GRASP* proviene de las siglas de *Greedy Randomized Adaptive Search Procedures* que en español sería al así como: *Procedimientos de Búsqueda basados en funciones Voraces Aleatorizadas Adaptativas*. Veamos los elementos de este procedimiento.

En la fase de construcción se construye iterativamente una solución posible, considerando un elemento en cada paso. En cada iteración la elección del próximo elemento para ser añadido a la solución parcial viene determinada por una función greedy. Esta función mide el beneficio de añadir cada uno de los elementos según la función objetivo y elegir la mejor. Notar que esta medida es miope en el sentido que no tiene en cuenta qué ocurrirá en iteraciones sucesivas al realizar una elección, sino únicamente en esta iteración. Se dice que el heurístico greedy se

adapta porque en cada iteración se actualizan los beneficios obtenidos al añadir el elementos seleccionado a la solución parcial. Es decir, la evaluación que se tenga de añadir un determinado elemento a la solución en la iteración j no coincidirá necesariamente con la que se tenga en la iteración $j+1$.

El heurístico es aleatorizado porque no selecciona el mejor candidato según la función greedy adaptada sino que, con el objeto de diversificar y no repetir soluciones en dos construcciones diferentes, se construye una lista con los mejores candidatos de entre los que se toma uno al azar.

Al igual que ocurre en muchos métodos deterministas las soluciones generadas por la fase de construcción de *GRASP* no suelen ser óptimos locales. Dado que la fase inicial no garantiza la optimalidad local respecto a la estructura de entorno en la que se esté trabajando, (notar que hay selecciones aleatorias), se aplica un procedimiento de búsqueda local como postprocesamiento para mejorar la solución obtenida.

En la fase de mejora se suele emplear un procedimiento de intercambio simple con el objeto de no emplear mucho tiempo en esta mejora. Notar que *GRASP* se basa en realizar múltiples iteraciones y quedarse con la mejor, por lo que no es especialmente beneficioso para el método el detenerse demasiado en mejorar una solución dada.

Implementación del método

El siguiente esquema muestra el funcionamiento global del algoritmo distinguiendo cada una de sus fases, que se deben repetir hasta verificar una determinada condición de parada.

- **Fase Constructiva**:
  - Seleccionar una lista de elementos candidatos.
  - Considerar una lista restringida de los mejores candidatos.
  - Seleccionar un elemento aleatoriamente de la lista restringida.
- **Fase de Mejora**:
  - Búsqueda local a partir de la solución construida hasta que no se pueda mejorar más.
- **Fase de Actualización**:
  - Si la solución obtenida mejora a la mejor almacenada, actualizarla.

Al realizar muchas iteraciones, *GRASP* es una forma de realizar un muestreo del espacio de soluciones. Las implementaciones *GRASP* generalmente son robustas en el sentido de que es difícil el encontrar ejemplos patológicos en los cuales el método funcione arbitrariamente mal.

Algunas de las sugerencias de los autores para mejorar el procedimiento son: Se puede incluir una fase previa a la construcción: una fase determinista con el objetivo de ahorrar esfuerzo a la fase siguiente. Si se conoce que ciertas subestructuras forman parte de una solución óptima, estas pueden ser el punto de partida de la fase constructiva.

A modo de idea más precisa damos a continuación un pseudocódigo para el método.

Pseudocódigo

Si consideramos s como la semilla para la generación pseudoalatoria de números, el pseudocódigo del algoritmo *GRASP* general es el siguiente, ([53]):

```
[1]  Función GRASP(Max_Iter,s)
[2]    {Leer_Input();
[3]     For k = 1 to Max_Iter do  //Max_Iter es el número de iteraciones
[4]       {Solución←Construcción_Aleatoria_Greedy(s);
[5]        Solución←Búsqueda_Local(Solución);
[6]        Mejorar_Solución(Solución,Mejor_Solución);
[7]       }
[8]     return(Mejor_Solución);
[9]    }
```

Tal y como señalan Feo y Resende una de las características más relevantes de *GRASP* es su sencillez y facilidad de implementación. Basta con fijar el tamaño de la lista de candidatos y el número de iteraciones para determinar completamente el procedimiento. De esta forma se pueden concentrar los esfuerzos en diseñar estructuras de datos para optimizar la eficiencia del código y proporcionar una gran rapidez al algoritmo.

Parte I

Problemas de Visibilidad

Capítulo 2

Introducción

Uno de los campos más estudiados en Geometría Computacional es el de la Visibilidad, es decir, el conjunto de problemas que están relacionados con los conceptos de iluminación y vigilancia [99] en su concepción más general. Existe una gran variedad de problemas en este sentido respecto a la zona a iluminar: polígonos convexos, polígonos monótonos, polígonos generales, semiplanos, etc., y también respecto a los objetos que iluminan: *luces-vértice*, *luces-punto, luces-lado,etc.* Sin embargo, existe la necesidad de añadir a todos los conceptos teóricos de iluminación elementos que permitan la utilización de los resultados obtenidos de forma real o práctica. Por ello, se presentan en esta parte de la memoria resultados de iluminación tomando variaciones de la definición clásica de iluminación, como pueden ser la *visibilidad de alcance limitado* y la $t-$*buena iluminación.*

En la segunda parte de esta memoria se presentan algoritmos aproximados o heurísticos que solucionan distintos problemas de naturaleza $\mathcal{NP}$, entre los que se encuentra el problema ya mencionado MinN-p-Pvk(P). En el estudio de este problema aparece, aunque de forma indirecta, el cálculo del área iluminada a la vez por k focos interiores a un polígono P de n vértices. Este problema que denotaremos como p-Pvk(P, T) se analiza en el Capítulo 5 presentando dos algoritmos para solucionarlo: uno incremental de complejidad $O(kn)$ y otro de doble barrido de complejidad $O(n + k\log(kn))$.

2.1 Conceptos generales de iluminación

En Geometría Computacional uno de los campos más estudiados es el de Visibilidad. Es ésta una disciplina en la que se combinan la combinatoria, la geometría y informática y cuyos resultados tienen aplicaciones en multitud de campos, como la Robótica, Planificación de Trayectorias, Visión por Ordenador, Gráficos y Arquitectura Asistida por Ordenador, [11, 28, 70, 73, 101].

Básicamente la idea de visibilidad mantiene el concepto clásico del mismo. Dado un conjunto D en $\Re^d$ se dice que dos puntos x, $y \in D$ son visibles en D cuando el segmento $\overline{xy}$ está completamente contenido en D. El conjunto D se llamará *convexo* si cualquier par de puntos del mismo se ven. Un conjunto se dice *estrellado* si existe un punto que ve a todos los demás. El

conjunto de puntos con tal propiedad es el núcleo del mismo. Por tanto un conjunto es estrellado si tiene núcleo no vacío.

Podemos extender el concepto de visibilidad de puntos a conjuntos: si U y V son regiones contenidas en D, diremos que U ve fuertemente a V si todo punto de V es visible desde todo punto de U; diremos que U ve débilmente a V si todo punto de V es visible desde algún punto de U.

El concepto de visibilidad clásico en el que aparece el segmento que une dos puntos puede generalizarse o variarse cambiando dicho segmento por otros conjuntos [75, 93, 100]. Asimismo una variante importante del concepto de visibilidad es la visibilidad según cadenas de k eslabones, denominada L_k-visibilidad: dos puntos x e y son L_k-visibles en D si existe una poligonal de k lados contenida en D que conecta ambos puntos [90].

Una parte importante dentro de la Visibilidad es la que se ha venido llamando Galerías de Arte y que se inicia en 1973 con un problema planteado por Victor Klee: determinar el mínimo número de puntos de un polígono suficientes para ver a todos los demás. Considerando el polígono como planta de una galería de arte y los puntos buscados como guardias o focos, tenemos el nombre de esta importante rama de la Geometría Computacional. Chvátal [24] obtiene la primera respuesta en 1975: $\lfloor n/3 \rfloor$ *es el número de guardias a veces necesarias y siempre suficientes para iluminar un polígono P con de n vértices.* Este resultado se conoce como *Teorema de Galerias de Arte* y el primer tratado monográfico fue presentado por O'Rourke en 1987 [80]. Los avances relacionados con el tema se pueden encontrar en [91].

Chvátal basaba su demostración en la casuística. Sin embargo, en 1978 Fisk [42] obtuvo una demostración elegante: Obténgase una triangulación arbitraria del polígono, 3-coloréese el grafo de dicha triangulación y colóquense luces en los vértices que van etiquetados con el color menos frecuente de la coloración.

Por tanto, $\lfloor n/3 \rfloor$ luces son siempre suficientes para iluminar un polígono P de n vértices, pero en muchas ocasiones basta con menos. Así tiene sentido plantear el problema algorítmico siguiente: *dado un polígono calcular el mínimo número de luces que lo vigilan.* Este problema fue estudiado por Lee y Lin [71] y utilizando una reducción al problema 3-Sat [45], probaron que es de naturaleza $\mathcal{NP}$. Si nos restringimos a polígonos ortogonales, Schuchardt y Hecker [92] han demostrado que minimizar el número de *luces punto* y *vértice* que iluminan todo el polígono es también un problema $\mathcal{NP}$-duro. En esta memoria nos hemos planteado la búsqueda de un algoritmo aproximado, (no podría ser de otra manera al ser de naturaleza $\mathcal{NP}$), para la búsqueda del mínimo número de *luces-punto* que iluminan el polígono P. Así, en el Capítulo 8 proponemos utilizando técnicas heurísticas, (*simulated annealing* y *algoritmos genéticos*), soluciones aproximadas a este problema que hemos denotado con MinN-p-Pvk(P). Una solución también aproximada pero para *luces-vértice* fue propuesta en 1987 por Ghosh [49].

Para el problema algorítmico de colocar a lo más $\lfloor n/3 \rfloor$ luces se obtuvo un algoritmo polinómico [5], siendo lineal el algoritmo si consideramos la triangulación de Chazelle [21].

En polígonos ortogonales la cota de luces suficientes para iluminar el polígono se reduce a $\lfloor n/4 \rfloor$ [65] y pueden situarse en tiempo lineal [36].

Respecto a la iluminación de polígonos con agujeros no se ha probado la equivalencia combinatoria entre iluminación con *luces-vértices* y con *luces-punto*. En este sentido, se define $g^P(n,h)$ como el número de *luces-punto* que vigilan todo polígono de n vértices y h agujeros. Para

luces-vértice se define análogamente $g^V(n,h)$. Se ha demostrado que $g^P(n,h) = \lfloor (n+h)/3 \rfloor$ [13, 59]. Si consideramos polígonos ortogonales con agujeros también ortogonales, se ha obtenido que el número de *luces-punto* es $g_o^P(n,h) = \lfloor n/4 \rfloor$ [58] y se conjetura que $g_o^V(n,h) = \lfloor (n+h)/4 \rfloor$.

2.2 Problemas estudiados

En muchas aplicaciones industriales reales relacionadas de alguna manera con la *iluminación* o *vigilancia*, tales como iluminación de calles, de naves industriales, vigilancia de recintos, se pueden utilizar los resultados teóricos que los investigadores han obtenido en los últimos años, en este campo de la *Geometría Computacional*. Sin embargo, la utilización práctica de estos resultados nos lleva a que en muchas ocasiones los elementos físicos reales, que existen en la actualidad, discrepan en algún sentido de los modelos teóricos utilizados. Por ello, se deben añadir a las definiciones clásicas de *iluminación,* elementos que permitan obtener modelos cada vez más cercanos a la realidad y que por tanto permitan, cada vez con mayor frencuencia, su utilización en campos de la ingeniería y la construcción. Presentamos en esta primera parte de la memoria soluciones a dos problemas planteados en este sentido: la *visibilidad de alcance limitado* en el Capítulo 3 y la *$t-$buena iluminación* en el Capítulo 4, que de forma introductoria presentamos a continuación.

Una de las primeras observaciones que podemos tener en este sentido práctico, es que la luz que pueden emitir un foco luminoso pierde intensidad según nos alejamos de dicho foco. Por tanto, cuando nos alejamos de un foco luminoso suficientemente, la intensidad de luz que podemos recibir es muy deficitaria y por tanto no se podría considerar que dicho foco nos ilumina.

Teniendo en cuenta esta limitación, una de las variaciones a la definición clásica de iluminación es la que presenta en 1992 Ntafos en [77], donde se define el concepto de *visibilidad de alcance limitado d*: *"dos puntos son d-visibles si son visibles y su distancia es a lo más d"*, como se ilustra en la Figura 2.1.

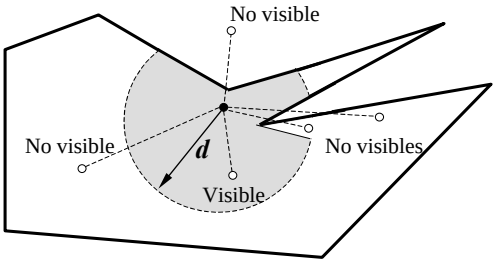


Figura 2.1: Visibilidad de alcance d

Ntafos describe algoritmos aproximados para el problema de la *Ruta del d-Vigilante*, es decir, encontrar un camino cerrado en un polígono P, (el lugar que se vigila), de forma que cada uno de los puntos del borde de P sea d-visible desde algún punto del camino. También se estudia y resuelve de forma aproximada el problema del barrendero con alcance limitado, que es un problema $\mathcal{NP}$-duro, y que consiste en buscar un camino para un barrendero, (que según avanza barre todos los puntos situados a distancia menor que d), que cubra todo el polígono P.

Persiguiendo esta misma idea de la limitación en la visibilidad, en 1995 Sung-Ho Kim y otros [95] presentan un algoritmo de complejidad $O(n)$ para construir el polígono de d-visibilidad desde un lado del polígono como se muestra en la siguiente figura, donde aparece el polígono de d-visibilidad del lado $\overline{v_i v_{i+1}}$ del polígono P.

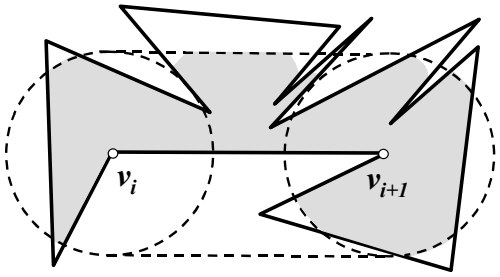


Figura 2.2: Polígono de visibilidad limitada desde un lado

Ese mismo año Jesús García [46] realiza en su tesis doctoral un estudio pormenorizado de la *visibilidad de alcance limitado* en polígonos convexos, polígonos simples y configuraciones de polígonos. Si las luces son de alcance L, García llama L_{min} al valor más pequeño de L para el que colocando luces de alcance L en todos los vértices de P, éste quede totalmente iluminado. Designando por D al diámetro del polígono, el intervalo de valores significativos para el alcance L de las luces situadas en los vértices es $[L_{\min}, D]$. En la tesis se obtienen resultados, tanto combinatorios como algorítmicos, para polígonos convexos y polígonos simples de n vértices. Por ejemplo, el intervalo $[L_{\min}, D]$ se calcula en tiempo $O(n)$ para polígonos convexos y en tiempo $O(n \log n)$ para polígonos simples, probándose que $L_{\min} \leq D/\sqrt{3}$. En cuanto a resultados combinatorios, se prueba para polígonos simples que el mínimo número p de *luces-vértice* que iluminan el polígono son:

$$\begin{cases} p = n & si\ L = L_{\min} \\ p \geq \left\lfloor \frac{3(n-1)}{4} \right\rfloor & si\ L = D/\sqrt{3} \\ p = \left\lfloor \frac{n}{3} \right\rfloor & si\ L = D \end{cases} \tag{2.2.1}$$

Si las luces no se sitúan en los vértices del polígono, el estudio de la visibilidad de alcance L es muy complicado, planteándose en la tesis numerosos problemas que continúan abiertos. Por ejemplo, si se tienen k luces en posiciones dadas en el interior de un polígono P, ¿cuál es el mínimo alcance L para iluminar todo P?, ¿cuál es el mínimo número de esas luces que iluminan P? El único resultado que se presenta en esta línea es un algoritmo para calcular en tiempo $O(k^4 n)$, el alcance mínimo $L_{\min}$. La conjetura es que muchos de los problemas planteados son $\mathcal{NP}$-completos.

Más recientemente, en el año 2001, Ghosh y otros presentan [51] resultados para obtener la región visible por un robot que se mueve en el interior de un polígono P con obstáculos, pero con dos restricciones importantes. La primera de ellas es que el robot tiene una *visibilidad limitada* y la segunda es que la visibilidad solo es posible en un conjunto discreto de puntos sobre la trayectoria que se marque. Nótese que esta segunda condición impone restricciones importantes

sobre las definiciones de visibilidad limitada tratadas hasta ahora.

Continuando con el análisis de la *visibilidad de alcance limitado*, en el Capítulo 3 de esta memoria estudiamos este tipo de iluminación para polígonos *escalera* y polígonos *pirámide*. Se define el concepto de radio r de una escalera y se demuestra en la Sección 3.2 que en polígonos escalera $L_{\min} = \frac{1}{2}r$. Asimismo se ajustan las cotas combinatorias respecto al número de *luces-vértice* que iluminan un polígono escalera P de n vértices, probando que el número de luces a veces necesarias y siempre suficientes para vigilarlo son:

$$\begin{cases} 1 & si\ L = r \\ \lfloor \frac{n}{4} \rfloor + c & si\ \frac{r}{2} \leq L < r \end{cases} \tag{2.2.2}$$

Para finalizar este capítulo se presentan en la Sección 3.3 resultados de alcance limitado para polígonos pirámide. Se demuestra que también para este tipo de polígonos $L_{\min} = \frac{1}{2}r$ y que si $L \geq r$ el mínimo número de *luces-vértice* que iluminan la pirámide es $\lceil \frac{n}{6} \rceil$. Estos problemas combinatorios han sido denotados por CombN-v-Es(P,L) en el caso de los polígonos escalera y CombN-v-Pi(P,L) en el de polígonos pirámide.

El otro tipo de iluminación que busca la utilidad práctica de los resultados es un nuevo tipo de iluminación, que se presenta en el Capítulo 4 de esta memoria por primera vez y que denominaremos $t-$*buena iluminación*. La idea de este tipo de iluminación se basa en considerar que un punto no está bien iluminado si todas las luces que lo iluminan están solamente agrupadas a un "lado" del punto, es decir, si no están, en cierto sentido, bien distribuidas alrededor del punto con una determinada proporcionalidad. Así, si tenemos un punto p en el plano euclídeo y un conjunto $F = \{f_1, ..., f_k\}$ de k focos se dice que "*un punto p está $t-$bien iluminado por F si todo semiplano con borde en p contiene al menos t focos que lo iluminan*". Como se puede observar en la Figura 2.3 el punto p está $1-$*bien iluminado* por los focos f_1, f_2 y f_3, ya que ninguno de los obstáculos evita que cualquier semiplano que pasa por p deje focos a ambos lados del mismo. Sin embargo, el punto q no está $1-$*bien iluminado* ya que dibujando un semiplano con borde en q que deje a los focos f_2 y f_3 a un lado y f_1 al otro, éste no iluminará a q.

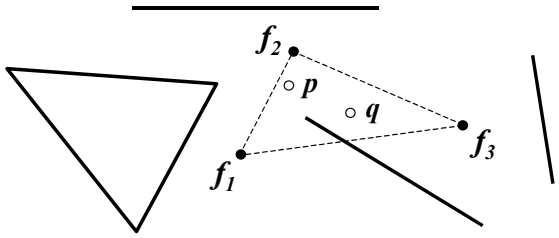


Figura 2.3: Un ejemplo de $1-$buena iluminación

En este capítulo se define la $t-$*buena iluminación* y se presentan algoritmos para su cálculo en diferentes situaciones geométricas, que de forma resumida son las siguientes:

- Cuando tenemos un conjunto $F = \{f_1, ..., f_k\}$ de k focos en el plano euclídeo y no hay ningún obstáculo que dificulte la visión o vigilancia, (problema B$t-$IGenk(F):).

- Cuando tenemos un polígono P de n vértices y un conjunto $F = \{f_1, ..., f_n\}$ de n focos situados en los vértices del polígono, (problema $\mathsf{B}t{-}\mathsf{ICon}(P)$: si P es convexo y $\mathsf{B2}{-}\mathsf{IPol}(P)$: si no lo es, con $t = 2$).
- Cuando tenemos un conjunto $F = \{f_1, ..., f_k\}$ de k focos en el plano euclídeo y un obstáculo convexo C con n vértices que impide la visión, (problema $\mathsf{B1}{-}\mathsf{ICk}(C, F)$).

Para finalizar esta primera parte de la memoria que hemos llamado "*Problemas de Visibilidad*", presentamos en el Capítulo 5 algoritmos para el cálculo de la región iluminada, (ahora sin limitación en el alcance), por un conjunto $F = \{f_1, ..., f_k\}$ de k focos interiores a un polígono P. Este problema que denotaremos con $\mathsf{p\text{-}Pvk}(P, T)$ y para el que se presentan dos algoritmos: uno incremental de complejidad $O(kn)$ y otro de doble barrido de complejidad $O(n + k\log(kn))$.

Capítulo 3

Visibilidad de alcance limitado

La visibilidad clásica utilizada, por ejemplo, para solucionar el problema de Galerías de Arte, ([99]), no introduce el concepto de limitación en el alcance para la vigilancia o iluminación de guardias o luces respectivamente. Esta situación no es en absoluto real, pues los mecanismos de iluminación que se utilizan no tienen un alcance ilimitado.

Por tanto, tiene sentido introducir un concepto de visibilidad o iluminación donde el alcance esté limitado a una distancia L, concepto que se estudia en [46] para polígonos convexos , en [77] para de rutas de vigilancia y en [95] para visibilidad desde un lado del polígono. Formalizaremos a continuación estos conceptos que evidentemente estarán relacionados con el alcance de iluminación L.

En esta situación aparecen preguntas combinatorias y algorítmicas de gran interés, como pueden ser, por ejemplo: ¿Cuál es el alcance mínimo L para que, situando una luz en cada vértice del polígono, se ilumine totalmente? ¿Para qué valores de L se puede iluminar todo el polígono? Dada una longitud L, ¿cuál es el número mínimo de luces vértice de alcance L que iluminan todo el polígono?

En este capítulo respondemos a algunas de estas preguntas para dos tipos concretos de polígonos: *escaleras* y *pirámides*.

3.1 Definiciones

Definiremos en primer lugar lo que se entiende por puntos visibles con alcance L y después daremos algunas definiciones particulares para polígonos *escalera*, que nos ayudaran en el desarrollo de este capítulo respecto a este tipo de polígonos.

Definición 3.1.1 Visibilidad de Alcance *L. Dos puntos x e y se dicen visibles con alcance L en un polígono P, si el segmento xy está contenido completamente en P y la distancia $d(x, y) \leq L$, (ver Figura 3.1).*

Una vez que conocemos el concepto de visibilidad *limitada* o de *alcance limitado* veamos las definiciones propias de los polígonos *escalera*.

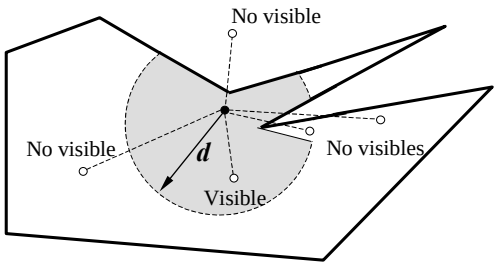


Figura 3.1: Visibilidad de alcance L

Definición 3.1.2 Polígono escalera. *Se denomina polígono escalera a todo polígono ortogonal P, tal que existe un lado horizontal h, (respectivamente vectical v), cuya longitud es igual a la suma de las longitudes de los restantes lados horizontales, (respectivamente verticales). Designaremos con la notación $P(V, a_1,\ a_2,\ ....,\ a_m)$, al polígono escalera en el que V es la intersección de h y v y $a_1,\ a_2,\ ....,\ a_m$ son los restantes vértices convexos.*

Así si llamamos $b_i,\ i = 1, 2, ..., m-3$ al vértice cóncavo cuya abcisa es la abcisa del vértice a_{i-1} y cuya ordenada es la del vértice $a_{i+2,}$ se tiene:

$$\overline{a_1a_2} + \sum_{i=1}^{m-3} \overline{b_ia_{i+2}} = \overline{Va_m} \text{ y } \sum_{i=1}^{m-3} \overline{b_ia_{i+1}} + \overline{a_{m-1}a_m} = \overline{Va_1} \tag{3.1.1}$$

como se muestra en la Figura 3.2.

Definición 3.1.3 Radio de una escalera. *Dado un polígono escalera $P(V, a_1,\ a_2,\ ....,\ a_m)$ se llaman radios exteriores de P y se denotan por r_k a los segmentos $\overline{Va_k}\ \ \forall\ k = 1, 2, ..., m$. Se llama radio de P y se denota por r al máximo de los radios exteriores de P.*

$$r = \max(r_k) \quad \forall k = 1, 2, ..., m. \tag{3.1.2}$$

Para comprender bien la definición anterior podemos ver la Figura 3.2, donde se dibujan los radios exteriores de un polígono *escalera*. Pasemos a estudiar ahora la iluminación de *alcance limitado* en este tipo de polígonos, considerando que las luces se situan sobre los vértices, es decir considerando *luces-vértice* y no *luces punto*.

3.2 Iluminación de Alcance Limitado en Escaleras

El problema combinatorio que planteamos en este capítulo respecto a los polígonos *escalera* consiste básicamente en encontrar el número mínimo de *luces-vértice* necesarias para iluminar el polígono. Evidentemente, dado que estamos estudiando la visibilidad de *alcance limitado*, esta cota dependerá del alcance L de iluminación. Formalmente el problema se puede enunciar de la

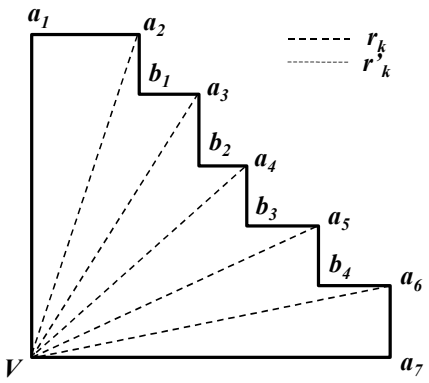


Figura 3.2: Polígono escalera

siguiente manera:

número de luces-vértice de alcance L que iluminan un polígono escalera

CombN-v-Es(P,L):

ENTRADA: Un polígono P de n vértices y un alcance de iluminación L.

PREGUNTA: ¿Cuál es el número mínimo k de *luces-vértice* de alcance L necesarias para iluminar el polígono P?

Es evidente que si el alcance de iluminación L es suficientemente grande el polígono se podrá iluminar con una sola luz. Así si tomamos $L \geq r$, donde r es el radio del polígono, colocando una luz en el vértice V tendremos iluminado todo el polígono. Planteamos ahora otro problema distinto, permitiendo colocar una luz en cada vértice y preguntándonos por el mínimo alcance de iluminación, (L_{min}), necesario para iluminar todo el polígono.

3.2.1 Alcance mínimo

Dado un polígono P, se define L_{min} como el valor más pequeño para el que colocando luces de alcance L_{min} en todos los vértices del polígono P, éste quede totalmente iluminado. Por tanto, el intervalo de valores significativos para el alcance L resulta ser $[L_{min}, r]$. El primer problema que se plantea es el cálculo de L_{min}. Damos respuesta a esta cuestión en la siguiente proposición.

Proposición 3.2.1 *Dado un polígono escalera* $P(V, a_1, a_2, ..., a_m)$ *de radio* r, *se tiene que* $L_{\min} = \frac{1}{2}r$.

Demostración. Resulta claro, observando la Figura 3.3, que si tomamos $L = \frac{1}{2}r - \varepsilon \quad \forall \varepsilon > 0$, siempre podemos encontrar un polígono escalera P_ε que no es posible iluminar colocando una luz de alcance L en cada uno de sus vértices.

Probemos ahora que todo polígono escalera P de radio r se ilumina con luces vértice colocando en cada vértice una luz de alcance $\frac{1}{2}$ r. Para ello descomponemos el polígono en cuñas uniendo cada vértice con V, como en la Figura 3.4 Para iluminar P basta iluminar cada una de esas cuñas. Ahora bien, las cuñas son de dos tipos: aquellas en que uno de sus tres lados es un

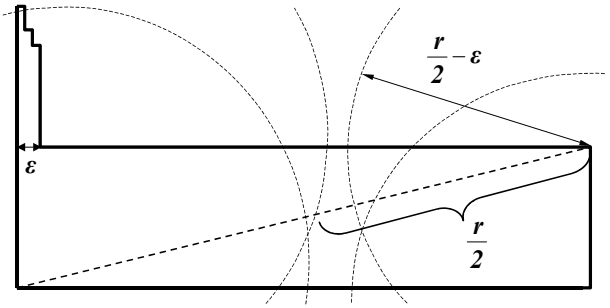


Figura 3.3: Ilustración de P_ε

lado horizontal del polígono y que llamaremos **Tipo H** y aquellas en que uno de sus tres lados es un lado vertical del polígono y que llamaremos **Tipo V**. Tanto en las cuñas de tipo H como en las de tipo V, la longitud del lado más grande de la cuña es como máximo r. Así, si llamamos z al punto medio del lado mayor de cada cuña, la distancia de cada vértice de la cuña al punto z es siempre menor o igual que $\frac{1}{2}\ r$, con lo que probamos que siempre podemos iluminar las cuñas colocando luces de alcance $\frac{1}{2}\ r$ en todos sus vértices y, por tanto, también podemos iluminar todo el polígono. ■

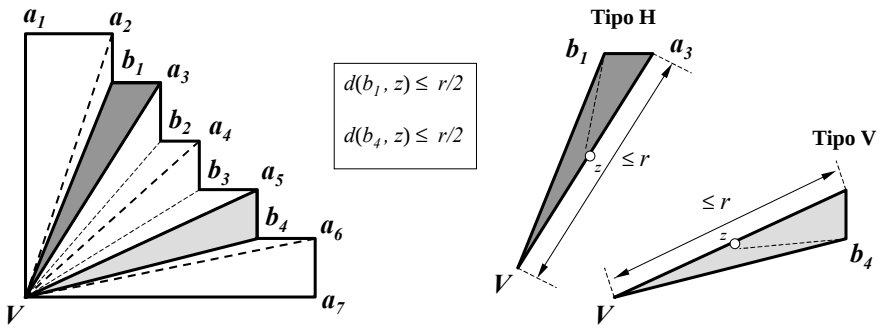


Figura 3.4: Cuñas de un polígono escalera

Ya sabemos, por tanto, que colocando luces de alcance $\frac{1}{2}\ r$ en todos los vértices de P iluminamos todo el polígono. Pero, ¿cuántas luces vértice necesitamos para iluminar todo el polígono con un alcance de iluminación L? Evidentemente esta distancia L debe pertenece al intervalo de valores significativos para el alcance, $[L_{min},\ r]$.

Si $L = r$, podemos iluminar todo el polígono colocando una luz en V. ¿Cuantas luces vértice son necesarias si $L \in [L_{min},\ r)$?

3.2.2 Luces Vértice con alcance L

Todo polígono escalera P con n vértices, se puede iluminar con una sola luz de alcance r. ¿Qué ocurre si disminuimos una cantidad pequeña el alcance L? Esta cuestión la estudiaremos en el Lema 3.2.3, pero previamente analizaremos algunas propiedades que son necesarias para la

demostración del Teorema 3.2.5 que caracteriza el número de luces para iluminar el polígono P en función de L.

Lema 3.2.2 *Los vértices V, a_1 y a_m son necesarios en algunos casos para iluminar el polígono P con luces de alcance $\frac{r}{2}$.*

Demostración. Si nos fijamos en el polígono de la Figura 3.5, es claro que para iluminarlo con una luz de alcance $\frac{r}{2}$, necesitamos al menos colocar luces en los vértices V, a_1 y a_m, ya que en otro caso el polígono no quedaría iluminado. Claramente la situación extrema será el rectángulo en cuyo caso necesitamos como mínimo colocar 4 luces de alcance $\frac{r}{2}$, en cada uno de sus vértices. ■

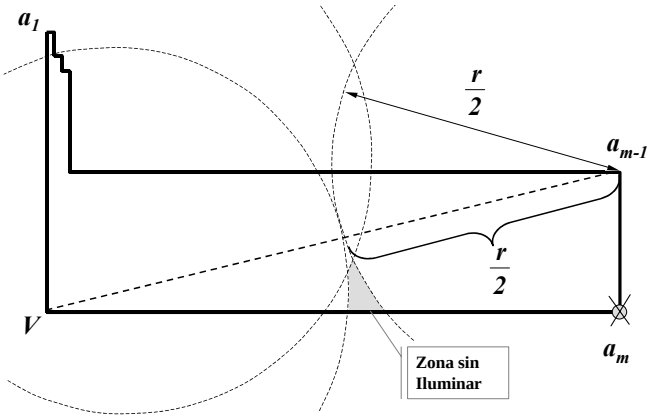


Figura 3.5: Zona sin iluminar desde a_m

Ya hemos visto en el ejemplo anterior que los vértices V, a_1 y a_m son a veces necesarios para iluminar el polígono pirámide P, cuando el alcance de iluminación es $L_{\min}$. En el siguiente lema analizamos que ocurre cuando las luces que situamos en los vértices del polígono son de alcance muy próximo a r, es decir, cuando $L = r - \varepsilon$, con ε tan pequeño como deseemos.

Lema 3.2.3 *Si ε es suficientemente pequeño, $n \geq 8$ y $L = r - \varepsilon$, entonces existe un polígono escalera $P(V, a_1, a_2, ..., a_m)$ de n vértices que necesita $\left\lfloor \frac{n}{4} \right\rfloor + 2$ luces de alcance L para ser iluminado.*

Demostración. Consideramos el polígono de la Figura 3.6, donde todos los vértices convexos a_i, $i = 2, ..., m-1$ de P se encuentran a distancia r de V. Es obvio que desde V no se ilumina el fondo de ninguna cuña si $L = r - \varepsilon$. Desde cada vértice cóncavo se iluminan solo sus cuñas adyacentes, por tanto, necesitamos colocar luces en los vértices cóncavos alternadamente. Así, como el número de vértices cóncavos es $\frac{n}{2} - 2$, el número de luces es:

$$\left\lfloor \frac{1}{2}\left(\frac{n}{2} - 2\right) \right\rfloor + 3 = \left\lfloor \frac{n}{4} \right\rfloor + 2 \tag{3.2.3}$$

donde el 3 corresponde a los vértices V, a_1 y a_m ■

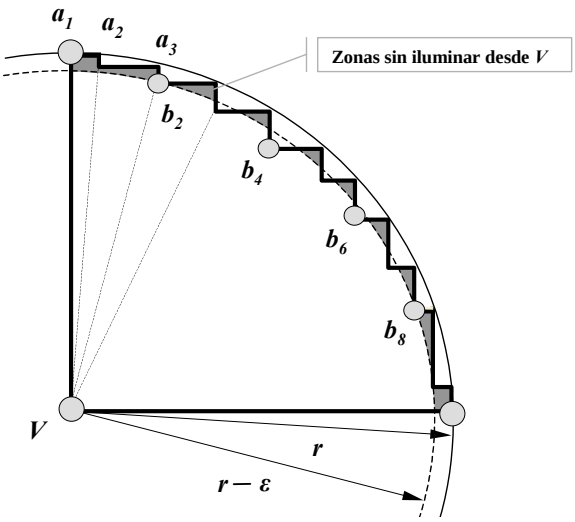


Figura 3.6: Visibilidad desde V con $L = r - \varepsilon$

Podemos preguntarnos ahora por el número de luces necesarias para iluminar un polígono escalera P con n vértices, cuando el alcance de iluminación toma el valor inferior en el intervalo significativo, es decir, cuando $L = \frac{r}{2}$. Demostramos en el siguiente lema que existen polígonos escalera en este caso, que necesitan $\lfloor \frac{n}{4} \rfloor + 4$ luces para ser iluminados.

Lema 3.2.4 *Si $L = \frac{r}{2}$, entonces existe un polígono escalera $P(V, a_1, ..., a_m)$ de n vértices que necesita $\lfloor \frac{n}{4} \rfloor + 4$ luces de alcance L para ser iluminado.*

Demostración. En la Figura 3.7 tenemos un ejemplo de un polígono escalera que se ha construido agrupando sus peldaños de la siguiente manera: tenemos cuatro grupos de peldaños con k vértices cóncavos, siendo k par y tal que el último vértice cóncavo de cada grupo dista más de $\frac{r}{2}$ del primero del siguiente grupo. Cada grupo necesita $\frac{k}{2} + 1$ luces para ser iluminado. Por tanto si tenemos en cuenta que necesitamos situar también una luz en V, el número de focos será:

$$4\left(\frac{k}{2} + 1\right) + 1 = 2k + 5 \tag{3.2.4}$$

Como P tiene $4k$ vértices cóncavos, se verifica que $4k = \frac{n}{2} - 2$, o lo que es igual $2k = \frac{n}{4} - 1$. Sustituyendo en la 3.2.4 tenemos:

$$\frac{n}{4} - 1 + 5 = \frac{n}{4} + 4 \tag{3.2.5}$$

Si para alguno de los tramos el número de cóncavos k es impar, entonces n no es múltiplo de 4, pero como no aumenta el número de luces, resulta que dicho número es $\lfloor \frac{n}{4} \rfloor + 4$. ■

Una vez analizadas las luces a veces necesarias para iluminar P para $L = r - \varepsilon$ y $L = \frac{r}{2}$, ya podemos estudiar ya de forma conjunta el número de luces necesarias y suficientes para iluminar

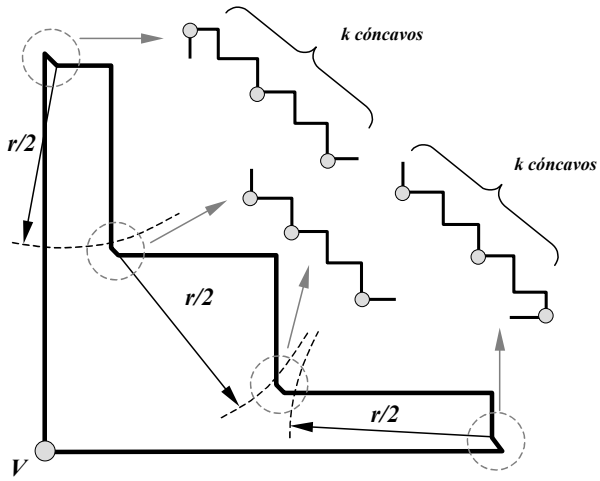


Figura 3.7: P necesita $\lfloor \frac{n}{4} \rfloor + 4$ luces vértice de alcance $L = \frac{r}{2}$

el polígono P. Evidentemente este número depende de L, pues por ejemplo si $L = r$ el número el polígono se iluminará colococando una luz en el vértice V. En el siguiente teorema analizamos el número de luces necesarias y suficientes para iluminar completamente P, en función del alcance de iluminación L.

Teorema 3.2.5 *Para todo polígono escalera P con n vértices, el número de luces vértice de alcance L, a veces necesarias y siempre suficientes para iluminarlo son:*

$$\left\{ \begin{array}{ll} 1 & si \;\; L = r \\ \lfloor \frac{n}{4} \rfloor + c & si \; \frac{r}{2} \leq L < r \end{array} \right\} \tag{3.2.6}$$

Demostración. El caso $L = r$ es evidente colocando una luz en V. En el Lema 3.2.3 hemos visto que al rebajar ligeramente el alcance, podemos llegar a necesitar un número de luces igual a $\lfloor \frac{n}{4} \rfloor + 2$, es decir en este caso $c = 2$. Para terminar la demostración basta comprobar que $\lfloor \frac{n}{4} \rfloor + 4$ luces de alcance $\frac{r}{2}$ son sufientes para iluminar cualquier polígono escalera de n vértices. Por el Lema 3.2.2 tendremos siempre tres luces en los vértices V, a_1 y a_m.

Como se puede observar en la Figura 3.8, colocando luces de alcance $L = \frac{r}{2}$ en V, a_1, a_m y en los vértices cóncavos alternos, (lo que supone un total $\lfloor \frac{n}{4} \rfloor + 2$ luces), puede quedar totalmente iluminado el polígono P.

Sin embargo, esta "buena" situación respecto a la iluminación no siempre se verifica, pudiendo darse casos "desfavorables", que aumentarán el número de luces y que analizamos a continuación.

- **Caso 1**: **Existen vértices cóncavos consecutivos que distan más de $\frac{r}{2}$ o peldaños de anchura o altura superior a $\frac{r}{2}$.**

  Según se muestra en las Figuras 3.7 y 3.9 (a) puede suceder que al colocar luces en los vértices cóncavos alternos no quede totalmente iluminado el polígono, si la distancia entre

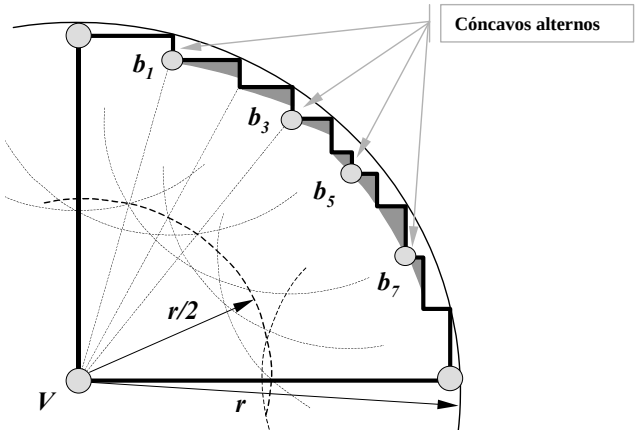


Figura 3.8: Iluminación de P con vértices cóncavos, V, a_1 y a_m

los dos vértices cóncavos que determinan un peldaño es superior a $L = \frac{r}{2}$. Para solucionar esta situación, (que se puede producir como máximo tres veces), es necesario colocar luces en ambos vértices cóncavos, lo que puede determinar dos vértices adicionales. Por otra parte, como se muestra en la Figura 3.9 (b), si un peldaño de P tiene anchura o altura superior a $\frac{r}{2}$, puede suceder que necesitemos también colocar una luz en el vértice cónvexo de dicho peldaño. Sin embargo, esta situación sólamente se puede presentar una vez.

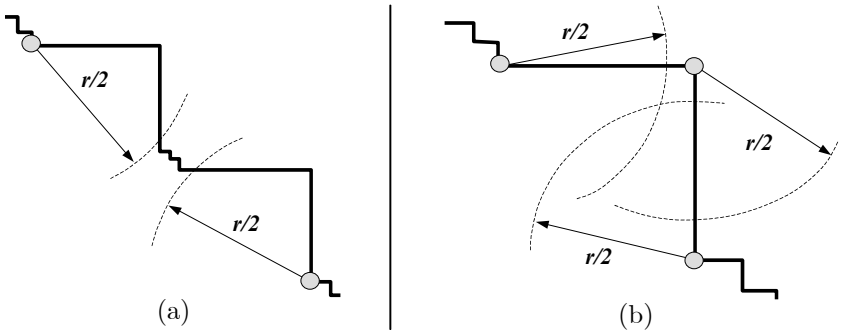


Figura 3.9: Situaciones de peldaños desfavorables

Por tanto, el número de luces de alcance $\frac{r}{2}$ suficientes para iluminar P es:

$$\left\lfloor \frac{n}{4} \right\rfloor + 2 + 2 = \left\lfloor \frac{n}{4} \right\rfloor + 4 \tag{3.2.7}$$

- **Caso 2**: **Existen zonas interiores a P sin iluminar desde cóncavos alternos.**

  En este caso, como se muestra en la Figura 3.10, pudiera suceder que colocando luces de alcance $\frac{r}{2}$ en V y en dos vértices cóncavos alternos, b_k y b_{k+2}, no se iluminara una zona de P. Colocando una luz en el vértice intermedio b_{k+1} y colocando alternadamente luces en los vértices cóncavos a partir de él, quedará iluminado el polígono. ¿Cuántas veces se

puede dar la situación anterior? En el peor de los casos esta situación se repetirá dos veces y podrá producir que necesitemos colocar una luz más en un vértice cóncavo, por el orden de colocación de luces en dichos vértices. Por tanto el número de luces suficientes para iluminar sigue siendo:

$$\left\lfloor \frac{n}{4} \right\rfloor + 2 + 1 = \left\lfloor \frac{n}{4} \right\rfloor + 3 \tag{3.2.8}$$

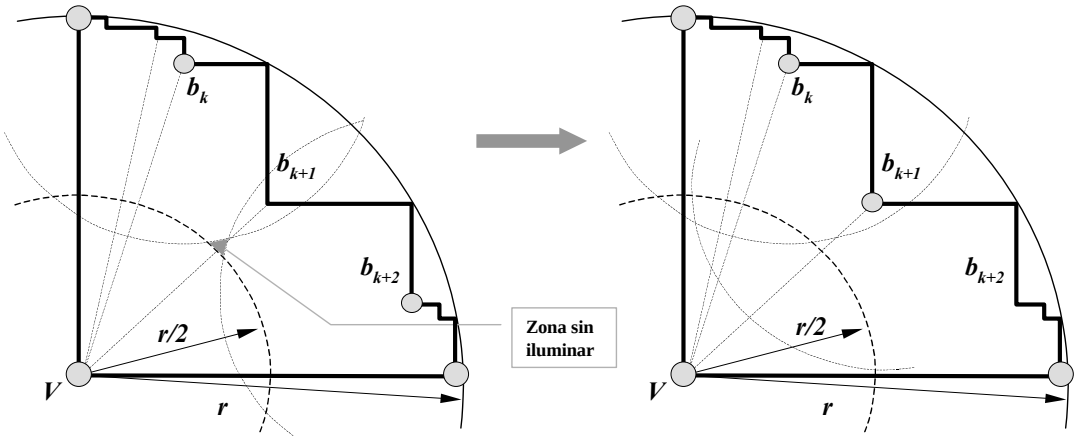


Figura 3.10: Existen zonas interiores a P sin iluminar

Por otra parte podemos tener situaciones en las que tengamos un peldaño desfavorable del Caso 1 y otro del Caso 2, pero sólamente se pueden dar uno de cada situación, con lo que el número de luces suficientes para iluminar P seguirá siendo $\lfloor \frac{n}{4} \rfloor + 4$. ■

Concluido el estudio para polígonos *escalera* realizamos a continuación un estudio similar para polígonos *pirámide* que en cierto sentido se puede considerar doble *pirámides* o generalizaciones de éstas.

3.3 Iluminación de Alcance Limitado en Pirámides

Un polígono *pirámide* puede considerarse una generalización de un polígono *escalera*. Por tanto, tiene sentido plantearse el mismo problema combinatorio estudiado para escaleras. Formalmente el problema que nos planteamos en esta sección es el siguiente:

número de luces-vértice de alcance L que iluminan un polígono pirámide

CombN-v-Pi(P,L):
ENTRADA: Un polígono P pirámide de n vértices y un alcance de iluminación L..
PREGUNTA: ¿Cuál es el número mínimo k de *luces-vértice* de alcance L necesarias para iluminar el polígono P?

Para atacar este problema necesitamos también un conjunto de definiciones similares a las

dadas para polígonos *escalera*, tales como el radio de una *pirámide*. Estas definiciones se exponen en la sección siguiente.

3.3.1 Definiciones

Definición 3.3.1 Polígono pirámide. *Se denomina polígono pirámide a todo polígono ortogonal tal que existe un lado horizontal cuya longitud es igual a la suma de las longitudes de los restantes lados horizontales. Si* $a_1, ..., a_m$ *son los vértices convexos de un polígono pirámideP, lo designaremos por* $P(a_1,\ a_2,\ ....,\ a_m)$.

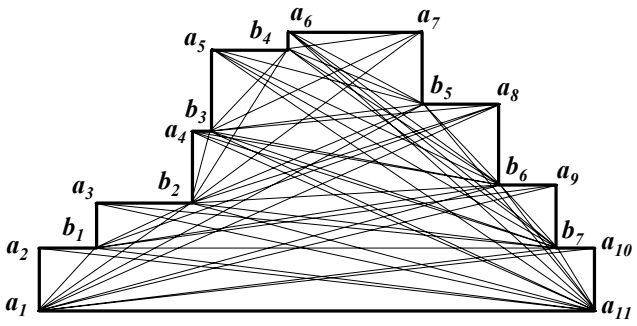


Figura 3.11: Un polígono pirámide y su grafo de visibilidad

De forma análoga a como definíamos el concepto de radio en un polígono *escalera*, definiremos radio de una *pirámide*. Para ello utilizamos el concepto de grafo de visibilidad de un polígono *pirámide*.

Definición 3.3.2 Grafo de Visibilidad de una Pirámide. *Dado un polígono pirámide* $P(a_1,\ a_2,\ ....,\ a_m)$ *se llama grafo de visibilidad de* P *y de denota por* $GV_P(a_1,\ a_2,\ ....,\ a_m)$ *al grafo cuyas aristas unen los vértices visibles de* P. *Se llama radio de* P *y se denota* r *al máximo de la longitud de la aristas del grafo de visibilidad* $GV_P(a_1,\ a_2,\ ....,\ a_m)$.

3.3.2 Alcance mínimo

Dado un polígono escalera P, se define L_{min} como el valor más pequeño para el que colocando luces de alcance L_{min} en todos los vértices del polígono P, éste quede totalmente iluminado. Veremos más adelante que el intervalo de valores significativos para el alcance L resulta ser $[L_{min}, r]$. También se verifica ahora que $L_{min} = \frac{1}{2}r$, como probamos a continuación.

Proposición 3.3.3 *Dado un polígono pirámide* $P(a_1,\ a_2,\ ....,\ a_m)$ *de radio* r, *se tiene que* $L_{min} = \frac{1}{2}r$.

Demostración. Resulta evidente, observando la pirámide de la Figura 3.12, que si tomamos $L = \frac{1}{2}r - \varepsilon \quad \forall \varepsilon > 0$, siempre podemos encontrar un polígono pirámide P_ε que no es posible iluminar colocando una luz de alcance L en cada uno de sus vértices.

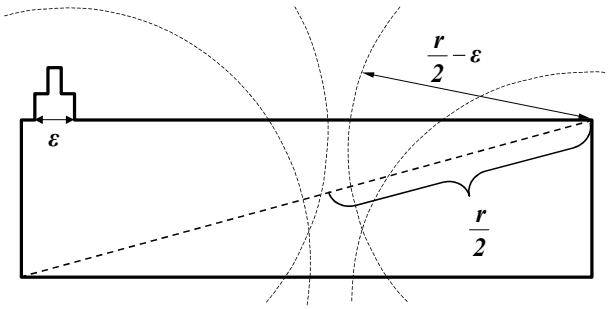


Figura 3.12: P_ε en una pirámide

Para finalizar la demostración de la proposición debemos probar ahora que todo polígono pirámide P de radio r se ilumina con luces vértice colocando en cada vértice una luz de alcance $\frac{1}{2}r$. Si unimos los vértices a_1 o/y a_m de la pirámide con el vértice visible cóncavo más alto del lado opuesto, (ver Figura 3.13), podemos observar que solamente necesitamos iluminar la parte sombreada en oscuro de la pirámide, pues el resto por un mecanismo de cuñas, (que denominaremos cuñas **Tipo 1**), como el utilizado en las escaleras es evidentemente visible.

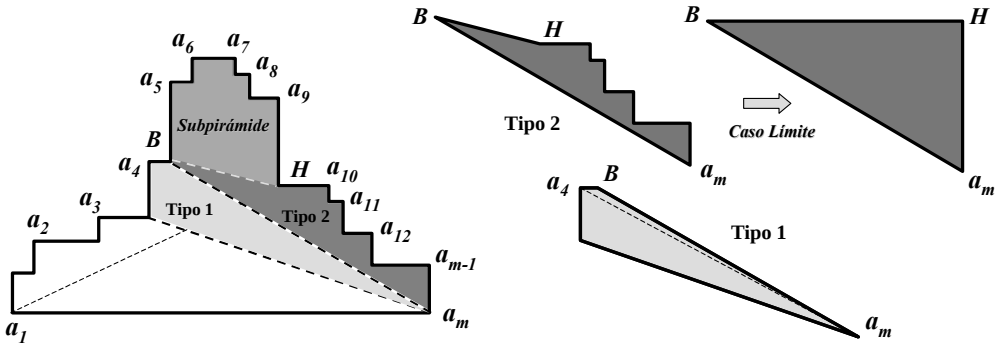


Figura 3.13: Cuñas de un polígono pirámide

Para iluminar la parte superior actuamos así: Uniendo primero el vértice B con H, (que es el vértice cóncavo inmediatamente por debajo de B en el lado opuesto de la pirámide), aparecen cuñas que llamaremos de **Tipo 2** o cuñas ortogonales, que se pueden iluminar colocando luces de alcance $\frac{1}{2}r$ en todos sus vértices, pues caso límite es aquel en el que el ángulo $\widehat{BHa_m}$ es $\frac{\pi}{2}$. Por tanto solamente debemos iluminar la pirámide cuya base es el segmento $\overline{BH}$ y que podremos iluminar aplicando un mecanismo recursivo sobre esta subpirámide. ■

3.3.3 Luces Vértice con Alcance L

Igual que se estudió en los polígonos escalera, analizamos a continuación el número de luces necesarias y suficientes para iluminar el polígono pirámide. En este caso nos restringimos a luces vértice con alcance $L \geq r$.

Proposición 3.3.4 *Para todo polígono pirámide P con n vértices , el número de luces vértice de alcance $L \geq r$, a veces necesarias y siempre suficientes para iluminarlo es $\lceil \frac{n}{6} \rceil$.*

Demostración. Para demostrar la necesidad de $\lceil \frac{n}{6} \rceil$ luces, nos fijamos en la pirámide de la Figura 3.14. Esta pirámide se forma apilando piezas en forma de L, etiquetadas como indica la figura. Desde cada pieza no se ilumina el vértice 3 de la pieza inferior, ni el vértice 6 de la pieza superior. Por tanto necesitamos una luz por cada pieza, es decir, $\lceil \frac{n}{6} \rceil$. Además las terminaciones de la pieza superior que se indican en la figura prueban la necesidad de que sea la parte superior de $\frac{n}{6}$.

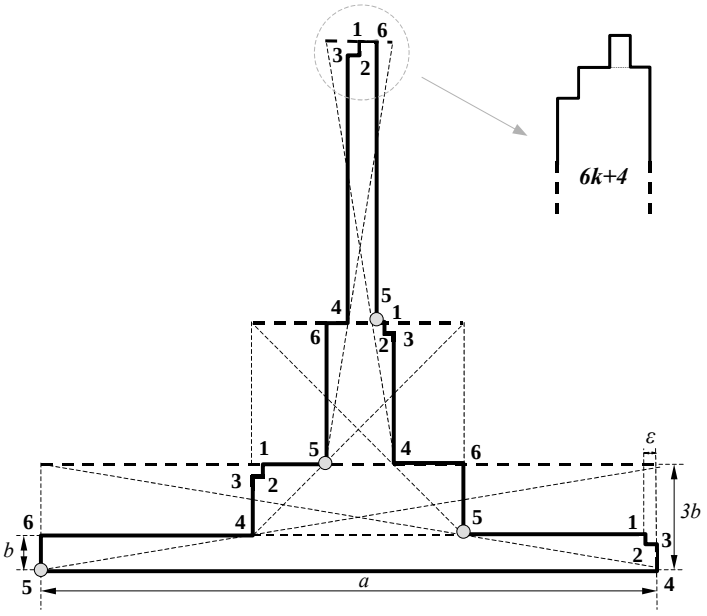


Figura 3.14: Necesidad de $\frac{n}{6}$ con $L \geq r$

Para demostrar la suficiencia debemos probar que todo polígono pirámide se puede iluminar con $\lceil \frac{n}{6} \rceil$ luces de alcance $L \geq r$. Para ello, (ver Figura 3.18), realizamos una descomposición de la pirámide en polígonos estrellados de ocho vértices, cada uno de los cuales tiene seis vértices no comunes con cualquier otra pieza de la descomposición. Etiquetamos los vértices de la pirámide con un barrido descendente, construyendo las piezas que aparecen: la primera arista horizontal se etiqueta con 1 y 2, (1 a la izquierda), y la siguiente con 3 y 4, (3 en el interior). A continuacion iremos numerando descendentemente los vértices del polígono hasta el número 6, teniendo que:

— Si el vértice 5 es cóncavo, se etiqueta con 6 al siguiente cóncavo en la escalera opuesta y se cierra la pieza, (esto sucederá necesarimente en la pieza superior).

— Si el vértice 5 es convexo, se etiqueta con 6 al siguiente cóncavo de su misma escalera y se cierra la pieza con el vértice 6 de la pieza superior.

Con estas dos condiciones conseguimos que el vértice 1 siempre sea convexo y el 6 siempre cóncavo. Las piezas que aparecen en la descomposición pueden ser de tres tipos:

- **Tipo A**: Los vértices 1 y 3 pertenecen a distinta escalera que 2. Estas piezas son estrelladas desde el vértice 3 y por tanto, se podrán iluminar colocando una luz de alcance $L \geq r$ en dicho vértice.

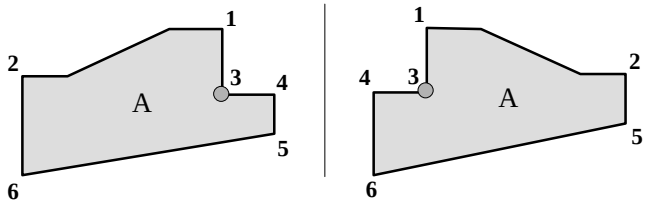


Figura 3.15: Piezas estrelladas Tipo A

- **Tipo B**: Los vértices 2 y 3 pertenecen a distinta escalera que 1. Estas piezas son estrelladas desde el extremo de la arista vertical correspondiente a 1.

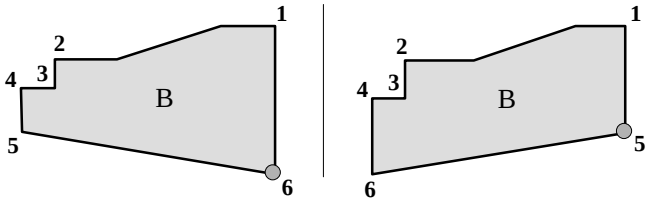


Figura 3.16: Piezas estrelladas Tipo B

- **Tipo C**: Los vértices 1, 2 y 3 están en la misma escalera. La pieza es estrellada desde el vértice 6 de la pieza superior.

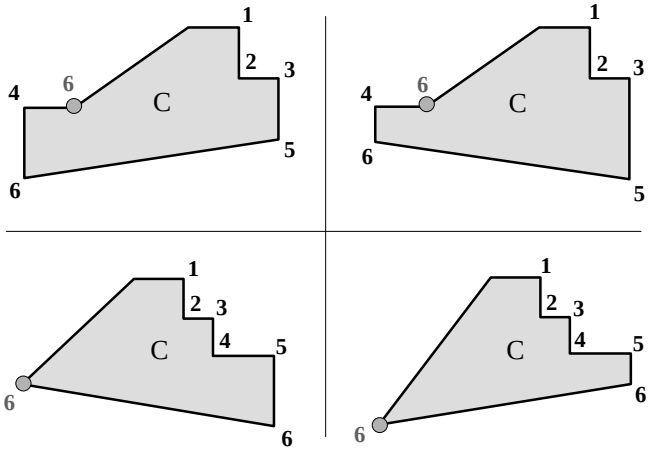


Figura 3.17: Piezas estrelladas Tipo C

Es inmediato comprobar que no hay más tipos de piezas diferentes en la descomposición y que todas las piezas son estrelladas desde un vértice. Además cada nueva pieza que se construye utiliza seis nuevos vértices en el etiquetado. Por tanto, con $\lceil \frac{n}{6} \rceil$ queda iluminada toda pirámide. ■

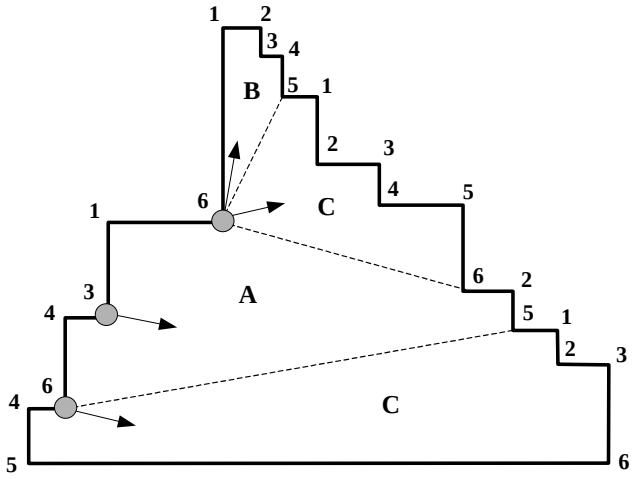


Figura 3.18: Descomposición de una pirámide en polígonos de tipos A, B, y C

Podemos encontrar otras cotas que resultan más evidentes según lo visto hasta ahora y que por ello omitimos aquí su demostración. Como consecuencia de la Proposición 3.3.3 podemos enunciar la siguiente proposición, cuya demostración resulta trivial colocando luces de alcance $\frac{r}{2}$ en los vértices de P.

Proposición 3.3.5 *Todo polígono pirámide P de n vértices se puede iluminar con n luces de alcance $\frac{r}{2}$.*

Por otra parte, como todo *polígono escalera* se puede considerar un caso particular de *polígono pirámide*, tomando el ejemplo descrito en el Lema 3.2.3 es evidente que:

Proposición 3.3.6 *Si ε es suficientemente pequeño y $L = r - \varepsilon$, entonces existe un polígono pirámide P de n vértices que necesita $\lfloor \frac{n}{4} \rfloor + 2$ luces de alcance L para ser iluminado.*

3.4 Conclusiones y trabajos futuros

Hemos estudiado en este capítulo de esta memoria algunos aspectos de la visibilidad de *alcance limitado* en polígonos *escalera* y polígonos *pirámide*.. Se demuestra que el intervalo de valores significativos para el alcance L en dichos polígonos es $[\frac{r}{2}, r]$, donde r es el radio del polígono. Nótese que esta propiedad es general para ambos tipos de polígonos. Además se prueba que $\lfloor \frac{n}{4} \rfloor + c$ es el número de luces a veces necesario y siempre suficiente para iluminar cualquier polígono *escalera*, cuando el alcance $L \in [\frac{r}{2}, r)$ y que esta cota es de $\lceil \frac{n}{6} \rceil$ para polígonos pirámide, con $L \geq r$. Queda pendiente el estudio de otros tipos de polígonos ortogonales y de la iluminación

con luces no situadas en los vértices. A modo de resumen se presentan en la siguiente tabla los resultados combinatorios obtenidos. Recuérdese que el problema combinatorio tratado se llamó CombN-v-Es(P, L), para polígonos *escalera* y CombN-v-Pi(P, L) para polígonos *pirámide*.

Tipo de polígono	L_{min}	*Problema Combinatorio*
Escalera	$\frac{1}{2}\ r$	$\text{CombN-v-Es}(P,L) = \left\{ \begin{array}{ll} 1 & \text{si } L = r \\ \lfloor \frac{n}{4} \rfloor + c & \text{si } \frac{r}{2} \leq L < r \end{array} \right\}$
Pirámide	$\frac{1}{2}\ r$	$\text{CombN-v-Pi}(P,L) = \lceil \frac{n}{6} \rceil \quad \text{si } L \geq r$

Tabla 3.1: Resultados obtenidos en visibilidad limitada

El caso de polígonos pirámide solamente ha sido resuelto cuando $L \geq r$. Por tanto queda abierto el problema CombN-v-Pi(P, L) con $L < r$ para futuras investigaciones.

Capítulo 4

t−Buena iluminación

Uno de los objetivos de esta memoría consistía en buscar definiciones de *visibilidad* o *iluminación* más reales que permitiesen su utilización en campos como la ingeniería ó la física. Así, estudiamos en este capítulo la $t-$*buena iluminación*. La idea de este tipo de iluminación se basa en considerar que un punto no está bien iluminado si todas las luces que lo iluminan están agrupadas únicamente a un "lado" del punto, es decir, si no están en cierto sentido, bien distribuidas alrededor del punto.

Así, consideraremos la siguiente variación de *visibilidad*: un punto p está $t-$*bien iluminado* por un conjunto de luces o focos F si cada semiplano con borde en x contiene al menos t elementos de F visibles desde p. En este capítulo presentamos resultados para el cálculo de la *región* $t-$*bien iliminada*, con $t = 1$ ó $t = 2$, en diferentes situaciones geométricas. Se analizan también casos con t general para situaciones que se pueden solucionar utilizando resultados conocidos.

4.1 Introducción

En muchos problemas geométricos en la ingeniería y la física aparece el concepto de *iluminación* o *vigilancia*. Sin embargo en ocasiones, los resultados obtenidos en Geometría Computacional no pueden ser utilizados de forma práctica en estos campos al no reflejar totalmente circunstancias reales.

En el intento de buscar una definición de *iluminación* o *vigilancia* que se aproxime más a la realidad, definimos en este capítulo la $t-$*buena iluminación*. La idea fundamental de esta nueva iluminación consiste en considerar que un punto p no está bien iluminado si todos los focos que lo iluminan están en cierto sentido, "mal colocados" respecto de p.

El problema original de la Galería de Arte propuesto por V. Klee pregunta cuántas luces son suficientes para iluminar todo punto interior a un polígono P con n vértices. Chvátal demostró que $\lfloor \frac{n}{3} \rfloor$ guardias o luces son siempre suficientes y a veces necesarias [24]. Muchas variaciones de este teorema han sido ya estudiados como podemos comprobar en [80] y [99].

En 1994, Belleville y otros, [8], definen el concepto de $k-guarding$ usando guardias situados en las aristas del polígono. Un polígono P está $k-guardado$ por un conjunto de guardias si cada

punto de P es vigilado por k guardias y hay a lo más un guardia en cada arista de P. Belleville y otros obtienen resultados sobre $1-guarding$ y $2-guarding$.

Introducimos en el presente capítulo el concepto de $t-buena\ iluminación$ donde no sólo un conjunto de luces deben iluminar un punto p, sino que las luces deben estar bien situadas alrededor de p. Consideraremos inicialmente que tenemos k luces en el plano euclídeo en que podemos encontrar obstáculos de diferente naturaleza: segmentos, rectas, polígonos, etc. Así, podemos definir la $t-buena\ iluminación$ de la siguiente manera:

Definición 4.1.1 *Sea $F = \{f_1, f_2, ..., f_k\}$ un conjunto de k luces o focos en el plano y O un conjunto de obstáculos. Un punto p está $t-$bien iluminado por L, $1 \leq t \leq \frac{k}{2}$, si todo semiplano con borde en p contiene al menos t focos de L que lo iluminan.*

Como se puede observar en la Figura 4.1 el punto p está $1-bien\ iluminado$ por los focos f_1, f_2 y f_3, ya que ninguno de los obstáculos evita que cualquier semiplano que pasa por p deje focos a ambos lados del mismo. Sin embargo, el punto q no está $1-bien\ iluminado$ ya que dibujando un semiplano con borde en q que deje a los focos f_2 y f_3 a un lado y f_1 al otro, éste no iluminará a q. Según la definición anterior puede darse el caso que un foco f_i pertenezca a la recta que genera los semiplanos. En este caso consideraremos que f_i ilumina ambos semiplanos.

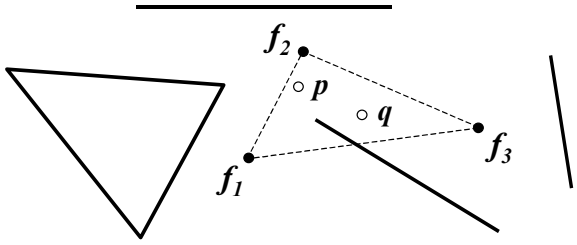


Figura 4.1: Un ejemplo de $1-$buena iluminación

El conjunto de puntos del plano $t-bien\ iluminados$ por F en presencia de un conjunto de obstáculos O lo llamaremos *region $t-$bien iluminada por F con respecto a O* y lo denotaremos con $W_t(F, O)$. Evidentemente la *región $t-$bien iluminada* puede ser no conexa. Por tanto hablaremos indistintamente de las *regiones* o la *región $t-$bien iluminada* por un conjunto F de focos.

Ahora bien, ¿qué forma tienen las *regiones $t-$bien iluminadas*? En la siguiente proposición demostramos que estas regiones son siempre convexas, independientemente de la posición de los focos, si el conjunto de obstáculos O es vacío.

Proposición 4.1.2 *Sea $F = \{f_1, f_2, ..., f_k\}$ un conjunto de k de focos. Si el conjunto O de obstáculos es vacío o ningún obstáculo interseca a $CH(F)$, entonces las regiones $W_t(F, O)$ son siempre convexas.*

Demostración. Como se muestra en la Figura 4.2, sean a, b dos puntos $t-bien\ iluminados$ por F y c un punto del segmento $\overline{ab}$. Para cualquier semiplano π que pase por c existe otro π' que pasa por b y cuyo borde es paralelo al de π. Como π' contiene t focos y $\pi' \subset \pi$, entonces c es un

punto t-*bien iluminado.* ■

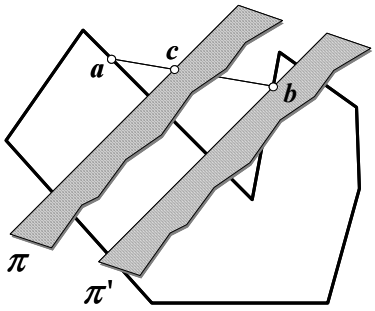


Figura 4.2: La región $t-$bien iluminada es convexa

Estudiamos en este capítulo el cálculo de estas regiones en diferentes situaciones geométricas. En la Sección 4.2 consideramos el caso en que ningún obstáculo interseca al cierre convexo del conjunto de luces, $CH(F)$, viendo que en este caso la *región* $t-$*bien iluminada* coincide con los *niveles de profundidad* de un conjunto de luces [25]. Este problema se ha denotado con B$t-$IGenk(F) y formalmente se puede enunciar de la siguiente manera:

buena $t-$iluminación en posición general

B$t-$IGenk(F):
ENTRADA: Un conjunto F de k focos en el plano euclídeo en posición general.
PREGUNTA: ¿Cómo podemos calcular la *región bien* $t-$*iluminada* por los focos de F?

En las Secciones 4.3 y 4.4 presentamos dos casos con obstáculos. El primero es el caso en el que el obstáculo es el borde de un polígono simple y las luces están situadas en los vértices. En este caso estudiamos el cálculo de $W_t(F,P)$ si el polígono es convexo y presentamos un algoritmo para hallar la *región* $2-$*bien iluminada*, $W_2(F,P)$, si el polígono no es convexo.

buena $t-$iluminación en un polígono convexo

B$t-$ICon(P):
ENTRADA: Un polígono convexo P con n vértices.
PREGUNTA: ¿Cómo podemos calcular la *región* $t-$*bien iluminada* por n focos situados en los vértices de P?

buena $t-$iluminación en un polígono no convexo

B2$-$IPol(P):
ENTRADA: Un polígono P no convexo de n vértices.
PREGUNTA: ¿Cómo podemos calcular la *región* $2-$*bien iluminada* por n focos situados en los vértices de P?

En el segundo de los casos con obstáculos que estudiamos, existe un obstáculo poligonal convexo C y un conjunto F de k focos o luces exteriores a él, dando un algoritmo de complejidad $O(k(\log n+\log k)+n)$ para el cálculo de $W_1(F,C)$, donde n es el número de vértices del convexo. Este algoritmo precisa de un preproceso que ordene angularmente los focos alrededor del convexo

con un coste computacional de $O(k \log k)$.

buena $1-$iluminación para un obstáculo convexo
B1−ICk(C, F):
ENTRADA: Un polígono convexo C con n vértices y conjunto F de k focos exteriores a C.
PREGUNTA: ¿Cómo podemos calcular la *región* $1-$*bien iluminada* por los focos de F?

Como se puede imaginar un estudio completo de este nuevo tipo de iluminación, la $t-$*buena iluminación*, sería muy extenso. Hemos pretendido en este capítulo definir un nuevo concepto de iluminación y estudiar problemas concretos que pueden servir como base para la investigación en otras situaciones geométricas.

4.2 t−Buena iluminación sin obstáculos

Si no hay obstáculos, o los obstáculos no intersecan al cierre convexo de F, $CH(F)$, la *región* $t-$*bien iluminada* coincide con el *nivel de profundidad* t del conjunto F, (también llamados *niveles de profundidad* de Tukey, (*depth levels*); ver [98] para los artículos originales de Tukey). En la Figura 4.3 mostramos un ejemplo de esta región.

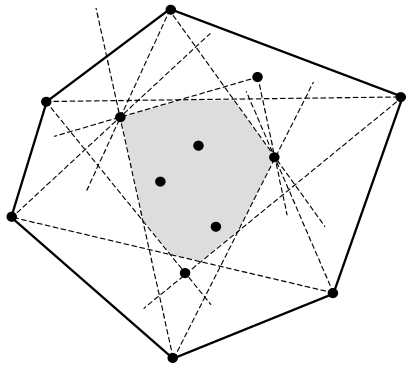


Figura 4.3: Zona $3-$bien iluminada por focos en posición general

Claverol en [25] demostró que se pueden calcular los *niveles de profundidad* en tiempo óptimo $O(n^2)$. Podemos aplicar este resultado para calcular las *regiones* $t-$*bien iluminadas* por F, cuando el conjunto de obstáculos O no intersecan a $CH(F)$, en tiempo óptimo $O(k^2)$.

4.3 t−Buena iluminación en un polígono

Consideremos un polígono P con un foco en cada uno de sus vértices $\{v_1, v_2, ..., v_n\}$ y supongamos que queremos calcular la región de P $t-$*bien iluminada* por esos focos. Distinguiremos dos casos: que P sea un polígono convexo o que no lo sea. Veremos en el caso convexo que la región $t-$*bien iluminada* coincide con el $(t-1)$-núcleo de P y por tanto se podrá calcular en tiempo $\Theta(n)$.

4.3.1 Polígono convexo

Nos planteamos ahora la búsqueda de un algoritmo para el problema B$t-$ICon(P), es decir, buscamos la región $t-$*bien iluminada* por un conjunto F de focos situados en los vértices de un polígono convexo P. En [1] se define el r-núcleo de un polígono convexo P de n vértices en los siguientes términos:

Definición 4.3.1 *Si P es un polígono convexo de vértices $V = \{v_1, ..., v_n\}$ y $r \in \mathbb{Z}$, $0 \leq r \leq n$, llamamos r-núcleo de P y lo designamos por $Ker_r(P)$ al conjunto*

$$Ker_r(P) = \bigcap_{\{i_1,...,i_r\}} CH(V \backslash \{v_i, ..., v_r\}) \tag{4.3.1}$$

donde la intersección recorre todos los subconjuntos de r elementos de $\{1, ..., n\}$ y $CH(A)$ indica el cierre convexo de A.

En la siguiente proposición probamos que la región $t-$*bien iluminada* por los vértices de P es $Ker_{t-1}(P)$, es el $(t-1)$-núcleo de P.

Proposición 4.3.2 *Sea P un polígono convexo con n vértices $\{v_1, v_2, ..., v_n\}$. La región $t-$bien iluminada por n focos situados en los vértices de P es $Ker_{t-1}(P)$.*

Demostración. Según la Definición 4.3.1, $Ker_{t-1}(P)$ es la intersección de todos los cierres convexos que dejan $t-1$ vértices de P fuera de ellos. Por tanto para todo punto $p \in Ker_{t-1}(P)$, cualquier semiplano con borde en p dejará a cada lado al menos t vértices de P y por tanto t focos, (nótese que los focos sobre la recta que determina el semiplano se considera que iluminan ambos semiplanos). Además para cualquier otro punto $q \notin Ker_{t-1}(P)$ siempre podemos encontrar un semiplano con borde en q que deja $t-1$ focos a uno de sus lados, tomándolo adecuadamente paralelo a uno de los semiplanos que determinan $Ker_{t-1}(P)$. ■

En la Figura 4.4 se ilustra esta situación y se construye la *región* $3-$*bien iluminada* por los vértices de P, es decir, $Ker_2(P)$.

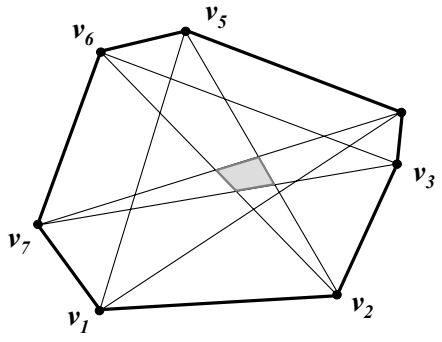


Figura 4.4: Región $3-$bien iluminada por los vértices de P, $(Ker_2(P))$

Así, podemos caracterizar la *región* $t-$*bien iluminada* de la siguiente manera:

Lema 4.3.3 *Un punto x de un polígono convexo P está $t-$bien iluminado por los vértices de P si y solo si x está contenido en el semiplano de borde $\overline{v_i v_{i+t}}$ que no contiene a v_{i+1} para todo $i = 1, 2, ..., n$ (considerando los índices módulo n).*

Por tanto como sabemos que el t-núcleo de un conjunto de puntos en posición convexa se puede calcular en tiempo óptimo $\Theta(n)$, podemos enunciar el siguiente teorema:

Teorema 4.3.4 *La construcción de la región $t-$bien iluminada por los n vértices de un polígono convexo se efectúa en tiempo $\Theta(n)$.*

Una vez que hemos estudiado el cálculo de la *región $t-$bien iluminada* por focos situados en los vértices de un polígono convexo P, pasamos a diseñar un algoritmo que calculará la $W_2(F, P)$ con focos situados en los vértices de un polígono P no convexo, (problema B2−IPol(P)). Obsérvese que estudiamos directamente el caso $t = 2$, pues si $t = 1$ y los focos se encuentran situados en los vértices de un polígono P, la región $1-$*bien iluminada* coincide con P, ya que si triangulamos el polígono todo punto interior cae dentro de algún triángulo de dicha triangulación y por tanto está $1-$*bien iluminado.*

4.3.2 Polígono no convexo

Si el polígono P no es convexo y $t = 2$ debemos sustituir la diagonal $\overline{v_i v_{i+2}}$ por el camino geodésico que une v_i con v_{i+2} en el interior de P. Los puntos de la región poligonal H_i determinada por el camino geodésico de v_i a v_{i+2} y el vértice v_{i+1}no están $2-$*bien iluminados*, (ver Figura 4.5). Si $z \in H_i$, un semiplano cuyo borde pasa por z y no corta al camino geodésico de v_i a v_{i+2} contiene menos de 2 focos, por lo que no está $2-$*bien iluminado.*

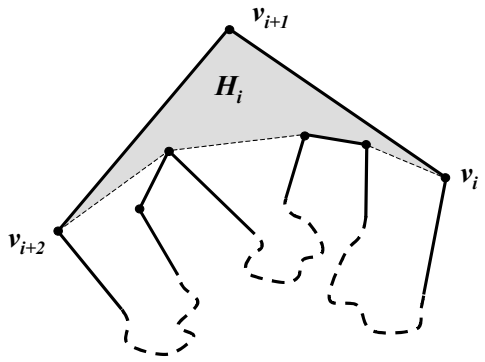


Figura 4.5: Región H_i sin $2-$buena iluminación

Así, la región $2-$*bien iluminada* está contenida en la intersección de las regiones $P \setminus H_i$ para $i = 1, 2, ..., n$. Pero no coincide con ella según muestra el siguiente ejemplo: en el polígono de la Figura 4.6, tomando z un punto del triángulo $\Delta(axd)$, el semiplano determinado por la recta paralela a $\overline{da}$ que pasa por z solamente contiene al foco x, por lo que z no es un punto $2-$*bien iluminado.* Un triángulo no bien iluminado aparece en cada lado del polígono incidente con el vértice cóncavo.

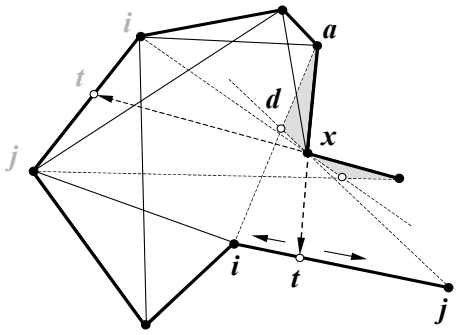


Figura 4.6: $2-$buena iluminación en un polígono P

Con todo lo anterior podemos describir el siguiente algoritmo para la construcción de $W_2(F, P)$. El proceso se divide en tres pasos: los dos primeros construyen las regiones no $2-$bien iluminadas y en el tercero se eliminan de P.

Algoritmo de $2-$buena iluminación en un polígono no convexo

ENTRADA: Un polígono P con n vértices y un conjunto $F = \{f_1, ..., f_n\}$ de n focos situados en los vértices de P.
SALIDA: La región $2-$*bien iluminada* en el interior de P por F, $W_2(F, P)$.

1. **Construcción de las regiones H_i.** Para cada vértice i trazamos las diagonales correspondientes al camino mínimo desde i al vértice $i + 2$. La región comprendida entre el camino y el vértice $i + 1$ no está $2-$*bien iluminada*, (ver Figura 4.6).

2. **Construcción de triángulos asociados a lados cóncavos.** Por cada arista $\overline{ax}$ incidente en el vértice cóncavo x se elimina una zona no $2-$*bien iluminada* construida del siguiente modo, (ver Figura 4.6): Prolongando el lado $\overline{ax}$ hacia el interior del polígono cortará a un lado de P en un punto t. Girando los segmentos $\overline{at}$ con centro en a y $\overline{xt}$ con centro en x y en sentidos contrarios, encontramos los primeros vértices visibles en ambos casos, que llamaremos i y j respectivamente. Si calculamos ahora el punto de intersección d de las rectas $\overline{xj}$ y $\overline{ai}$, podemos construir el triángulo $\Delta(adx)$ que según se justifico anteriormente es una región no *bien* $2-$*iluminada*. En la figura se muestra también la otra región no *bien* $2-$*iluminada* que se obtiene con el otro lado incidente en el vértice cóncavo x.

3. **Eliminación de las regiones no $2-$bien iluminadas.** Eliminar de P las regiones H_i construidas en el Paso 1 y los dos triángulos adyacentes a cada vértice cóncavo de P, construidos en el Paso 2.

La validez de este algoritmo y su complejidad se analiza en el siguiente teorema:

Teorema 4.3.5 *Si P es un polígono con n vértices y $F = \{f_1, ..., f_n\}$ es un conjunto de n focos situados en sus vértices, el algoritmo anterior construye $W_2(F, P)$ en tiempo $O(n^2)$.*

Demostración. Consideremos un punto q interior a P y que no está ni en el interior de ningún H_i $i = 1, ..., n$, ni en el interior de ninguno de los triángulos adyacentes a vértices cóncavos generados en el Paso 2 del algoritmo.

Si trazamos un semiplano con borde en q éste cortará a dos lados del polígono P, ($\overline{v_i v_{i+1}}$, $\overline{v_j v_{j+1}}$), en puntos i_1 y i_2 respectivamente, (ver Figura 4.7 (a)). Si nos fijamos en uno de los semiplanos, por ejemplo en el que se encuentran los vértices v_i y v_{j+1}, se pueden dar los siguientes casos:

1. Tanto v_i como v_{j+1} iluminan a q: en este caso q estará $2-$*bien iluminado* respecto al semiplano en el que se encuentran v_i y v_{j+1}, (ver Figura 4.7 (a)).

2. Tanto v_i como v_{j+1} iluminan a q, pero $v_i = v_{j+1}$: en este caso q no estaría $2-$*bien iluminado*. Esta situación no se podrá dar pues esto significaría que $q \in H_i$, (ver Figura 4.7 (b)), y este sector ha sido eliminado en el Paso 1 del algoritmo.

3. Uno de los dos focos no iluminan q: si por ejemplo v_i no ilumina a q, entonces existe un foco o vértice del polígono v_t desde el que se ve el punto q, con lo cual este punto estaría $2-$*bien iluminado* respecto al semiplano correspondiente, (ver Figura 4.7 (c)).

4. Uno de los dos focos no ilumina a q, pero es uno de los dos focos v_{j+1} o v_i el que impide la visión desde el otro, (ver Figura 4.7 (d)): en este caso el punto q no estará $2-$*bien iluminado*. Esta situación no se podrá dar pues en este caso q estaría en uno de los triángulos adyacentes al lado $\overline{v_j v_{j+1}}$, eliminado en el Paso 2 del algoritmo.

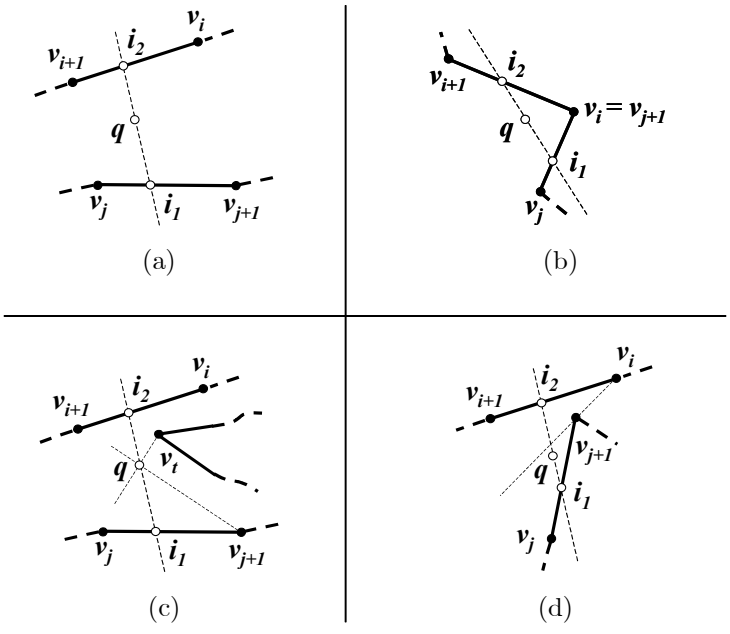


Figura 4.7: Situaciones con $2-$buena iluminación

Respecto a la complejidad, en el primer paso el coste de un camino geodésico es lineal. Por tanto el trazado de todas las diagonales se realiza en $O(n^2)$. Para la construcción de los triángulos eliminables se necesitan ordenar todos los vértices respecto de todos para encontrar i y j, (coste $O(n^2)$), más la obtención del punto t, (coste $O(n)$) y el cálculo de los triángulos que se realiza en tiempo constante para cada lado, es decir, $O(n^2)$ en total. Finalmente debemos recortar todos los triángulos con el polígono resultante de eliminar todas las regiones H_i de P. Esto puede hacerse en $O(n)$ para cada triángulo, lo que produce un coste final para todos los triángulos de $O(n^2)$. ■

Estudiamos a continuación el cálculo de la *región* $1-$*bien iluminada* por un conjunto $F = \{f_1, ..., f_k\}$ de k focos en posición general en el plano, existiendo un obstáculo convexo C con n vértices que impide la visibilidad. Este problema lo hemos denotado anteriormente con $\mathsf{B1{-}ICk}(C, F)$.

4.4 1−Buena iluminación con obstáculo convexo

Todo polígono convexo tiene buenas propiedades respecto a la iluminación o vigilancia. En este sentido podemos aprovechar dichas propiedades en el diseño de un algoritmo que calcule la zona $1-$*bien iluminada* por k focos exteriores a él. Según la Definición 4.1.1 para la $t-$*buena iluminación* caracterizamos la $1-$*buena iluminación* en el siguiente lema cuya demostración resulta trivial y por ello omitimos.

Proposición 4.4.1 *Un punto x está $1-$bien iluminado por un conjunto de focos F si existen tres focos f_1, f_2, f_3 de F que iluminan a x y tal que el triángulo $\Delta(f_1, f_2, f_3)$ contiene a x.*

Supongamos por tanto que tenemos un convexo C con n vértices y un conjunto de k focos exteriores a él, $F = \{f_1, f_2, ..., f_k\}$. Presentamos a continuación un algoritmo para el problema $\mathsf{B1{-}ICk}(C, F)$. Dicho algoritmo consta de un primer paso de preproceso en el cual ordenaremos angularmente los focos alrededor del convexo. El coste computacional de este preproceso se puede dividir en dos apartados:

- Determinación de las cuñas que generan la prolongación de los lados del convexos y determinación de los focos que pertenecen a cada cuña. Este paso tiene un coste conocido de $O(k \log n)$, siendo k el número de focos y n la cantidad de vértices del convexo C.

- Ordenación de los focos en cada cuña. La ordenación debe ser angular y se produce respecto al vértice v_i de C que determina la cuña. El coste de esta operación será $O(k_i \log k_i)$. Si suponemos que tenemos un número t de cuñas, el coste total será:

$$O\left(\sum_{i=1}^{t} k_i \log k_i\right) = O(k \log k) \tag{4.4.2}$$

Como hemos mencionado anteriormente el algoritmo que presentamos consta de una primera parte de preproceso y otra segunda donde se engloban los cálculos propios del algoritmo, como se detalla a continuación:

Algoritmo de $1-$buena iluminación para un convexo

ENTRADA: Un polígono convexo C con n vértices y un conjunto $F = \{f_1, ..., f_n\}$ de n focos exteriores a él..
SALIDA: La región $2-$*bien iluminada* por F, $W_1(F, C)$.

1. **Preproceso**. Determinar las cuñas que producen la prolongación de los lados del convexo y estudiar los focos que pertenecen a cada cuñas. Este preproceso se debe realizar tanto en las cuñas hacia la derecha como las cuñas hacia la izquierda, (es decir, cuando recorremos C en sentido derecho e izquierdo). En la Figura 4.8 presentamos un ejemplo de ordenación de focos alrededor del convexo en sentido derecho.

   Obsérvese que una vez realizada las dos ordenaciones, (en sentido derecho e izquierdo), quedan determinados los focos que vamos encontrando y abandonando al realizar un barrido por una recta que contiene a cualquier lado de C y gira en sentido negativo alrededor de él. Los focos que aparecerán vienen dados por la ordenación de las cuñas derechas y los que desaparecerán por la ordenación de las cuñas izquierdas.

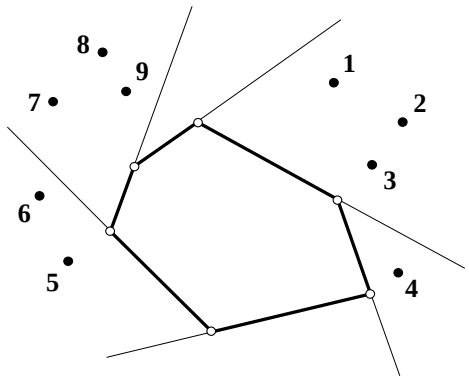


Figura 4.8: Preproceso para un convexo

2. **Cálculo**. El algoritmo consta de dos partes. La primera de ellas consiste en el cálculo de una zona poligonal $\mathcal{A}$, que es la unión de los cierres convexos dinámicos de los subconjuntos de F linealmente separados de C. La segunda parte consiste en completar $\mathcal{A}$ con sectores internos de *buena iluminación* $\mathcal{S}^i$, que no aparecen en la unión de los cierres convexos. Así, la zona bien iluminada por los focos de $F = \{f_1, f_2, ..., f_k\}$ será $\mathcal{A} \bigcup_{i=1}^{k} \mathcal{S}^i$. .Detalladamente estos son los pasos de esta parte de cálculo.

   (a) **Construcción del primer convexo:**
   - Trazar una recta t que contiene a un lado cualquiera de C.
   - Construir el cierre convexo de todos los puntos exteriores a t.

(b) **Cálculo de $\mathcal{A}$. Unión de los cierres convexos dinámicos**

— Girar en sentido negativo la recta t y construir de forma dinámica la unión de los cierres convexos que irán apareciendo en sentido negativo y desapareciendo en sentido izquierdo, (ver Figura 4.9).

(c) **Completar con sectores internos $\mathcal{S}^i$.**

— Para cada foco f_i construir su recta soporte r_i a C.

— Girar r_i en sentido negativo alrededor de C hasta encontrar el primer foco que aparece, f_j.

— Calcular lar recta soporte r_j de f_j a C.

— Hallar el sector $\mathcal{S}^i$ y unirlo a la zona bien iluminada $\mathcal{A}$.

En la Figura 4.9 presentamos la construcción de la zona 1−*bien iluminada* $\mathcal{A}$ resultado de la unión de los cierres convexos dinámicos y el sector $\mathcal{S}^i$ a unir para un foco f_i.

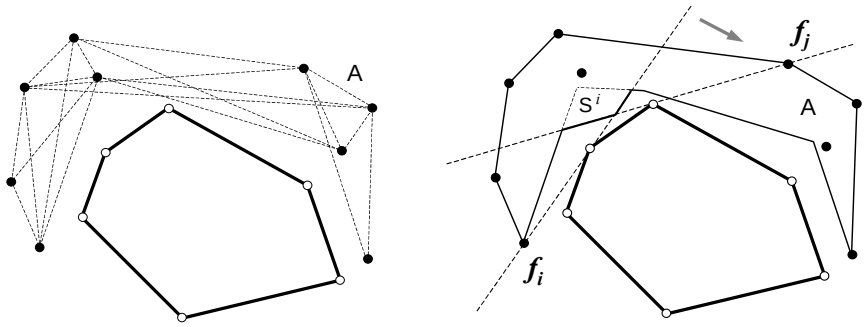


Figura 4.9: Buena 1−iluminación con un obstáculo convexo

Nos preguntamos ahora sobre la complejidad de este algoritmo. En el siguiente teorema establecemos la complejidad del algoritmo que resuelve el problema B1−ICk(C, F).

Teorema 4.4.2 *Dado un conjunto $F = \{f_1, f_2, ..., f_k\}$ de luces-punto en el plano y un obstáculo convexo C de n vértices, la zona 1−bien iluminada se puede calcular en $O(k(\log n + \log k) + n)$.*

Demostración. Analizamos anteriormente que el coste computacional del preproceso se dividía en dos partes. Una de ellas es la determinación de los focos que caen en cada cuña, ($O(k \log n)$) y la otra la ordenación de cada una de éstas, ($O(k \log k)$).

Respecto a la segunda parte de cálculo debemos construir los cierres convexos de forma dinámica, proceso que se realiza en tiempo $O(\log k)$ [18]. Por otra parte, debemos construir un sector para cada foco f_i. Ello precisa el cálculo de 2 rectas soporte a C, que se pueden encontrar recorriendo en sentido negativo el convexo C y que por tanto tendrá una complejidad $O(k + n)$. Así tenemos

$$O(k \log n + k \log k + \log k + k + n) \approx O(k(\log n + \log k) + n) \tag{4.4.3}$$

que prueba nuestro teorema. ■

La complejidad anterior es óptima si $k \in O(n)$ o si $k \in O(1)$. En el primer caso la complejidad es $O(n \log n)$ y si $CH(F)$ y C son disjuntos la *región* $1-$*bien iluminada* es $CH(F)$.

4.5 Conclusiones y trabajos futuros

En este capítulo se ha definido el concepto de $t-$*buena iluminación* imponiendo restricciones sobre la posición de los focos que deben iluminar un determinado punto. Se han estudiado algoritmos para el cálculo de las *regiones* $t-$*bien iluminadas* en diferentes situaciones geométricas: cuando los focos se encuentran en posición general en el plano y sin obstáculos, cuando los focos están situados en los vértices de un polígono P, (tanto convexo como no convexo), y cuando los focos se encuentran situados en posición general en el plano, pero existe un obstáculo convexo C que impide dicha iluminación. A modo de resumen los resultados obtenidos se presentan en la siguiente tabla:

Problema Estudiado	*Posición luces*	*Complejidad Computacional*
B$t-$IGenk(F)	general	$O(k^2)$
B$t-$ICon(P)	convexa	$\Theta(n)$
B2$-$IPol(P)	vértices polígono P	$O(n^2)$
B1$-$ICk(C, F)	exteriores convexo C	$O(k(\log n + \log k) + n)$

Tabla 4.1: Resultados obtenidos en $t-$buena iluminación

Queda abierto un abanico amplio de futuros trabajos en el diseño de técnicas para el cálculo de la $t-$*buena iluminación* con otras condiciones geométricas. Así, se podría estudiar el cálculo de la *región* $t-$*bien iluminada* por focos situados en el interior de un polígono P. Por razonamientos similares a los que se realizan en el Capítulo 5, estas regiones están relacionadas con el *cierre convexo relativo* de dichos focos respecto a P. Una vez cálculado dicho cierre se pueden diseñar algoritmos para el cálculo de la región $t-$*bien* iluminada, utilizando variaciones de los algoritmos expuestos en este capítulo, teniendo en cuenta que ahora no en todos los vértices del *cierre convexo relativo* se tiene situado un foco y además existen focos en su interior.

Capítulo 5

Iluminación múltiple en un polígono

El *polígono de visibilidad* de un punto en el interior de un polígono de n lados puede calcularse en tiempo lineal, (ver [72]). Si estudiamos una solución heurística o aproximada para el problema MinN-p-Pvk(P), (ver Capítulo 8), aparece de forma casi inmediata el problema del cálculo de la unión de *polígonos de visibilidad*. En este capítulo no estudiamos esta unión, sino que nos preguntamos sobre la intersección de *polígonos de visibilidad*, es decir, sobre el cálculo del lugar geométrico de los puntos del polígono iluminados por k focos. Este problema se podría titular *"iluminación múltiple en un polígono"* y como núcleo de un conjunto de puntos ha sido estudiado por Ghosh, ([50] describe un algoritmo para la búsqueda de un punto interior al polígono P iluminado por k focos) y por Ke y O'Rourke en su versión generalizada [67]. Planteamos en este capítulo otra solución al problema que aunque no reduce la complejidad sustancialmente, si resulta interesante respecto a la sencillez de los algoritmos descritos. Se presentan dos algoritmos. Uno incremental de complejidad $O(kn)$ en el que cada inserción se realiza en tiempo lineal. Otro hace uso del *cierre convexo relativo*. Una vez calculado, lo que puede hacerse en $O(n+k\log(kn))$, el algoritmo resuelve el problema en tiempo $O(n+k)$.

5.1 Introducción

Un problema clásico en el campo de la Geometría Computacional es el problema de la *galería de arte*, (ver [99]), sobre iluminación de un polígono. Se conocen cotas combinatorias ajustadas sobre el número de luces necesarias para iluminar un polígono P de n vértices. Sin embargo, minimizar el número de luces necesarias para iluminar un polígono dado es un problema $\mathcal{NP}$-completo. En este sentido, resulta interesante el estudio de las propiedades que tienen los *polígonos de visibilidad* de puntos en el interior de un polígono P. Estas propiedades podrán ayudar en la búsqueda de soluciones aproximadas en nuestros problemas de visibilidad, como estudiamos en la segunda parte de esta memoria.

El problema que planteamos en este capítulo es el siguiente: *"Dado un polígono P de n* vértices *y un conjunto T* de *k focos dentro de él, calcular la región iluminada por todos ellos a*

la vez". Formalmente lo podemos enunciar de la siguiente manera:

iluminacion múltiple en un polígono
p-Pvk(P, T):
ENTRADA: Un polígono P de n vértices y un conjunto T de k *luces-punto* interiores a él.
PREGUNTA: ¿Cómo podemos calcular la zona iluminada por los k focos a la vez?

Comenzamos definiendo en la Sección 5.2 los conceptos básicos sobre visibilidad dentro de un polígono P. En la Sección 5.3 se presenta un algoritmo incremental de complejidad $O(kn)$ para el cálculo de la intersección de *polígonos de visibilidad*. En la Sección 5.4 definimos el concepto de foco *esencial* y se presenta un algoritmo de complejidad $O(n + k\log(kn))$, utilizando para ello una técnica de doble barrido.

5.2 Definiciones y Propiedades

Damos a continuación las definiciones básicas sobre visibilidad que nos ocuparán en el transcurso de este capítulo.

Definición 5.2.1 *Dado un conjunto D de $\Re^d$ se dice que dos puntos x,$y \in D$ son visibles en D cuando el segmento $\overline{xy}$ está completamente contenido en D.*

Definición 5.2.2 *Dado un polígono P, se llama polígono de visibilidad de un punto t de P, que denotaremos con $V(P,t)$, al polígono que contiene a todos los puntos visibles desde t en el polígono P.*

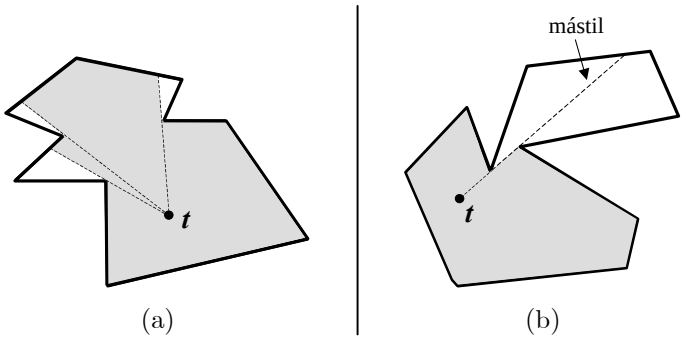


Figura 5.1: (a) Polígono de visibilidad (b) Polígono de visibilidad con mástil

La definición anterior de *polígono de visibilidad* puede producir regiones de visibilidad que no son polígonos simples, como se muestra en la Figura 5.1 (b). Estos polígonos los llamaremos *polígonos de visibilidad con mástiles* y se considerarán situaciones degeneradas. Ni esta situación ni aquella en la que dos focos estén alineados con un vértice de P se considerarán en este trabajo.

En el lema siguiente se exponen algunas propiedades básicas de los *polígonos de visibilidad.*

Lema 5.2.3 *Si P es un polígono de n vértices, el polígono de visibilidad de un punto t, $V(P,t)$, es un polígono estrellado con, a lo sumo, n vértices. Los lados de $V(P,t)$ o bien están contenidos en los lados de P o son cuerdas que unen un vértice de P con un punto de otro lado de P.*

5.3 Algoritmo incremental

Presentamos a continuación un algoritmo incremental de complejidad $O(kn)$. Este algoritmo es óptimo si se procesan los focos al vuelo (según van llegando). Analizamos en primer lugar el caso en que tenemos únicamente dos focos t_1 y t_2, generalizando a continuación a k focos.

5.3.1 Algoritmo de intersección

Dado un polígono P y dos puntos interiores t_1 y t_2 a P, cuyos *polígonos de visibilidad* son $V(P,t_1)$ y $V(P,t_2)$, describimos un algoritmo para el cálculo del polígono intersección, (que denotaremos con $V_I(P,t_1,t_2)$), distinguiendo dos casos: que t_1 y t_2 sean visibles o que no lo sean. Si son visibles, el cálculo de $V_I(P,t_1,t_2)$ se reducirá a calcular el *polígono de visibilidad* de t_1 dentro de $V(P,t_2)$, (es decir, $V_I(P,t_1,t_2) = V(V(P,t_2),t_1)$), o a calcular el *polígono de visibilidad* de t_2 dentro de $V(P,t_1)$. Si los puntos t_1 y t_2 no son visibles podemos distinguir dos subcasos: el primero de ellos es que no exista un punto c que vea a ambos puntos t_1 y t_2, en cuyo caso $V_I(P,t_1,t_2) = \emptyset$; por el contrario, si existe un punto c que vea a t_1 y t_2, $V_I(P,t_1,t_2) = V(V(P,t_1),c) \cap V(V(P,t_2),c)$. Además el polígono solución será estrellado desde este punto c.

A continuación describimos con detalle los pasos del algoritmo que se ilustra en la Figura 5.2, estudiando los casos posibles.

Algoritmo de intersección de dos polígonos de visibilidad

ENTRADA: Dos *polígonos de visibilidad* $V(P,t_1)$ y $V(P,t_2)$.
SALIDA: El polígono *intersección* $V_I(P,t_1,t_2) = V(P,t_1) \cap V(P,t_2)$.

1. **Caso A** — t_1 y t_2 visibles.
   - Calcular $V(P,t_1)$.
   - $V_I(P,t_1,t_2) = V(V(P,t_1),t_2)$.
2. **Caso B** – t_1 y t_2 no son visibles. El segmento $\overline{t_1t_2}$ corta a ∂P. Consideramos las cadenas de ∂P que empiezan y terminan en dicho segmento, como se muestra en la Figura 5.2 y se calcula el cierre convexo H de la unión de dichas cadenas. Los pasos son:
   - Si t_1 ó t_2 están contenidos en H entonces $V_I(P,t_1,t_2) = \emptyset$ y fin.
   - En caso contrario, se trazan las semirrectas soporte de H desde t_1 y t_2 y se calcula su punto de intersección c_1. Si c_1 no existe, $V_I(P,t_1,t_2) = \emptyset$ y fin.
   - Si $\overline{t_1c_1}$ o $\overline{t_2c_1} \not\subset P$ entonces $V_I(P,t_1,t_2) = \emptyset$ y fin.
     En otro caso $V_I(P,t_1,t_2) = V(V(P,t_1),c_1) \cap V(V(P,t_2),c_1)$.

Siguiendo la misma idea de este algoritmo para la intersección de dos *polígonos de visibilidad*, podemos diseñar el siguiente algoritmo para calcular la itersección de k *polígonos de visibilidad*.

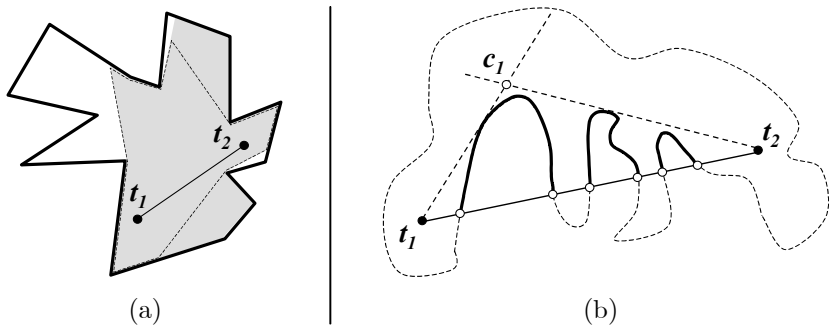


Figura 5.2: (a) Puntos visibles (b) Cadenas generadas por puntos no visibles

Intersección de k polígonos de visibilidad

Realizar la intersección de k *polígonos de visibilidad,* correspondientes a los focos $t_1, ..., t_k$, se puede realizar de forma incremental utilizando como caso básico el algoritmo anterior. Describimos el paso k para la intersección de $V_I(P, t_1, ..., t_{k-1})$ con $V(P, t_k)$.

Algoritmo de intersección de k polígonos de visibilidad

ENTRADA: El polígono $V_I(P, t_1, ..., t_{k-1})$ estrellado desde c_{k-1} y $V(P, t_k)$.
SALIDA: El polígono *intersección* $V_I(P, t_1, ..., t_k) = V(P, t_1) \cap ... \cap V(P, t_k)$.

1. **Caso A** — c_{k-1} y t_k visibles.

   – $V_I(P, t_1, ..., t_k) = V(V(P, t_k), c_{k-1}) \cap V_I(P, t_1, ..., t_{k-1})$.

2. **Caso B –** c_{k-1} y t_k no son visibles. El segmento $\overline{c_{k-1}t_k}$ corta a ∂P. Consideramos las cadenas de ∂P determinadas por dicho segmento, como se ha explicado en el algoritmo anterior, y se calcula el cierre convexo H de la unión sde dichas cadenas.

   2.1. Si c_{k-1} ó t_k están contenidos en H entonces $V_I(P, t_1, ..., t_k) = \emptyset$ y fin.

   2.2. En caso contrario, hallar la recta soporte desde t_k a H y calcular, si existe, el punto de intersección c_k con $V_I(P, t_1, ..., t_{k-1})$ más cercano a t_k. Si c_k no existe $V_I(P, t_1, ..., t_k) = \emptyset$ y fin.

   2.3. Si $\overline{t_k c_k} \not\subset P$ entonces $V_I(P, t_1, ..., t_k) = \emptyset$ y fin. En caso contrario $V_I(P, t_1, ..., t_k) = V(V_I(P, t_1, ..., t_{k-1}), c_k) \cap V(V(P, t_k), c_k)$.

Como consecuencia del algoritmo anterior tenemos el siguiente lema.

Lema 5.3.1 *Si el polígono intersección de k polígonos de visibilidad $V_I(P, t_1, ..., t_k)$ es no vacío entonces es un polígono estrellado.*

Estudio de la Complejidad

La complejidad del algoritmo es $O(kn)$ ya que es un proceso incremental en el que cada inserción se realiza en $O(n)$. Efectivamente, en el Caso A se calcula un *polígono de visibilidad* y una intersección de dos polígonos estrellados desde un mismo punto. Ambas operaciones se calculan en $O(n)$. En el Caso B el cálculo del cierre convexo H se realiza en tiempo lineal ya que las cadenas están ordenadas. El paso 2.1. se realiza en $O(\log n)$ y el 2.2. en $O(n)$. Un argumento similar al del Caso A prueba que el apartado 2.3. se realiza en tiempo lineal.

5.4 Algoritmo de doble barrido

Si los focos se dan de uno en uno y se actualiza la región común de visibilidad tras cada inserción, el algoritmo anterior es óptimo. Si se conocen todos los focos, es posible obtener un algoritmo más rápido que presentamos en esta sección.

5.4.1 Focos esenciales

Definición 5.4.1 *Dado un polígono P de n vértices y un conjunto $T = \{t_1, ..., t_k\}$ de k focos interiores a P, se dice que el foco t_i es no esencial si $V_I(P,T) = V_I(P, T\backslash\{t_i\})$. En caso contrario llamaremos a t_i, foco esencial.*

Pero, ¿cuántos focos esenciales podemos tener dentro de P? En la siguiente proposición relacionamos el número de focos esenciales con el de cadenas cóncavas, (es decir cadenas que sólo contienen vértices cóncavos), que aparecen en P.

Proposición 5.4.2 *Por cada cadena cóncava de un polígono P de n vértices, tendremos a lo sumo 2 focos esenciales.*

Demostración. Si nos fijamos en la Figura 5.3 observamos que cuando la cadena cóncava está generada únicamente por 1 vértice, los focos esenciales se encuentran en los sectores determinados por las rectas que continenen a los segmentos $\overline{v_{i-1}v_i}$ y $\overline{v_iv_{i+1}}$, donde v_i es el vértice cóncavo, (en la Figura 5.3 (a) se muestran marcados con una fecha gris y otra negra). Los focos que determinan la menor zona iluminada sobre la otra sección de la zona cóncava, se encuentran lo más cerca, (angularmente), posible a los segmentos $\overline{v_{i-1}v_i}$ y $\overline{v_iv_{i+1}}$, pues en otro caso iluminarían mayor zona. Además cualquier otro vértice que encontremos a continuación no será esencial para esta cadena cóncava, ya que iluminarán mayor zona en el otro sector de la concavidad. Por tanto, para encontrar dichos vértices es suficiente girar los segmentos anteriores en ambos sentidos alrededor de v_i hasta encontrar dichos vértices, o hasta alinearse con el otro segmento, (lo que indicará que en uno de los sectores no tendremos vértice esencial).

Cuando el número de vértices cóncavos es mayor que 1, la situación es similar, pero la búsqueda de los vértices asociados a los focos esenciales cambia. La diferencia radica en la forma de giro. Los giros ahora no se deben realizar sobre un único vértice cóncavo, sino que se debe cambiar el centro del giro al siguiente vértice cóncavo, cuando el giro se alinea con la siguiente arista de P. Así, si nos fijamos en la Figura 5.3 (b) y comenzamos girando el segmento $\overline{v_{i-2}v_{i-1}}$

alrededor de v_{i-1}, pararemos cuando nos alineamos con el segmento $\overline{v_{i-1}v_i}$, cambiando en este caso el centro del giro al vértice v_i. Este proceso se itera hasta encontrar el foco esencial t_5. A continuación repetimos el proceso comenzando por el otro extremo de la cadena cóncava, es decir, por el segmento $\overline{v_{i+2}v_{i+3}}$, hasta hola que tal estamos encontrar el foco esencial t_3.

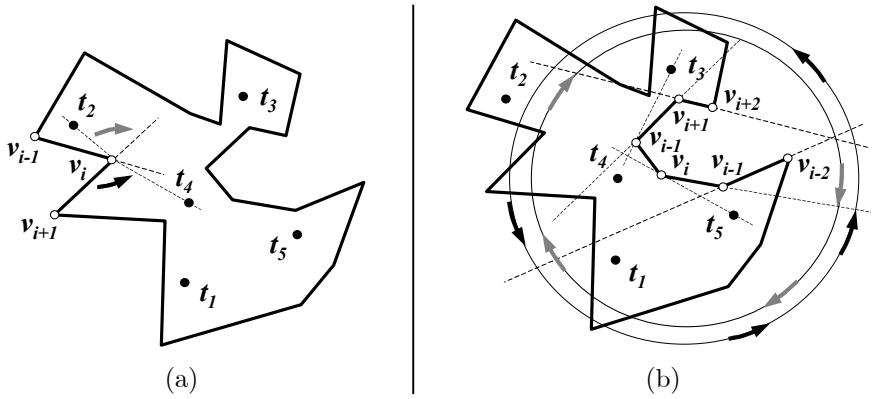


Figura 5.3: Focos esenciales sobre cadenas cóncavas

La construcción realizada para la búsqueda de los focos esenciales determina también en este caso, que el número de focos esenciales debe ser 2, uno por cada zona de la concavidad, ya que nos interesan los focos más cercanos, (angularmente), a los primeros lados de la cadena cóncava, Estos focos son los que iluminan la menor zona posible de la otra cara de la concavidad y por tanto serán esenciales en la intersección. Cualquier otro foco iluminará mayor zona de la otra cara de la concavidad y por tanto no será esencial para la intersección. ■

Como consecuencia de esta proposición anterior podemos enunciar la siguiente que determina el número máximo de focos esenciales que pueden aparecen en el interior de un polígono P de n vértices.

Proposición 5.4.3 *Si P es un polígono con n vértices, el número de focos esenciales que podemos tener en su interior es a lo sumo n. Además esta cota es ajustada.*

Demostración. Según la Proposición 5.4.2, el número de focos *esenciales* por cada cadena cóncava de un polígono P es 2. Como sabemos el número máximo de cadenas cóncavas de un polígono P de n vértices es $\frac{n}{2}$, de donde se deduce que la intersección tiene a lo sumo n vértices. Además la Figura 5.4 muestra que dicha cota es ajustada, pues tenemos un polígono con el mismo número de vértices que focos esenciales. ■

Del mismo modo que se puede acotar el número de focos esenciales, podemos establecer una cota para el número de vértices del polígono intersección. Se demuestra que el número de vértices de dicho polígono es a lo más n, siendo n el número de vértices del polígono inicial P.

Teorema 5.4.4 *Si $T = \{t_1, ..., t_k\}$ es un conjunto de k focos interiores a un polígono P de n vértices, entonces el polígono $V_I(P, T)$ tiene a lo sumo n vértices. Además dicha cota es ajustada.*

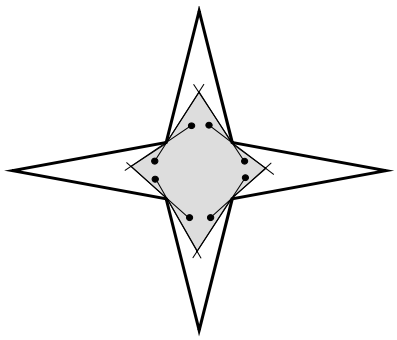


Figura 5.4: Cota ajustada de la Proposición 5.4.3 y del Teorema 5.4.4

Demostración. A cada vértice v de V_I se le asocia de manera unívoca un vértice del polígono del siguiente modo: Si es un vértice del polígono se le asocia el mismo. En caso contrario, existen dos focos esenciales y dos semirrectas que parten de ellos y que se cortan en v, apoyándose en dos vértices u_1 y u_2 de P. Si entre u_1 y u_2 hay algún vértice de P, se le asigna éste a v. En otro caso, ni u_1 ni u_2 son vértices de la intersección. Asociamos v a u_1 (resp. u_2), si $sig(u_1, v, u_2) < 0$ (resp. > 0). Además la cota es ajustada, según se muestra en la Figura 5.4. ■

5.4.2 Cierre convexo relativo

El *cierre convexo relativo* de un conjunto T de k focos dentro del polígono P, $CCR(P,T)$, (ver [97]), es la herramienta adecuada para eliminar focos *no esenciales*. En los siguientes lemas estudiamos las relaciones entre este *cierre convexo relativo* y los focos esenciales interiores a un polígono P.

Lema 5.4.5 *Si P es un polígono de n vértices y T un conjunto de k focos interiores a P, entonces $CCR(P,T)$ contiene en su frontera a todos los focos esenciales.*

Demostración. Si calculamos el *cierre convexo relativo* de un conjunto T de focos interiores a P, obtenemos un polígono como el que se muestra en el interior del polígono de la Figura 5.5. Los vértices de CCR son o bien focos, o bien vértices cóncavos de P. Además, si un foco t_i está unido con un vértice cóncavo v_i, entonces t_i es un foco esencial asociado a la cadena cóncava a la que pertenece v_i, ya que es el foco más cercano angularmente al lado del polígono al que pertenece v_i, (el lado correspondiente al sector en el que está el foco t_i). ■

El cálculo del *cierre convexo relativo* tiene un coste computacional que es $O(n + k\log(kn))$ y constituye la parte más costosa de nuestro algoritmo final de doble barrido, para el cálculo de V_I. Observando también la Figura 5.5, deducimos de forma inmediata el siguiente lema.

Lema 5.4.6 *Los vértices cóncavos de $CCR(P,T)$ son vértices de P y la región común de visibilidad de T no varía si se añaden a T dichos vértices*

Según se mencionó en el resumen inicial, el objetivo fundamental de este capítulo es el estudio de algoritmo para el cálculo del lugar geométrico de los puntos interiores a un polígono

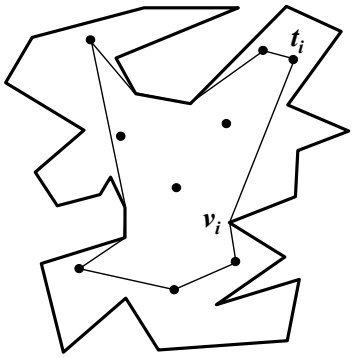


Figura 5.5: Cierre convexo relativo de un conjunto T de focos interiores a P

P iluminados por k focos, es decir, el cálculo de la intersección de sus *polígonos de visibilidad*. Estudiamos a continuación la idea general de otro algoritmo que llamaremos de *"doble barrido"*, pues realizamos dos barridos: uno hacia la derecha y otro hacia la izquierda. Presentamos en el Sección 5.4.3 este algoritmo para un único foco, generalizando en la Sección 5.4.4 a un conjunto de focos situados en los vértices de una región convexa, (es decir, situados en posición convexa), y que por ello titularemos *"región de visibilidad de un convexo por doble barrido"*. Finalmente en la Sección 5.4.6 presentamos el algoritmo para el caso en que las *luces-punto* o focos se encuentren en posición general en el interior de P.

5.4.3 Región de visibilidad de un punto por doble barrido

El algoritmo para el cálculo de la región de visibilidad de un punto en un polígono es bien conocido [72]. No obstante, en esta sección presentamos una variante del mismo cuya generalización permitirá resolver nuestro problema de iluminación múltiple.

Definición 5.4.7 *Dado q interior a P, decimos que v_i es un punto de giro hacia afuera, (resp. hacia adentro), respecto a q si los ángulos $\angle v_{i-1}qv_i$ y $\angle v_iqv_{i+1}$ tienen signos distintos y v_{i+1} y q están en lados opuestos (en el mismo lado) de la recta que pasa por la arista $v_{i-1}v_i$.*

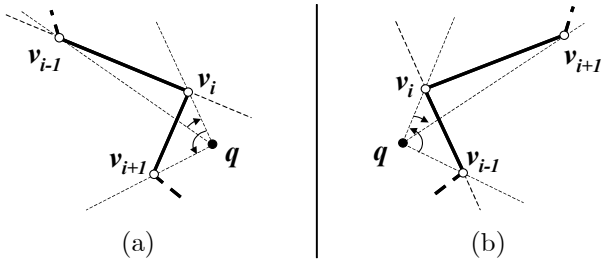


Figura 5.6: (a) Giro hacia adentro (b) Giro hacia afuera

Sea P^o el polígono P pero recorrido en sentido contrario. Entonces los puntos de giro hacia adentro para P son puntos de giro hacia afuera para P^o. Así podemos definir el siguiente algo-

ritmo para calcular el *polígono de visibilidad* de un punto q interior a un polígono P, $V(P,q)$.

Algoritmo I: Región de visibilidad de un punto por doble barrido

ENTRADA: Un polígono P de n vértices y una *luz-punto* q en su interior.
SALIDA: El *polígono de visibilidad* $V(P,q)$.

Básicamente recorremos el polígono P eliminando todos los puntos de giro hacia afuera.

- Dado v un punto de giro hacia afuera puede ocurrir que exista un punto $a(v)$ tal que se dan las siguientes circunstancias:

  1. $a(v)$ está en la intersección del borde de P con la recta que une a q con v.
  2. v está entre $a(v)$ y q
  3. No existe ningún punto del borde de P entre v y $a(v)$.

- Si recortamos P trazando cuerdas entre v y $a(v)$ para todo punto de giro hacia afuera v para el cual existe $a(v)$ conseguimos un polígono P' sin puntos de giro hacia afuera. La determinación de los puntos v y $a(v)$ se puede realizar en tiempo lineal haciendo un recorrido de los vértices v_i de P en el que en cada paso se calcule el ángulo $\angle v_{i-1}qv_i$ (por ello el nombre de barrido) mientras se mantienen unas estructuras con los candidatos a $a(v_i)$.

- Ahora basta con cambiarle la orientación a P' y aplicarle el barrido anterior para conseguir un polígono sin giros, ni hacia afuera ni hacia adentro, que es el polígono de visibilidad $V(P,q)$.

Este algoritmo se puede generalizar para calcular la región iluminada por focos situados en posición convexa. En este sentido hablaremos de visibilidad de un convexo.

5.4.4 Región de visibilidad de un convexo por doble barrido

Consideremos ahora un polígono convexo Q con k vértices y sea p un punto exterior a Q, (que podrá ser un vértice de nuestro polígono P). Como se muestra en la Figura 5.7 existe un único vértice w_i de Q tal que p está a la izquierda de $w_{i-1}w_i$ y a la derecha de w_iw_{i+1}, ya que Q es convexo. Sea $l(p) = w_i$. Similarmente existe un único vértice w_j de Q tal que p está a la derecha de $w_{j-1}w_j$ y a la izquierda de w_jw_{j+1}. Sea $r(p) = w_j$. Aprovechando la convexidad de Q se pueden determinar $l(p)$ y $r(p)$ en tiempo $\log k$.

Supongamos que Q está contenido en el polígono P de n vértices y que los puntos exteriores p para los que calcularemos $l(p)$ y $r(p)$, son los vértices del polígono P. Podemos definir el siguiente algoritmo de doble barrido para calcular la zona visible desde todos los vértices del convexo Q:

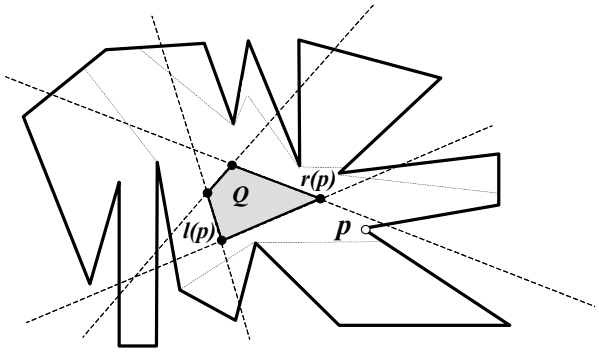


Figura 5.7: Vértices $l(p)$ y $r(p)$ en un convexo Q para un punto p exterior

---

Algoritmo II: Región de visibilidad de un convexo por doble barrido

ENTRADA: Un polígono P de n vértices y un conjunto $T = \{t_1, ..., t_k\}$ de k luces en posición convexa.
SALIDA: El *polígono de visibilidad* $V(P, T)$.

Igual que el Algoritmo I sólo que en el primer barrido los vértices v de giro hacia afuera y los puntos $a(v)$ se calculan con respecto a $l(v)$, y en el segundo barrido se calculan respecto a $r(v)$.

Este algoritmo tiene complejidad $O(n \log k)$, ya que se deben calcular los focos o luces punto $l(v)$ y $r(v)$ para cada vértices v del polígono P. En el caso general que estudiamos en la Sección 5.4.6, aparece un conjunto de focos en posición convexa, situados en una región del polígono que tiene todos sus vértices convexos, excepto dos. Estas zonas las llamaremos *orejas* de P y para solucionar el caso general, necesitamos aplicar el algoritmo de doble barrido para un convexo descrito, pero restringido a esta *oreja*. Esta situación se analiza en el siguiente punto.

5.4.5 Orejas

Ahora sea Q un polígono simple con vértices $t_0, \ldots, t_{k+2} = t_0$ tal que los vértices $t_1, \ldots, t_k$ son convexos, es decir, el ángulo que forman los vectores $\overline{t_{i-1}t_i}$ y $\overline{t_it_{i+1}}$ es positivo si $1 \leq i \leq k$.

Sea P un polígono de n vértices tal que: (1) la arista $\overline{t_{k+1}t_0}$ también es una arista de P, (2) Q es interior a P salvo por $\overline{t_{k+1}t_0}$. Nótese que si eliminamos la arista común $\overline{t_{k+1}t_0}$ obtenemos un polígono simple O cuyo interior está dado por la intersección del exterior de Q y el interior de P. Llamamos *oreja* a cualquier polígono que se pueda describir de esta forma.

Sea $\mathcal{C}$ la intersección de los semiplanos izquierdos de los vectores $\overline{t_kt_{k+1}}$, $\overline{t_{k+1}t_0}$ y $\overline{t_0t_1}$. Queremos hallar la región común de visibilidad de los focos $t_1, \ldots, t_k$ dentro de $P \cap \mathcal{C}$. Para ello basta generalizar cuidadosamente el Algoritmo II de modo que los vértices de giro hacia afuera v de P y los puntos $a(v)$ se calculen con respecto al vértice adecuado t_i de Q, $1 \leq i \leq k$. Llamemos a este algoritmo el **Algoritmo III**. Este algoritmo tiene complejidad $O(n \log k)$. El resultado de aplicarle el Algoritmo III a una oreja O es una oreja O'. La región común de visibilidad dentro

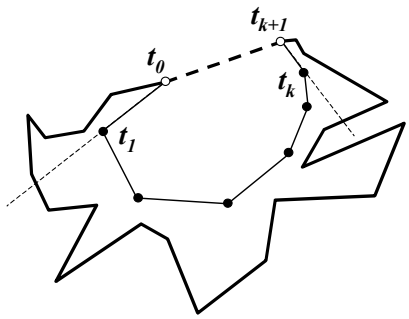


Figura 5.8: Ilustración de polígono oreja

de $P \cap \mathcal{C}$ se obtiene intersecando O' con $\mathcal{C}$.

5.4.6 Caso general

Consideremos finalmente el caso general en el que tenemos un conjunto de k puntos interiores a un polígono simple P. Sea $\mathcal{R}$ la región común de visibilidad de los k focos, es decir, $\mathcal{R} = \cap_i V(P, t_i)$.

Sea CCR el cierre convexo relativo del conjunto de los k focos dentro del poligono P. Asumimos que no hay focos en el interior de CCR ya que sabemos que los focos esenciales están siempre en el borde de éste.

Si CCR es convexo, entonces aplicamos el Algoritmo II. Si no, como se ilustra en la Figura 5.9, los vértices de CCR son de dos tipos: focos y vértices del polígono P. Los vértices convexos de CCR son exactamente los focos salvo en el caso en que CCR no es un polígono simple. En este caso, en virtud del Lema 5.4.6, añadimos los vértices cóncavos al conjunto de focos sin que la región de visibilidad común varie. Por lo tanto CCR está dividido en k' cadenas convexas maximales, $(1 \leq k' \leq k)$, cuyos vértices interiores son focos y cuyos vértices extremos son vértices del polígono y k' cadenas cóncavas cuyos vértices interiores son vértices del polígono y los extremos focos.

Sean, $C_1, \ldots, C_{k'}$ las cadenas convexas de CCR y $W_1, \ldots, W_{k'}$ las cóncavas. Sean p_i y q_i los puntos extremos de C_i y supongamos que p_i antecede a q_i al recorrer C_i. Llamemos O'_i al polígono con borde dado por la cadena convexa C_i con la orientación cambiada y por la cadena poligonal que coincide con P entre p_i y q_i. Sea O_i el polígono que se obtiene al aplicar el Algoritmo III a O'_i.

Consideremos los dos semiplanos S_i y S'_i izquierdos asociados a las aristas extremas de una cadena concava W_i. Como los vértices extremos de W_i son focos es claro que la región $\mathcal{R}$ ha de estar contenida en $S_i \cap S'_i$ en caso de que W_i realice un giro total menor que π. Si el giro es igual o mayor que π entonces $\mathcal{R} = \emptyset$. Por lo tanto asumimos que todas las cadenas cóncavas W_i realizan giros menores que π.

Sea $\mathcal{C} = \bigcap_i (S_i \cap S'_i)$. Sabido es que se puede calcular $\mathcal{C}$ en tiempo $k \log k$. Claramente $\mathcal{R} \subseteq \mathcal{C}$, luego $\mathcal{R} = \emptyset$ si $\mathcal{C} = \emptyset$. Además, se verifica que si $\mathcal{C} \neq \emptyset$ entonces los polígonos O'_i son orejas y por lo tanto también lo son los O_i.

Sea N el núcleo de CCR es decir, el conjunto de puntos interiores a CCR que ven a todos los vértices de CCR, $N = \mathcal{R} \cap CCR$. Como $\mathcal{R} \subseteq \mathcal{C}$ cabe preguntarse quien es $\mathcal{C} \cap CCR$. Observando también la Figura 5.9, podemos enunciar la siguiente proposición:

Proposición 5.4.8 *Sea P un polígono con n vértices y CCR el cierre convexo relativo de un conjunto de focos en su interior. Entonces si N es el núcleo de CCR, $\mathcal{C} = \bigcap_i (S_i \cap S_i')$ y $\mathcal{R} = \cap_i V(P, t_i)$, $\forall t_i \in T$, se tiene:*

$$\mathcal{C} \cap CCR = N \subseteq \mathcal{R} \subseteq \mathcal{C} \tag{5.4.1}$$

y si $\mathcal{C} \cap CCR = \emptyset$ y $\mathcal{R} \neq \emptyset$ entonces existe i tal que $\mathcal{R} \subseteq O_i$.

Y por lo tanto, la zona común de iluminación estará generada por el núcleo de CCR más las zonas iluminadas en las orejas que estén dentro del convexo $\mathcal{C}$. Así, podemos enunciar la siguiente proposición:

Proposición 5.4.9 *En las condiciones expuestas anterioremente se tiene que:*

$$\mathcal{R} = N \cup (\mathcal{C} \cap (\cup_i O_i)) \tag{5.4.2}$$

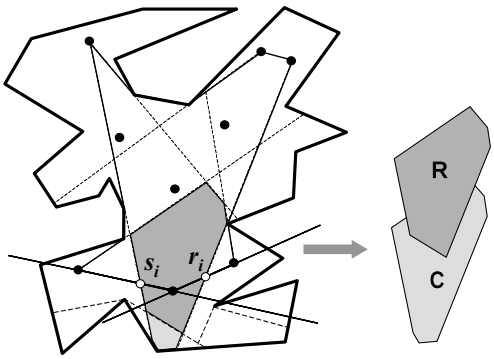


Figura 5.9: Caso general para el algoritmo de doble barrido

La pregunta que podemos hacernos ahora es si será posible recortar $\mathcal{C}$ con todos los O_i en tiempo $O(k+n)$. Efectivamente, esto será así porque basta intersecar la oreja O_i con aquella parte de $\mathcal{C}$ que sale de CCR atravesando la cadena convexa C_i.

Nótese que el borde de $\mathcal{C}$ puede intersecar a una cadena C_i en a lo sumo dos puntos, pues los semiplanos que definen a $\mathcal{C}$ pueden intersecar a C_i en a lo sumo un punto. Sean s_i y r_i los puntos donde el borde de $\mathcal{C}$ interseca a C_i saliendo y entrando de CCR respectivamente, si existen. La determinación del conjunto de los s_i y r_i se puede hacer en tiempo $O(k)$. Para ello intersecamos los conjuntos CCR y $\mathcal{C}$ mediante el conocido algoritmo de las hoces para dos convexos, pero adaptado a la presente situación: al avanzar por CCR saltamos de la última arista de cada cadena convexa a la primera de la siguiente, ya que sabemos que estas aristas junto con la cadena cóncava que limitan están en el exterior de CCR.

Si los bordes de CCR y $\mathcal{C}$ no se cortan, entonces $\mathcal{C} \subseteq CCR$ ó $\mathcal{R} = \mathcal{C} \cap O_i$ para algún i. Tomamos un vértice v cualquiera de $\mathcal{C}$ y en tiempo $O(n+k)$ determinamos si $v \in CCR$. Si pertenece, tendremos que $\mathcal{C} = N = \mathcal{R}$ y hemos terminado; si no pertenece, entonces $N = \emptyset$ y hay que hallar el único j tal que $\mathcal{R} \subseteq O_j$. Obsérvese que, por definición de $\mathcal{C}$, necesariamente $\angle w_i v w_{i+1}$ es positivo si w_i y w_{i+1} son vértices consecutivos de cualquier cadena cóncava W_h. Si además $\angle z_i v z_{i+1}$ fuese positivo para todo par de vértices consecutivos de cualquier C_j entonces tendríamos que v pertenece a N. Como estamos suponiendo $N = \emptyset$ concluimos que existe C_{j_0} tal que $\angle z_i v z_{i+1}$ es negativo. Este j_0 es único y $\mathcal{R} \subseteq O_{j_0}$. Para obtener $\mathcal{R}$ sencillamente intersecamos el convexo $\mathcal{C}$ con la oreja O_{j_0}, que es un polígono acotado por una cadena poligonal convexa y otra estrellada.

Si los bordes de CCR y $\mathcal{C}$ se cortan, podemos recortar $\mathcal{C}$ con todos los O_i en tiempo $O(k+n)$, pues ya conocemos los puntos r_i y s_i. Intersecamos la oreja O_i con la parte de $\mathcal{C}$ que sale de CCR atravesando la cadena convexa C_i. Esto es poco más que intersecar un convexo con un estrellado y se realiza en $O(k_i + n_i)$ pasos, donde k_i es el número de vértices de $\mathcal{C}$ entre s_i y r_i, n_i el número de vértices de la oreja O_i. Como $\sum k_i \leq k$, $\sum n_i \leq n$ se sigue que la intersección de $\mathcal{C}$ con todos los O_i se realiza en tiempo $O(k+n)$. El conjunto que se obtiene al recortar $\mathcal{C}$ de esta manera es la región común de visibilidad $\mathcal{R}$.

Concluimos por lo tanto con el siguiente teorema, que determina la complejidad de nuestro algoritmo si no tenemos en cuenta el cálculo del *cierre convexo relativo* del conjunto de focos, interiores a P.

Teorema 5.4.10 *Si se conoce el cierre convexo relativo del conjunto de focos, se puede hallar la región común de visibilidad $\mathcal{R}$ en tiempo $O(k+n)$.*

5.5 Conclusiones

Para resolver el problema de la *iluminación múltiple*, hemos obtenido dos algoritmos. Por una parte, un algoritmo incremental de complejidad $O(kn)$ y, por otra, un algoritmo de doble barrido de complejidad $O(n + k\log(kn))$. Ambos calculan la intersección de k *polígonos de visibilidad* en un polígono. En la siguiente tabla se exponen a modo de resumen los resultados obtenidos en este capítulo.

Tipo de algoritmo	*Posición luces*	p-Pvk(P)
Incremental	interior polígono P	$O(kn)$
DobleBarrido	interior polígono P	$O(n + k\log(kn))$

Tabla 5.1: Resultados sobre iluminación múltiple

Parte II

Métodos aproximados en problemas geométricos

Capítulo 6

Introducción

Algunos problemas en el ámbito de la *Geometría Computacional* son de naturaleza $\mathcal{NP}$-dura. No obstante en muchas ocasiones, se tiene la necesidad de obtener una respuesta al problema aunque ésta sea aproximada. Así, para este tipo de problemas tiene sentido el estudio de algoritmos aproximados para solucionarlos. Presentamos en esta parte de la memoria soluciones heurísticas a problemas geométricos, que o bien, son $\mathcal{NP}$-duros o se desconoce hasta el momento una cota óptima que lo solucione.

Esquemáticamente se han abordado dos problemas: el problema MinN-p-Pvk(P), que consiste en minimizar el número de luces que iluminan completamente un polígono P de n vértices y el problema MaxA-p-Vor(N), que buscará donde situar un nuevo punto p en un diagrama de *Voronoi* ya dado de N puntos, tal que la región de *Voronoi* asociada a p en el nuevo diagrama, tenga área máxima. Para solucionar el problema MinN-p-Pvk(P) ha sido necesario atacar previamente dos problemas: el problema MaxA-p-Pv1(P), que busca el punto interior a un polígono P de n vértices, cuya área iluminada sea máxima, (para el que se presentan cuatro métodos heurísticos y una prueba experimental sobre un algoritmo exacto polinomial), y el problema MaxA-p-Pvk(P, k), que maximizará el área iluminada por k luces interiores a P.

Dada la naturaleza aproximada de los algoritmos presentados se describen también los resultados experimentales obtenidos por las implementaciones heurísticas construidas para solucionar estos mismos problemas utilizando técnicas como son *Recocido Simulado*, (*simulated annealing SA*), y los *algoritmos genéticos* introducidas en el Capítulo 1. Además de estas técnicas generales se presentan métodos construidos expresamente para estos problemas.

6.1 Situación previa

Entre los problemas clásicos en *Geometría Computacional* y más concretamente dentro del campo de la *visibilidad* se encuentran los problemas de minimización del número de *luces-punto* y *luces-vértice* que pueden iluminar completamente un polígono P con n vértices. Formalmente estos problemas que denotaremos con MinN-v-Pvk(P) y MinN-p-Pvk(P) los podemos enun-

ciar de la siguiente manera:

minimización del número de luces vértice que iluminan un polígono
MinN-v-Pvk(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cuál es el número mínimo k de *luces vértice* necesarios para iluminar el polígono P?

minimización del número de luces punto que iluminan un polígono
MinN-p-Pvk(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cuál es el número mínimo k de *luces punto* necesarios para iluminar el polígono P?

En la Figura 6.1 presentamos un polígono generado aleatoriamente P con 50 vértices, como ejemplo de entrada para estos dos problemas y por tanto para los problemas que mencionaremos más adelante MaxA-p-Pv1(P) y MaxA-p-Pvk(P, k).

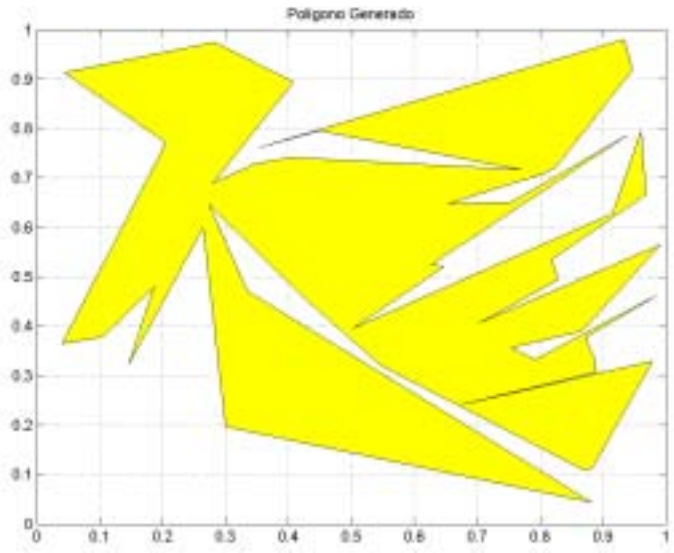


Figura 6.1: Una entrada para problema MinV-p-PV1(P) con un polígono P de 50 vértices

Los dos problemas de minimización mencionados han sido estudiados por Lee y Lin [71] y utilizando una reducción al problema 3-Sat [45], probaron que son problemas $\mathcal{NP}$-duros. Si nos restringimos a polígonos ortogonales, Schuchardt y Hecker [92] han demostrado que minimizar el número de *luces punto* y *vértice* que iluminan todo el polígono es también un problema $\mathcal{NP}$-duro.

Por tanto, cabe preguntarse sobre el diseño de algoritmos aproximados o heurísticos que solucionen estos problemas. En este sentido, existen pocos análisis que ataquen estos problemas de forma heurística. Encontramos en la bibliografica estudios de Ghosh [49], donde se presenta una heurística *greedy* de complejidad $O(n^5 \log n)$ que produce como máximo $O(\log n)$ veces el mínimo número de *luces-vértice* necesarias para vigilar un polígono P con n vértices. Mejorando estos

resultados, Efrat y Har-Peled han obtenido recientemente una solución $O(\log k_{opt})$-aproximada [37].

Respecto a puntos en el interior del polígono P también existen pocos trabajos. Una heurística *greedy* que ha sido estudiada consiste en buscar en primer lugar el punto de máxima iluminación en el interior de P, para a continuación repetir el proceso sobre el polígono que se obtiene al eliminar en P el *polígono de visibilidad* del punto obtenido en el primer paso. Utilizando este algoritmo Hochbaum y Pathria [57] construyen una $(1 - 1/e)$-aproximación para el problema más general Set-Cover. Por otra parte, Ntafos y Tsoukalas [78] describieron como buscar para cada $\delta > 0$ un punto x, interior a un polígono P, cuyo área iluminada sea $(1 - \delta)\mu_{opt}$. Basándose en estos resultados Cheong y otros en 2004 [23], han demostrado que para $\delta > 0$ se puede computar en $O((n^2/\delta^4)\log^3(n/\delta))$ la búsqueda de un punto $x \in P$ tal que $\mu(x) \geq (1 - \delta)\mu_{opt}$, siendo $\mu(x)$ el área iluminada por el punto x, obteniendo mediante técnicas *greedy* un algoritmo también aproximado para la búsqueda de la k *luces-punto* de máxima iluminación. Este problema directamente relacionado con MinN-p-Pvk(P), lo denotaremos en la siguiente sección con MaxA-p-Pvk(P, k).

La utilización de técnicas *voraces* o *greedy* para solucionar estos problemas puede producir un error tan grande como se quiera sobre ciertos polígonos, como se muestra en el polígono de la Figura 6.2, en el que utilizando técnicas *greedy* se consigue una 4-*aproximación*, utilizando el cociente $R_A(I)$ descrito en el Capítulo 1. Por ello, presentamos en esta memoria soluciones heurísticas para el problema MinN-p-Pvk(P), atacando previamente los problemas MaxA-p-Pv1(P) y su generalización MaxA-p-Pvk(P, k). Se dan respuestas utilizando diferentes técnicas heurísticas como son por ejemplo *simulated annealig* y los *algoritmos genéticos* descritos también en el Capítulo 1.

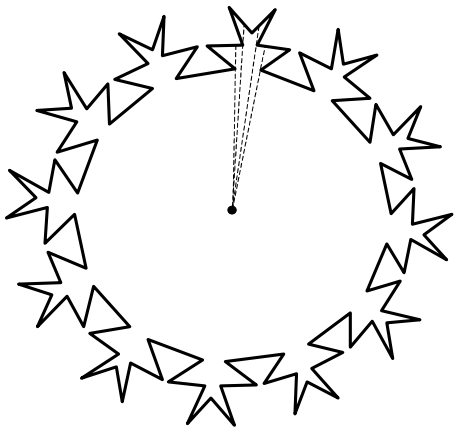


Figura 6.2: Un polígono P en el que una técnica greedy produce una 4-aproximación

Respecto a los problemas tratados no relacionados con los conceptos de *visibilidad,* se han estudiado problemas enmarcados dentro de los *Diagramas de Voronoi* . Se conocen algoritmos óptimos para calcular el *diagrama de Voronoi* de una nube de puntos [99]. Sin embargo, dado un *diagrama de Voronoi* de $n - 1$ puntos, encontrar un punto q tal que el área de la zona asignada a este punto en un nuevo *diagrama de Voronoi* sea máxima es un problema que se

encuentra abierto y que trataremos en esta memoria de forma aproximada. Formalmente lo podemos enunciar de la siguiente manera:

maximización de la región de voronoi asociada a un nuevo punto

MaxA-p-Vor(N):
ENTRADA: El *diagrama de Voronoi* de un conjunto N de puntos en el plano..
PREGUNTA: ¿Cuál es el punto q, tal que el área de la *región de Voronoi* que se le asocia en el *diagrama de Voronoi* de la nube $N \cup \{q\}$ sea máxima?

En la Figura 6.3 presentamos una entrada para este problema sobre un conjunto N de cardinal 50, generado aleatoriamente. En el Apéndice B se encuentran los códigos implementadios para la construcción de los *diagramas de Voronoi* de nubes de puntos generadas aleatoriamente y que se utilizarán para realizar las experimentaciones de las heurísticas sobre este problema.

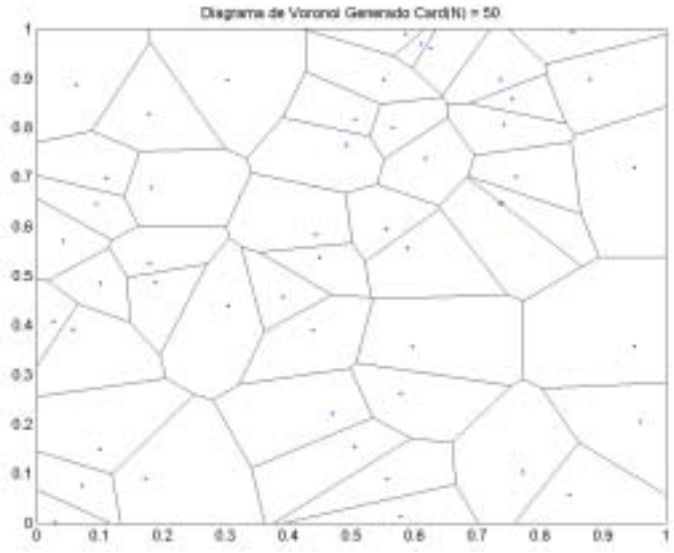


Figura 6.3: Una entrada para problema MaxA-p-Vor(C) con Card(C)$= 50$

Recientemente Dehne y otros [30], han estudiado este problema para el caso concreto de que los vecinos del nuevo punto q estén en posición convexas. En esta memoria presentamos soluciones heurísticas para problema general, sin exigir ninguna condición sobre el punto q. Estas soluciones pueden permitir realizar un estudio para la búsqueda de una solución exacta del problema, o para la demostración de su naturaleza $\mathcal{NP}$-dura.

6.2 Problemas estudiados

Como se ha mencionado en la sección anterior los problemas tratados en esta parte de la memoria se enmarcan dentro de dos ámbitos: *visibilidad* y *geometría computacional* en general. Como se ha mencionado en el apartado anterior dentro de los problemas de *visibilidad* nuestro objetivo final es dar respuesta de forma aproximada al problema MinN-p-Pvk(P); sin embargo, para

solucionar este problema necesitamos abordar previamente otros problemas, que nos ayudarán a atacarlo, y que detallamos a continuación:

búsqueda del punto de máxima iluminación interior a un polígono
MaxA-p-Pv1(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde se debe colocar una *luz punto* en el interior del polígono P para que el área iluminada por dicha luz sea máxima?

El análisis y estudio de este problema lo realizamos en el Capítulo 7, donde se presentan cuatro métodos heurísticos para solucionarlo, (exponiendo la comparativa de los resultados experimentales obtenidos). Estas técnicas nos permitirán construir la superficie que se obtiene asociando a cada punto x de un polígono P su área iluminada y que denominaremos *superficie de áreas*. Observando dichas superficies comprobamos que aparece una descomposición $\mathcal{S}$ del polígono P en regiones de visibilidad. Dicha descomposición está contenida en otra descomposición $\mathcal{T}$ ya estudiada por Bose y otros [15]. Utilizando estas descomposición presentamos también en el Capítulo 7 un estudio para la búsqueda de un algoritmo exacto polinomial que solucione el problema MaxA-p-Pv1(P). Todas las experimentaciones, tanto de las heurísticas como el estudio de la *superficie de áreas*, se han realizado sobre conjuntos de polígonos generados aleatoriamente, mediante un generador aleatorio *RPG*, (implementado en esta memoria), como se menciona en la Sección 6.3.

Una vez realizado el estudio del problema MaxA-p-Pv1(P), nos planteamos la búsqueda heurística, (ya que nuestro problema final es $\mathcal{NP}$-duro), de los k, con $2 \leq k \leq n$, puntos cuya zona de iluminación conjunta dentro de un polígono P de n vértices sea máxima. Este problema lo podemos enunciar de la siguiente manera:

búsqueda de los k puntos de máxima iluminación interiores a un polígono
MaxA-p-Pvk(P, k):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde se deben colocar k *luces punto* en el interior del polígono P para que el área iluminada por dichas luces sea máxima?

Responder positivamente al problema MaxA-p-Pvk(P, k) es el paso previo para solucionar heurísticamente nuestro problema final MinN-p-Pvk(P). Ambos problemas se abordan en el Capítulo 8. Se presentan algoritmos aproximados teóricos y se describen e implementan técnicas heurísticas utilizando entre otras *simulated annealig SA* y *algoritmos genéticos GA*. La elección de este tipo de algoritmos para solucionar nuestros problemas se basa en que son considerados como una de las técnicas heurísticas más potentes y eficaces, lo que está motivando que estos algoritmos sean utilizandos, cada vez con más frecuencia, en diferentes campos de investigación [62, 74, 87].

Por otra parte, para resolver con técnicas heurísticas el problema MaxA-p-Pvk(P, k) necesitamos calcular el área iluminada por k focos interiores a un polígono, es decir, necesitamos calcular la unión de k *polígonos de visibilidad* correspondientes a k *luces punto* interiores al polígono P. Este problema ha sido solucionado por Cheong y Oostrum [83], con un algoritmo

de complejidad $O(((k(h+1)^2 + kn\log k)\log(k+n))$, siendo n el número de vértices y h el de agujeros del polígono P. Sin embargo, dada la necesidad de implementación del algoritmo para solucionar heurísticamente este problema, hemos utilizado una pequeña modificación del algoritmo de recorte de *Weiler-Atherton* [56] para la unión de *polígonos de visibilidad,* (que puede ser un polígono con agujeros), como se describe en el Capítulo 8.

Para finalizar presentamos en el Capítulo 9 las descripciones y los resultados experimentales obtenidos por los métodos diseñados para solucionar el problema MaxA-p-Vor(C). Para ello, se utilizan *diagramas de Voronoi* construidos para nubes de puntos generadas aleatoriamente.

6.3 Sobre las implementaciones

Todos los métodos heurísticos presentados en esta segunda parte de la memoria han sido implementados y se encuentran detallados en el Apéndice B. Por ello, no se expondrán detalles de implementación tales como estructuras de datos, funciones, mapas de memoria, etc.., en las descripciones de los métodos heurísiticos. En el Apéndice B se presentan también las implementaciones diseñadas para la obtención de los resultados experimentales, así como las notebooks implementadas con el paquete matemáticos MatLab para la visualización de los resultados obtenidos por las heurísticas: *polígonos de visibilidad*, unión de *polígonos de visibilidad*, *nubes aleatorias*, *diagramas de Voronoi*, etc..

El lenguaje utilizado para todas los programas de ordenador ha sido C++ por considerarlo un lenguaje de alto nivel con la potencia necesaria para abordar los los problemas a estudiar. Se ha elegido MatLab para construir los programas de visualización, por su similitud con C++ y por la versatilidad de dicha herramienta.

Para la obtención de resultados experimentales de los problemas MaxA-p-Pv1(P), MaxA-p-Pv1(P) y MinN-p-Pvk(P), se ha implementado un *generador aleatorio de polígonos* que denotaremos con *RPG*, cuyos detalles se encuentran en el Apéndice A y cuyo código se muestra en el Apéndice B de listados. *RPG* está basado en el estudio sobre generación aleatoria de polígonos realizado por Thomas Auer y Martin Held en [6].

Igualmente para obtener conclusiones del problema MaxA-p-Vor(N), se ha implementado un algoritmo incremental que construye el *diagrama de Voronoi* de una nube de puntos generada aleatoriamente. Para que este algoritmo resulte realmente eficiente es preciso que las estructuras de datos que acompañen al *diagrama de Voronoi* tengan información acerca de las regiones adyacentes. De este modo, una vez localizado un punto q y construido el segmento bisector, se puede pasar de una región a su contigua en tiempo constante. Los detalles de estas implementaciones así como de todas la heurísticas relacionadas con este problema se presentan también en el Apéndice B, omitiendo dichas explicaciones en la descripción de los métodos heurísticos que se presentan en el Capítulo 9 para solucionar el problema MaxA-p-Vor(N).

Capítulo 7

Buscando el punto de máxima iluminación

Proponemos en este capítulo soluciones al problema de búsqueda de la *luz-punto* de máxima iluminación para un polígono P de n vértices. Este problema lo hemos denotado en el capítulo anterior como MaxA-p-Pv1(P) y se solucionará en primer lugar utilizando 4 heurísticas diferentes haciendo un estudio comparativo de los resultados obtenidos por ellas. Las dos primeras están basadas en técnicas generales como son el *recocido simulado*, (nos referiremos siempre a él con *simulated annealing),* y los *algoritmos genéticos*, descritos en el Capítulo 1. Las otras dos, son heurísticas diseñadas expresamente para solucionar MaxA-p-Pv1(P). Se realiza un estudio detallado de los resultados obtenidos por ellas, utilizando para ello el generador aleatorio de polígonos *RPG* descrito en el Apéndice A, y cuya implementación se puede encontrar en el Apéndice B. Las soluciones propuestas en este capítulo para el problema MaxA-p-Pv1(P) tendrán continuidad en el Capítulo 8, donde estudiaremos la búsqueda del conjunto de k *luces-punto* que maximicen su área iluminada, (MaxA-p-Pvk(P, k)) y que será el paso previo para dar respuesta al problema MinN-p-Pvk(P).

Profundizando más en el estudio MaxA-p-Pv1(P) estudiamos también un algoritmo determinista polinomial basado en una descomposición del polígono P en *regiones de visibilidad.* Dicha descomposición está relacionada con los resultados presentados por Bose en [15]. Concluimos el capítulo exponiendo algunos resultados experimentales sobre el porcentaje de área que puede iluminar una *luz-punto* en un polígono P, respecto al número n de vértices de éste. Este problema lo denotaremos con PorA-p-Pv1(P) y se comprobará utilizando técnicas heurísticas basada en *SA* y *RS*, que este porcentaje disminuye siguiendo una curva exponencial.

7.1 Introducción

Sabemos que $\lfloor n/3 \rfloor$ luces es el mínimo número de guardias que vigilan todo polígono de n vértices [42]. Sin embargo no todos los polígonos requieren este número de guardias para vigilarlo. Por tanto, tiene sentido plantear el problema algorítmico siguiente: *dado un polígono calcular el*

mínimo número de luces que lo iluminan. Como se vió en el Capítulo 6, este problema que hemos denotado con MinN-p-Pvk(P), ha sido estudiado por Lee y Lin [71] y utilizando una reducción al problema 3-Sat [45], probaron que es un problema $\mathcal{NP}$-duro. Como primera aproximación planteamos en este capítulo la búsqueda de la *luz-punto* interior a un polígono P de máxima iluminación. Este problema lo podemos formular de la siguiente manera:

búsqueda del punto de máxima iluminación interior a un polígono MaxA-p-Pv1(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿En qué punto interior a P debo colocar una luz para que el área iluminada por dicha luz sea máxima?

Abordamos este problema de dos formas diferentes: mediante técnicas heurísticas y mediante un algoritmo polinomial cuya demostración se presenta de forma experiemental. Respecto a la primera forma desarrollamos cuatro técnicas heurísticas que dan respuestas aproximadas al problema. Dos de estas técnicas están basadas en métodos generales de aproximación heurísticas y las otras dos han sido desarrolladas expresamente para solucionar el problema MaxA-p-Pv1(P).

1. Las técnicas generales utilizadas han sido *simulated annealing* y *algoritmos genéticos*. Una introducción a estas técnicas se puede encontrar en el Capítulo 1.

   - *Simulated Annealing-SA:* Esta técnica está basada en el templado o enfriado controlado con el que se producen determinadas sustancias; es el caso, por ejemplo, del proceso de cristalización del vidrio. En nuestro caso el mecanismo de enfriamiento se asocia con la búsqueda de puntos interiores al polígono P, de tal forma que cuanto más baja sea la temperatura a la que sometemos el proceso mayor será la probabilidad de que el punto producido por el algoritmo se encuentre próximo al óptimo. En la Sección 7.2 estudiamos las adaptaciones de MaxA-p-Pv1(P) a SA: conjunto S de configuraciones, vecindad, estrategias de templado y temperatura inicial. En la Sección 7.6 se realiza un estudio detallado de los resultados de la heurística sobre una base de datos de polígonos generados aleatoriamente con el generador aleatorio RPG atendiendo a distintos criterios de estos elementos.
   - *Algoritmos Genéticos-GA:* Como se mencionó en el Capítulo 1 los algoritmos genéticos han sido muy útiles en el tratamiendo de multitud de problemas. En este capítulo haremos uso de tales algoritmos para aproximar el problema MaxA-p-Pv1(P). En la Sección 7.3 presentamos la adaptación del esquema general de GA a nuestro problema, determinando el genotipo del problema, la función objetivo utilizada y los operadores de selección, cruce y mutación diseñados. Es importante destacar que el esquema genético que presentamos para solucionar MaxA-p-Pv1(P), se generalizará para solucionar MaxA-p-Pvk(P, k) como su extensión natural. Además dicho esquema se utilizará también para solucionar el problema MaxA-p-Vor(N) que buscará el punto que maximice el área de su región de *Voronoi*. Por tanto, podemos considerar el algoritmo genético que presentamos en este capítulo un esquema general que podrá

ser adaptado a problemas geométricos de distinta naturaleza y no sólo a problemas de *visibilidad*. En la Sección 7.6 presentamos también los resultados aportados por *GA* frente a las demás heurísticas expuestas.

2. Las dos técnicas heurísticas específicas diseñadas están relacionadas con la búsqueda aleatoria, (*random search, RS*), sobre el conjunto de soluciones del problema.

   - *Random Search-RS:* Dado un polígono P de n vértices podemos realizar una búsqueda aleatoria sobre el conjunto de puntos interiores a P para maximizar la región iluminada por uno de éstos. Evidentemente el tamaño del conjunto de puntos aleatorios sobre el que realizar la búsqueda determinará en gran medida la optimalidad del método. En la Sección 7.4 presentamos esta heurística y en la Sección 7.6 los resultados comparativos respecto a *SA* y *GA*.
   - *Método del Gradiente-GRAD:* Para todo punto p interior a un polígono P, que no sea un máximo local respecto al problema MaxA-p-Pv1(P), existirá una dirección $\overrightarrow{v}$ tal que si movemos p en esa dirección el área iluminada aumente, es decir, que p puede deslizarse siguiente un vector de "gradiente positivo", respecto al área iluminada. Siguiente esta idea presentamos en la Sección 7.5 lo que denominamos el *método del gradiente-GRAD,* realizando posteriormente un estudio comparativo de los resultados aportados sobre la misma base de datos de polígonos generados con la que se han realizado los estudios de *SA*, *GA* y *RS* y que han sido generados por un generador aleatorio de polígonos desarrollado también en esta memoria y que denominamos *RPG,* (ver Apéndice A).

Las técnicas que aparecen en la bibliografía para solucionar el problema MaxA-p-Pv1(P) y su generalización MaxA-p-Pvk(P, k) son algoritmos teóricos aproximados que utilizan técnicas *greedy* para solucionar el problema, según se menciona en el Capítulo 6 [23]. Aunque en esta memoria aportamos heurísticas que solucionan de forma real dicho problema, podemos preguntarnos si existirá un algoritmo exacto que lo solucione. En este sentido, presentamos en la Sección 7.8 un estudio sobre un algoritmo exacto de complejidad polinomial. Si a cada punto p interior a un polígono P le asociamos su porcentaje de área iluminada respecto a P, podemos construir una superficie que hemos denominado *superficie de áreas*, (aportamos además en el Apéndice B las implementaciones necesarias para construir dichas superficies). Observando dichas superficies comprobamos que existe una partición de P respecto a la iluminación. Esta partición, que denotaremos con $\mathcal{S}$, está contenida en una descomposición del polígono P estudiada por Bose [15], en un conjunto de regiones $\mathcal{T} = \{R_1, R_2, ..., R_m\}$, (ver Figura 7.21), que tienen buenas propiedades respecto a la vigilancia o iluminación. Dicha descomposición tiene asociado un digrafo $G^{\mathcal{T}}$, de tal forma que a cada region R_i se le asocia un nodo n_i del grafo; un nodo n_i se relaciona con otro nodo n_j si las regiones asociadas R_i y R_j comparten un lado y desde la región R_i se ve un vértice más o un vértice menos de P que desde la región R_j. De esta manera aparecerán en el grafo $G^{\mathcal{T}}$ *nodos-fuente* que corresponde a regiones de $\mathcal{T}$ tal que todas las regiones que comparte algún borde con ella ven un vértice menos. Evidentemente si P tiene núcleo, sólo tendremos un *nodo-fuente* en $G^{\mathcal{T}}$, asociado precisamente a dicho núcleo. Veremos en la Sección 7.8 que los

vértices de las regiones R_i, son puntos especiales respecto a la iluminación y determinarán el punto o la *luz-punto* de máxima iluminación buscada, que permitirá construir un algoritmo de complejidad $O(n^4)$.

Una vez encontrado el punto de máxima iluminación, es decir, una vez solucionado el problema MaxA-p-Pv1(P), nos preguntamos sobre el porcentaje máximo de área del polígono P que puede iluminar una *luz-punto*. Este problema lo hemos denotado con PorA-p-Pv1(P) y para dar respuesta a él se ha realizado un estudio experimental utilizando *SA* y *RS*, (cuyos resultados se exponen en la Sección 7.6), en el que probamos que de forma aproximada que dicho porcentaje es de orden $O(n^2)$, siendo n el número de vértices de P. Evidentemente, una pregunta que prodremos hacernos en el Capítulo 8 es si seguirá siendo cuadrático el porcentaje de área iluminada por k *luces-punto* interiores a P. A esta pregunta responderemos en dicho capítulo.

Para la generación aleatoria de los polígonos se ha implementado un generador aleatorio de polígonos descrito por Auer y Held en [6] y expuesto en el Apéndice A de esta memoria. Los códigos en C++ de las implementaciones, tanto de los algoritmos como del Generador Aletario de polígonos, (*RPG*), se encuentran detallados en el Apéndice B. Todas las experimentaciones han sido realizadas sobre una computadora Pentium V a 2.5Gh.

7.2 El problema MaxA-p-Pv1(P) con simulated annealing-SA

Retomando lo descrito en el Capítulo 1 sobre la técnica heurística de *simulated annealing*, (*recocido simulado*, *SA*), podemos abordar, siempre de forma aproximada, problemas de optimización combinatoria que pueden plantearse de la siguiente forma:

Dado un espacio finito de *configuraciones* o soluciones $S = \{x_1, ..., x_m\}$, donde m es la dimensión de dicho espacio, dada una función de costes $C : S \to \mathbb{R}$, determinar $x^* \in S$ tal que $C(x^*) \leq C(x) \quad \forall x \in S$.

Si $x \in S$ es la configuración inicial y T la temperatura en cada iteración siendo $T_0 > 0$ la temperatura inicial, el esquema general del *simulated annealig*, es el siguiente:

```
[01]  do
[02]    {do
[03]       {Genera solución y ∈ Vecindad(x) ⊂ S;
[04]        Evalúa δ ← C(x) − C(y);
[05]        if (δ < 0) x ← y
[06]        else
[07]           if ((δ ≥ 0) ∧ (U(0,1) < e^(−δ/T))) x ← y;
[08]        n ← n + 1;
[09]       }while (n ≤ N(T));
[10]    Disminuir T;
[11]    }while (parada==false);
```

Utilizando esta técnica heurística se ha abordado el problema MaxA-p-Pv1(P) implementando un algoritmo en el lenguaje de programación C++ y adaptando los elementos de la heurís-

tica a nuestro problema como se indica en el siguiente apartado. Los códigos implementados se encuentran en el Apéndice B, con los correspondientes comentarios de implementación.

7.2.1 Adaptación del problema

Simulated annealing, (*recocido simulado, SA*), se puede considerar como una variación de la simulación de *Monte Carlo*. En cada iteración, cada átomo es sometido a un desplazamiento aleatorio que provoca un cambio global en la energía del sistema (δ). Si $\delta < 0$, se acepta el cambio; en caso contrario, el cambio se acepta con una probabilidad $e^{(-\delta/T)}$, siendo T la temperatura absoluta.

Recordemos el enunciado concreto del problema:

búsqueda del punto de máxima iluminación interior a un polígono

MaxA-p-Pv1(P):

ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿En qué punto interior a P se debe colocar una luz para que el área iluminada por dicha luz sea máxima?

Por tanto, la entrada del problema serán simplemente los n vértices, (dados en sentido positivo), del polígono P y la salida serán las coordenadas (x, y) de un punto p interior a P. Mostramos en la Figura 7.1 una entrada para MaxA-p-Pv1(P) y la salida producida por la heurística, visualizando el punto de máxima iluminación y su correspondiente *polígono de visibilidad* mediante los programas de visualización implementados con el paquete informático MATLAB y cuyos códigos también se encuentran en el Apéndice B.

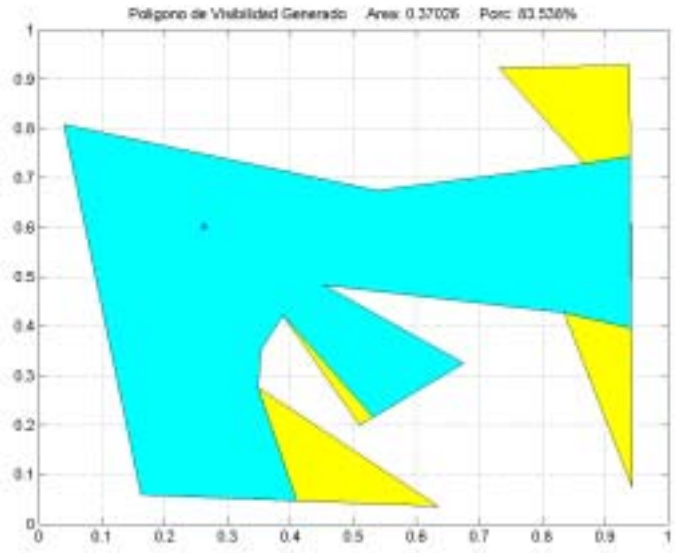


Figura 7.1: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con SA

Los elementos de configuración de la heurística adaptados a nuestro problema se pueden describir de la siguiente manera:

Conjunto S de configuraciones:

El conjunto de configuraciones o soluciones factibles de nuestro problema serán todos los puntos que se encuentran en el interior del polígono P. Así, consideraremos que el conjunto de configuraciones es infinito y que cada elemento, (cada punto), viene determinado por sus coordenadas (x, y) en el plano.

$$S = \{p_1 = (x_1, y_1), p_2 = (x_2, y_2), ..., p_n = (x_n, y_n), ...\} \tag{7.2.1}$$

Función de coste C:

La función de coste $C : S \to \Re$ asignará a cada elemento de la configuración S un valor real. En nuestro caso para cada $p_i \in S$ la función de costes producirá un valor que representa el área iluminada por una *luz-punto* situada precisamente en el punto p_i, es decir, representará el área del *polígono de visibilidad* del punto p_i, que es un punto interior a P. Para la obtención de este valor se ha implementado un algoritmo lineal, (ver [72]), para el cálculo del *polígono de visibilidad* $V(P, p_i)$ de un punto $p_i \in S$. Como todo *polígono de visibilidad* es un polígono estrellado desde el punto de iluminación, $V(P, p_i)$ es un polígono estrellado desde p_i y por tanto se puede descomponer en la unión de los triángulos T_j que se formán al unir cada vértice de $V(P, p_i)$ con p_i, $(V(P, p_i) = \bigcup_{j=1}^{k} T_j)$. La función de costes para p_i será la suma del área de los triángulos T_j con $i = 1, ..., k$.

$$C(p_i) = \sum_{j=1}^{k} Area(T_j) \tag{7.2.2}$$

Una vez que conocemos la función de costes C, analizamos en el siguiente apartado la función de vecindad utilizada, es decir, la función que producirá el siguiente elemento de S a analizar a partir del elemento p_i.

Vecindad de cada configuración:

Según se indica en el esquema general de *simulated annealing SA,* para cada elemento $x \in S$, se debe obtener un elemento $y \in S$ en la vecindad de x que será el elemento a analizar en la siguiente iteración de la heurística. Así, para cada punto $p_i = (x_i, y_i)$ debemos determinar mediante la función de vecindad el siguiente punto $p'_i = (x'_i, y'_i)$ a analizar. La bondad de los resultados que produzca la heurística dependerá en gran medida de la función de vecindad utilizada.

En nuestro algoritmo, dado un punto $p_i = (x_i, y_i)$ la función de vecindad calcula las coordenadas del punto p'_i sumando a cada coordenada de p_i un valor real que sigue la distribución $N(0, 1)$. Para obtener números aleatorios distribuidos uniformemente hemos utilizado la propia función `rand` de que dispone el lenguaje C y para los números aleatorios distribuidos normalmente hemos utilizado el método de Box-Muller [16]. El número $x \in [a, b)$ seguirá la distribución normal $N(0, 1)$ si

$$P\,[x < y] = \frac{1}{\sqrt{2\pi}} \int_a^y e^{-t^2} dt \quad \forall y \in [x, b) \tag{7.2.3}$$

El método de Box-Muller nos permite obtener, a partir de dos números u_1 y u_2 distribuidos uniformemente, otros dos números n_1 y n_2 normalmente distribuidos, mediante las fórmulas

$$\begin{cases} n_1 = \sqrt{-2 * \ln(u_1)} sen(2\ \pi\ u_2) \\ n_2 = \sqrt{-2 * \ln(u_1)} \cos(2\ \pi\ u_2) \end{cases} \tag{7.2.4}$$

Según la implementación realizada los polígonos que generamos en nuestra investigación se encuentran situados en el plano real y en el cuadrado de vértices $\{(0,0),(1,0),(1,1),(0,1)\}$. Por ello si sumamos a cada coordenada del punto $p_i = (x_i, y_i)$ un valor que siga la distribución $N(0,1)$, el punto obtenido será un punto exterior al polígono P con una probabilidad alta. Así, una vez generados los valores n_1 y n_2 estos han sido divididos por un *factor de vecindad* `div`, (que irá aumentando con cada temperatura), con lo que el punto obtenido tiene una probabilidad más alta de ser interior a P. Hemos considerado en nuestro algoritmo `div` $= 10.0$, (recuérdese que nos encontramos en el cuadrado de vértices $\{(0,0),(1,0),(1,1),(0,1)\}$), con lo cual conseguimos que los puntos estudiados no estén concentrados para evitar entrar en máximos locales. En forma de pseudocódigo la función que genera el vecino $p'_i = (x'_i, y'_i)$ al punto $p_i = (x_i, y_i)$, tiene la siguiente forma:

Función Generar-Vecino

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$, un punto $p_i = (x_i, y_i)$ y un factor de vecindad `div`.
SALIDA: Un punto $p'_i = (x'_i, y'_i)$ vecino de p_i e interior a P.

```
[01]  u1 ←rand()*1.0/RAND_MAX;
[02]  u2 ←rand()*1.0/RAND_MAX;
[03]  do
[04]    {n1 ←Sqrt(-2.0*ln(u1))*sin(2 * π * u2);
[05]     n2 ←Sqrt(-2.0*ln(u1))*cos(2 * π * u2);
[06]     x'i = xi + n1/div;
[07]     y'i = xi + n2/div;
[08]    }while (p'i = (x'i, y'i) exterior a P);
[09]    div ← div/0.9999;
[10]    return p'i;
```

Pero el algoritmo heurístico necesita unos datos iniciales de los que partir, es decir, necesita una configuración inicial. Esta configuración inicial según se mencionó en el Capítulo 1 depende de la naturaleza del problema. En el siguiente apartado exponemos la generación de la configuración inicial utilizada.

Configuración inicial:

La configuración inicial que necesita nuestra heurística para solucionar el problema MaxA-p-Pv1(P) se traduce en un punto inicial p_0 interior a P, que se considerará como primera solución

a analizar y en su caso a iterar. La elección de este primer punto p_0 a estudiar se realiza de forma aleatoria, generando mediante la función `rand` sus dos cooordenadas. La única condición exigida al punto p_0 es que sea un punto interior al polígono P.

En forma de pseudocódigo la función que genera la configuración inicial tiene la siguiente forma:

Función Generar-Configuracion Inicial

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Un punto $p_0 = (x_0, y_0)$ interior a P.

```
[01]  do
[02]    {x0 ←rand()*1.0/RAND_MAX;
[03]     y0 ←rand()*1.0/RAND_MAX;
[04]    }while (p0 = (x0, y0) exterior a P);
[09]    return p0;
```

Los elementos más importantes en la estrategia heurística de SA son la elección de la temperatura inicial T_0, la reducción de esta temperatura y el número de iteraciones del método heurístico para cada temperatura T_i. Todos estos elementos los exponemos en la siguiente subsección.

7.2.2 Estrategias de templado

La optimalidad del *simulated annealing* depende en gran medida de las condiciones de temperatura elegidas para cada problema. Esto es, la temperatura inicial T_0 y la disminución de dicha temperatura en cada iteración.

Temperatura inicial:

Podríamos decir, en general, que una de las propiedades que debe verificar todo método heurístico de búsqueda es la de no ser dependiente de la solución inicial elegida. Así, sería recomendable que *simulated annealing* parta de una temperatura inicial alta, con lo cual irá recorriendo temperaturas alejadas de la óptima. Sin embargo, no parece conveniente considerar para T_0 valores fijos independientes del problema. Según se muestra en [31], si consideramos que pudiera ser aceptable con un tanto por uno ϕ de probabilidad una solución que sea un μ por uno peor que la inicial S_0, tendríamos

$$\phi = e^{\frac{-\delta}{T_0}} = e^{\left(\frac{-\mu}{T_0}\right)C(S_0)} \Longrightarrow T_0 = \frac{\mu}{-\ln(\phi)}C(S_0) \tag{7.2.5}$$

donde $C(S_0)$ representa el coste de la solución inicial. En este sentido, la bibliografía aconseja realizar diferentes análisis sobre la temperatura inicial a elegir ya que ésta puede depender en gran medida del problema a solucionar. Siguiendo esta idea, se ha realizado un estudio comparativo teniendo en cuenta tres tipos diferentes de temperatura inicial T_0:

- Una temperatura inicial según se indica en 7.2.5:

$$T_0 = \frac{\mu}{-\ln(\phi)} C(S_0)$$

  donde S_0 representa la configuración inicial generada por la función `Generar-Configuración Inicial`. En nuestro estudio comparativo consideraremos $T_0 = C(S_0)$.

- Una temperatura inicial dependiente del número de vértices del polígono P: $T_0 = f(n)$, por ejemplo $f(n) = n$, como será otro de nuestros casos de comparación.

- Una temperatura inicial constante: $T_0 = 100.0$.

En la Sección 7.6 vemos como los resultados obtenidos para el algoritmo con funciones dependientes de n no mejoran los resultados obtenidos con $T_0 = 100.0$, (es decir no mejora el área iluminada), pero se incrementa, en algunos casos, el tiempo de respuesta del algoritmo.

Disminución de la temperatura en cada iteración:

A pesar de la numerosas investigaciones realizadas, no existe ningún mecanismo óptimo que garantice la convergencia de SA de forma rápida y eficiente. En general un buen programa de enfriamiento es aquel en el cual el equilibrio es alcanzado o aproximado en cada temperatura, a pesar de que existen buenos motivos para creer que el tiempo requerido es exponencial en la inversa de la temperatura [94]. En este sentido, se ha realizado un análisis sobre diferentes tipos de decrecimientos de temperatura que aparecen en la bibliografía.

Existen numerosos estudios que demuestran que la mayor desventaja de SA es su lenta convergencia [47]. Así, Geman y Geman [48] prueban que esta técnica heurística converge usando una función inversamente proporcional a una función logarítmica del tiempo, (que denotaremos con convergencia clásica CSA), es decir:

$$T(k) = \frac{T_0}{\ln(1+k)} \quad k = 1, 2, ... \tag{7.2.6}$$

La desventaja de 7.2.6 es la lentitud de la convergencia de $T(k)$ a 0 y por ello no la tendremos en cuenta en nuestro análisis, ya que los tiempos de respuesta con este decrecimiento de temperatura son demasiado elevados. Posteriormente Szu y Hartley [96] propusieron una convergencia rápida, (FSA), usando la función de tiempo 7.2.7, que fue mejorada por Ingber [61], ($VFSA$), proponiendo la función 7.2.8.

$$T(k) = \frac{T_0}{1+k} \quad k = 1, 2, ... \tag{7.2.7}$$

$$T(k) = \frac{T_0}{e^k} \quad k = 1, 2, ... \tag{7.2.8}$$

Finalmente Yao propone una nueva función de decrecimiento, (NSA), mediante la función:

$$T(k) = \frac{T_0}{\exp(e^k)} \quad k = 1, 2, ... \tag{7.2.9}$$

En la Sección 7.6 se muestran los resultados y los tiempos utilizados para diferentes polígonos generados aleatoriamente con dos de los tipos de decrementos propuestos: FSA *y* $VFSA$*).* El tipo de decremento NSA se ha descartado por el motivo contrario que se descarto CSA: la rapidez de NSA evita la búsqueda exhaustiva del punto de máxima iluminación.

Todos los resultados obtenidos se han comparado con un tipo de decremento recomendado por Dowsland para atacar problemas en la práctica [35] , que consiste en considerar una velocidad geométrica de decrecimiento, es decir:

$$T(k) = \alpha T(k-1) \tag{7.2.10}$$

donde α representa un factor de enfriamiento $0 \leq \alpha \leq 1$.

Número de iteraciones de cada temperatura $N(T)$:

El valor de $N(T)$ debe ser lo suficientemente grande como para que el sistema llegue a alcanzar su estado estacionario para esa temperatura T. En nuestro algoritmo el número de iteraciones para cada temperatura es inversamente proporcional a T. Con esto se consigue que la búsqueda se optimice para temperaturas bajas, que será cuando las soluciones se encuentren más cercanas al óptimo. Así, la función de iteración para cada temperatura T, elegida será:

$$N(T) = 1/T \tag{7.2.11}$$

Criterio de parada:

En los experimentos recogidos en la bibliografía con *simulated annealig* para resolver problemas de optimización, se presentan diferentes criterios de parada respecto al valor de la temperatura final T_f. En teoría, la temperatura correspondiente al "sistema frío" debería ser $T = 0$. Sin embargo, bastante antes de llegar a ese valor es prácticamenten nula la probabilidad $e^{-\delta/T}$ de que se acepte un movimiento hacia una solución peor. Por lo tanto, normalmente se puede finalizar con valores $T_f > 0$, sin pérdida de calidad en la solución. En este sentido, la condición de parada elegida en nuestro algoritmo consiste en finalizar la búsqueda cuando la temperatura sea menor que 0.005, es decir, $T_f \leq 0.005$. Evidentemente para temperaturas de parada menores la solución estará más cercana al óptimo, pero el tiempo de respuesta del algoritmo aumentará.

Otros criterios utilizados de sistema frío consisten en detener la búsqueda cuando para una temperatura T no se haya conseguido obtener una solución que mejore la mejor solución hallada hasta ese instante. Esta condición puede obtener mejoras en el tiempo respecto a nuestra elección, pero puede desechar soluciones que mejoren la solución actual y que se encuentren tras una iteración de SA en la que no mejora la función objetivo.

Presentamos en la Figura 7.2 un ejemplo de ejecución de nuestro algoritmo SA para un polígono P con 25 vértices, ejecutado con temperatura inicial constante $T_0 = 100$, con un decremento de temperatura en cada iteración $T(k) = \frac{T_0}{1+k}$ $k = 1, 2, ...,$ y una temperatura final de parada $T_f = 0.005$. En (a) se muestra el polígono P y en (b) el punto de máxima iluminación generado por el algoritmo SA y su polígono de visibilidad, dando el área y el porcentaje iluminado o vigilado. En el apartado (c) mostramos la nube de puntos analizados por el algoritmo y como se puede observar la disminución de la temperatura produce que la densidad de puntos sea mayor

entorno al óptimo, a medida que disminuye dicha temperatura. Finalmente presentamos en (d) todos los elementos conjuntamente.

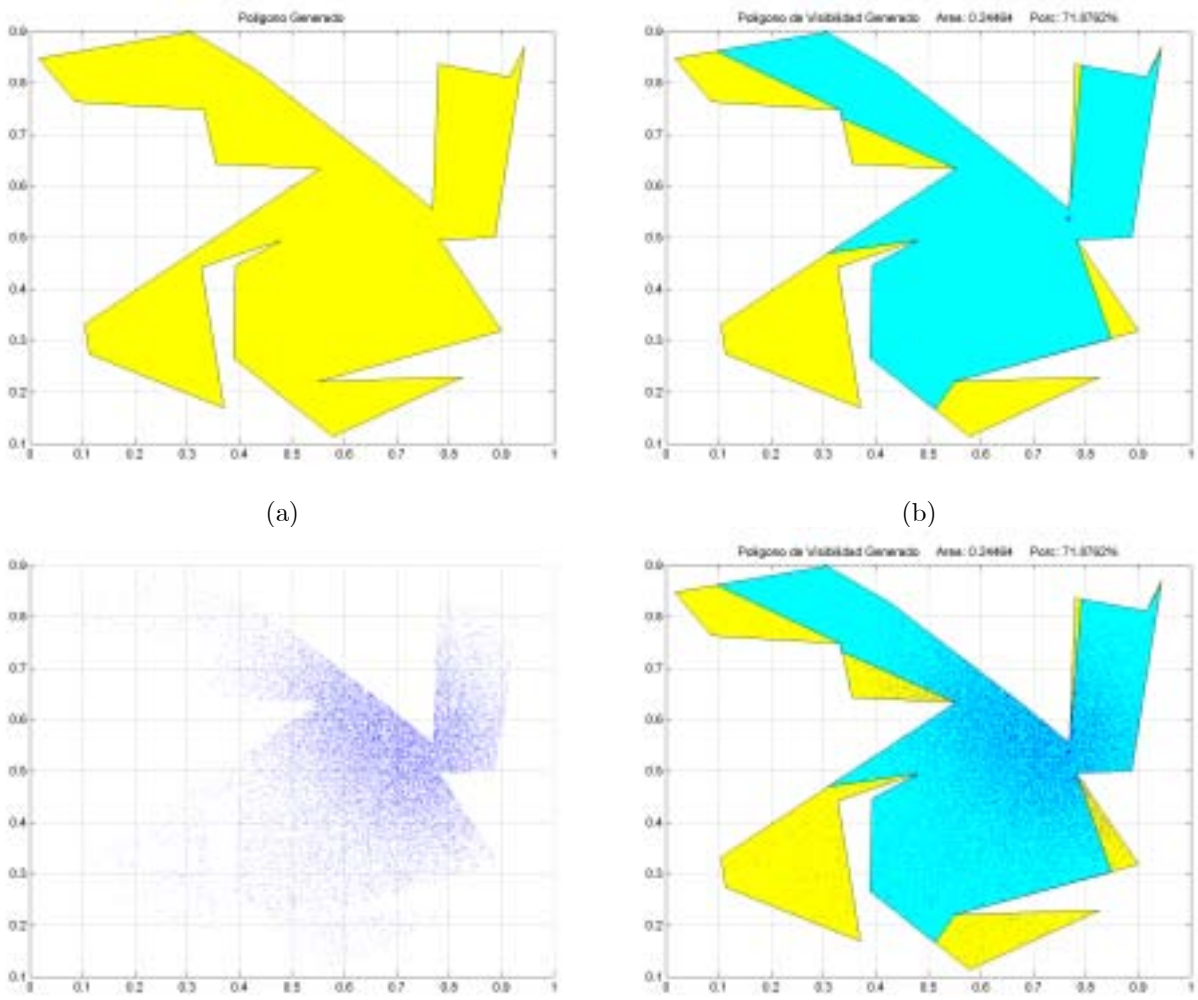


Figura 7.2: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con SA $T_0 = 100$ $T(k) = \frac{T_0}{1+k}$ $T_f = 0.005$ $N(T) = \frac{1}{T}$

```
ALGORITMO SIMULATED ANNEALING SA T0=100:
 * Polígono de visibilidad calculado
 - Punto de área máxima........ (0.767686,0.536000)
 - Área máxima................. 0.24464  71.8762%
```

Pasamos a continuación a describir los elementos utilizados para la construcción de un algorítmo que solucione también el problema MaxA-p-Pv1(P), pero utilizando otra técnica heurística como son los *algoritmos genéticos*.

7.3 El problema MaxA-p-Pv1(P) con algoritmos genéticos-GA

Holland [60] y Golberg [55] desarrollaron los fundamentos teóricos de la teoría de los *algorítmos genéticos*, usando la teoría de esquemas. Una breve descripción a esta teoría se encuentra expuesta en la Sección 1.5.2. La idea fundamental de un *algoritmo genético* es que la población inicial evoluciona sucesivamente hacia mejores regiones del espacio de búsqueda mediante dos procesos fundamentalmente:

- Selección de los individuos más adaptados en la población, (a mayor grado de adaptación mayor probabilidad de dejar descendencia).
- Modificación por recombinación y/o mutación de los individuos seleccionados.

La estructura de un algoritmo evolutivo básico es la siguiente:

```
[01]  begin
[02]    {t ← 0;
[03]     Inicializar(A(t));
[04]     Evaluar(A(t));
[05]     while(no parada)
[06]       {t← t + 1;
[07]        Seleccionar A(t) a partir de A(t-1);
[08]        Recombinar y/o mutar A(t);
[09]        Evaluar A(t);
[10]       }
[11]    }
[12]  end
```

Así según Michalewicz [74] y según se menciona en [87], un algoritmo genético ha de tener las siguientes cinco componentes:

1. Una representación genética de las soluciones posibles del problema (codificación).
2. Una manera de crear una población inicial de soluciones posibles (genomas y población).
3. Una función que evalúe a los individuos y haga el efecto de la selección natural, clasificando las soluciones atendiendo a su "fortaleza" (función objetivo).
4. Operadores genéticos que permitan alterar la composición de las soluciones, (selección cruce y mutación).
5. Valores de varios parámetros que utiliza el algoritmo genético.

A continuación describimos estas componentes diseñadas para el problema MaxA-p-Pv1(P) y que han sido las utilizadas en la implementación del algoritmo correspondiente cuyo código se encuentra en el Apéndice B.

Codificación

Dada una entrada del problema, un algoritmo genético debe poder alcanzar una solución del problema para dicha entrada. Las posibles soluciones del problema se deben poder representarse de alguna manera en el algoritmo genético mediante un conjunto de parámetros, a menudo llamados "individuos" o "genomas". En términos genéticos, este conjunto de parámetros constituye el "genotipo" y la codificación para una determinada solución constituye el "fenotipo".

En el algoritmo genético desarrollado para el problema MaxA-p-Pv1(P) se ha considerado que un individuo es un gen, que viene representado por las coordenadas de un punto, (individuo), interior al polígono P.

$$G = ((x, y))$$

Esta codificación de individuo o genoma será ampliada en el Capítulo 8 donde se estudia el problema MaxA-p-Pvk(P, k) y donde cada individuo de la población estará codificado por un genoma con k genes

$$G = ((x_1, y_1), (x_2, y_2), ..., (x_k, y_k)) \tag{7.3.12}$$

siendo k el cardinal del conjunto de puntos cuya visibilidad se quiere maximizar.

Genoma y población

La población para una determinada generación estará formada por un conjunto de genomas. El número total de genomas que hemos de elegir para cada población debe ser tal que tengamos suficiente diversidad para poder garantizar la eficiencia del algoritmo y que éste produzca buenos hijos durante el proceso de evolución.

El tamaño de la población deberá ser lo suficientemente grande como para garantizar la diversidad pero no tanto que perjudique la eficiencia del algoritmo. Según la bibliografía [87] un cardinal adecuado de población es el doble de la longitud del genoma, sin embargo, en nuestro caso se ha tomado como tamaño de población el número de vértices del polígono, relacionando de esta manera la entrada del problema con los elementos de la heurística.

Así, la población para la generación t de nuestro algoritmo vendrá representado por:

$$A(t) = \{G_1^t, G_2^t, ..., G_n^t\} \tag{7.3.13}$$

donde cada G_i^t representa a un individuo o genoma perteneciente a la población $A(t)$ y n es el número de vértices del polígono P.

Función objetivo

Debemos definir en este paso la *función objetivo.* Esta función debe ayudarnos a realizar la mejor selección posible de individuos que deberán reproducirse, de manera que asignará valores más altos a las soluciones cuanto más se acerquen a la solución óptima. Así, a cada individuo o genoma $G = ((x_i, y_i))$ se le asignará un valor igual al área de la zona iluminada por el gen (x_i, y_i). Si $V(P, (x_i, y_i))$ es el polígono de visibilidad de (x_i, y_i) dentro del polígono P, éste se

podrá expresar como $V(P,(x_i,y_i)) = \bigcup_{j=1}^{q} T_j$, siendo T_j $j = 1,..,q$ el conjunto de triángulos en los que se puede descomponer $V(P,(x_i,y_i))$ al ser un polígono estrellado. Por tanto la función objetivo de nuestro *algoritmo genético* se define

$$f(G) = \sum_{j=1}^{q} Area(T_j) \tag{7.3.14}$$

Para el cálculo de los *polígonos de visibilidad* se ha implementado un algoritmo lineal [72], cuya implementación se encuentra en el Apéndice B. Consideraremos como evaluación de la población $A(t)$, es decir, como *fitness de la población* al máximo valor de $f(G_i^t)$ $\forall i = 1,...,n$. Denotaremos el *fitness de una población* $A(t)$ en la generación t mediante

$$F(A(t)) = \max_{i=1,...,n} f(G_i^t) \tag{7.3.15}$$

Pasamos a continuación a explicar los mecanismos de selección cruce y mutación utilizados en nuestro algoritmo y el *fitness* de población considerado.

Selección, cruce y mutación

Los operadores de *selección*, *cruce* y *mutación* son elementos importantes en todo algoritmo genético, pues determinarán el éxito de los mecanismos de reproducción del algoritmo. El operador de selección debe elegir aleatoriamente de la población unos padres, de forma que los individuos para los que la función objetivo sea mayor tengan mayor probabilidad de ser elegidos. El método de selección utilizado en nuestro algoritmo es el de *selección proporcional* implementado mediante el método de la *ruleta,* (ver Capítulo 1), de tal forma que los individuos con mayor función objetivo tengan mayor probabilidad de ser elegidos. Los pasos seguidos en la implementación de éste método son los siguientes:

(a) Determinar la suma S de las adaptaciones de toda la población.

(b) Relacionar una a uno todos los individuos con segmentos contiguos de la recta real $[0,S)$, tal que cada segmento individual sea igual en su tamaño a su grado de adaptación.

(c) Generar un número aleatorio en $[0,S)$.

(d) Seleccionar el individuo cuyo segmento cubre el número aleatorio.

(e) Repetir el proceso hasta obtener el número deseado de muestras.

Así, si $\{G_1^t,...,G_n^t\}$ es la población en la generación t y $S = \sum_{i=1}^{n} f(G_i^t)$, elegimos un número aleatorio r uniformemente distribuido entre 0 y S. Si $r \leq f(G_1^t)$ elegimos el individuo G_1^t, en otro caso elegimos el individuo G_i^t tal que $\sum_{j=1}^{i-1} f(G_j^t) < r \leq \sum_{j=1}^{i} f(G_j^t)$. Este mecanismo se repite dos veces para obtener los dos padres de nuestra generación t, G_i^t y G_j^t.

Una vez seleccionados los padres para la generación t, $(G_i^t,\ G_j^t)$ pasamos a modificarlos mediante el operador de cruce como se mencionó en la Sección 1.5.2. Para ello generamos aleatoriamente un número r $(0 \leq r < 1)$. Si r es mayor que la probabilidad de cruce p_c estos individuos pasarán sin ser modificados a la generación siguiente, en caso contrario serán sometidos al proceso de cruce. El mecanismo de cruce utilizado es el *cruce en un solo punto* descrito en la fórmula 1.5.19. Elegimos al azar un número aleatorio $1 \leq r \leq k$ e intercambiamos los genes de los padres a partir del gen que ocupa el lugar r. Evidentemente como el caso que nos ocupa en esta sección es $k = 1$, este operador de cruce producirá , en caso de aplicarse, un intercambio del gen de un padre por el gen del otro padre y viceversa. Por ello, consideramos para $k = 1$ que no tenemos operador de cruce, modificando los genes de una generación a la siguiente mediante el operador de mutación.

Sin embargo si $k > 1$, caso que trataremos en el capítulo siguiente para solucionar el problema MaxA-p-Pvk(P, k), si tendrá sentido este operador de cruce, ya que en este caso cada individuo padre vendrá representado por k genes, con $k > 1$ y el operador de cruce en un punto producirá una mezcla de los genes de cada padre a partir del gen r. Así, si los genomas de los padres seleccionados vienen descritos por:

$$\left.\begin{array}{l} G_i^t = \left((x_1^{(i)}, y_1^{(i)}), ..., (x_r^{(i)}, y_r^{(i)}), (x_{r+1}^{(i)}, y_{r+1}^{(i)}), ..., (x_k^{(i)}, y_k^{(i)})\right) \\ G_j^t = \left((x_1^{(j)}, y_1^{(j)}), ..., (x_r^{(j)}, y_r^{(j)}), (x_{r+1}^{(j)}, y_{r+1}^{(j)}), ..., (x_k^{(j)}, y_k^{(j)})\right) \end{array}\right\} \tag{7.3.16}$$

tendremos tras el procedimiento de cruce los genomas

$$\left.\begin{array}{l} G_i^{*t} = \left((x_1^{(i)}, y_1^{(i)}), ..., (x_r^{(i)}, y_r^{(i)}), (x_{r+1}^{(j)}, y_{r+1}^{(j)}), ..., (x_k^{(j)}, y_k^{(j)})\right) \\ G_j^{*t} = \left((x_1^{(j)}, y_1^{(j)}), ..., (x_r^{(j)}, y_r^{(j)}), (x_{r+1}^{(i)}, y_{r+1}^{(i)}), ..., (x_k^{(i)}, y_k^{(i)})\right) \end{array}\right\} \tag{7.3.17}$$

Obtenidos los padres, (que son también hijos en el caso $k = 1$), para la generación $t + 1$ les aplicaremos el operador de mutación. Para cada individuo $G = ((x_i, y_i))$ generamos un número aleatorio r $(0 < r \leq 1)$. Si r es mayor que la probabilidad de mutación p_m no será sometido a mutación, en caso contrario generaremos dos números aleatorio a y b que siguen la distribución $N(0, 1)$ y sustituiremos el gen (x_i, y_i) por $(x_i + a/100, y_i + b/100)$. Debe recordarse en este punto que los polígonos que produce el generador aleatorio de polígonos RPG implementado y descrito en el Apéndice A, se generan en el cuadrado unidad, es decir, en el cuadrado de vértice$\{(0, 0), (0, 1), (1, 1), (1, 0)\}$, por lo que tiene sentido sumar a cada coordenada de un gen un número aleatorio que siga la distribución $N(0, 1)$ dividido por 100.

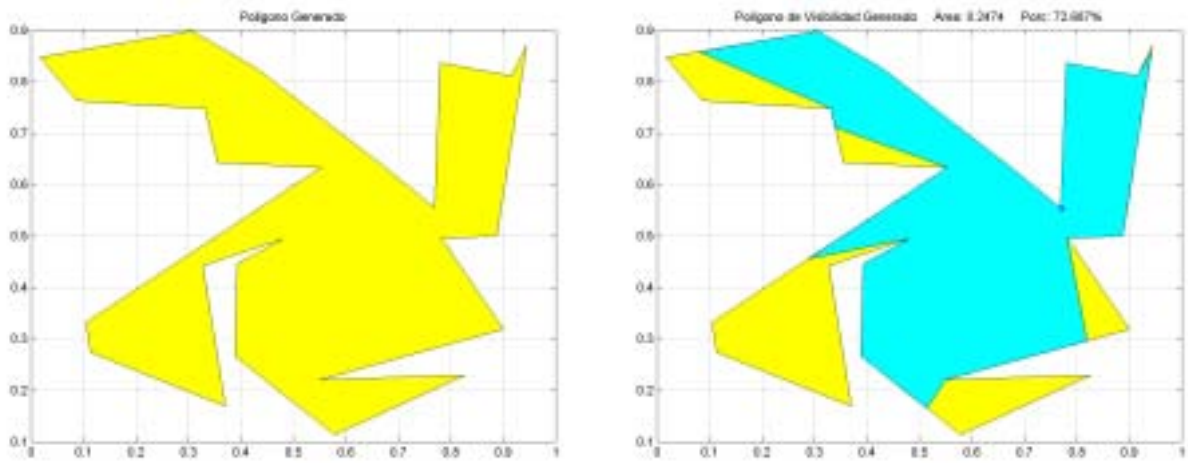
Condición de parada

Si en un número suficientemente grande de generaciones la *función objetivo* no se ha modificado, podemos suponer que estamos cerca del óptimo. Por ello, se ha considerado en nuestro algoritmo genético como condición de parada que el *fitness* de la población $F(A(t))$ no varíe durante un número h de generaciones. En nuestro caso se ha considerado $h = 20$. Es decir AG para cuando

$$\exists t \in \mathbb{N} \ / \ F(A(t + h)) \leq F(A(t)) \quad \forall h = 1, ..., 20$$

y por tanto t representa la generación de parada.

Presentamos en la Figura 7.3 un polígono de 25 vértices generado aleatoriamente por RPG y la solución aportada por el algoritmo genético GA, indicando el punto de máxima iluminación encontrado, el área iluminada y el porcentaje de ésta respecto al área del polígono total. Comparando con el resultado obtenido con SA se observa con en este caso se mejora el porcentaje de área iluminada. Mostramos en la Figura 8.16 la curva de crecimiento de la función objetivo según avanza el algoritmo en cada generación.



(a) (b)

Figura 7.3: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con GA $p_c = 0.8$ $p_m = 0.05$

```
ALGORITMO GENÉTICO GA pc=0.8 pm=0.05:
 * Polígono de visibilidad calculado
 - Punto de área máxima........ (0.770995,0.553071)
 - Área máxima................. 0.24740  72.6870%
```

Debe hacerse notar que la heurística AG para el caso $k = 1$ la podemos considerar un caso particular de un algoritmo *random search* que estudiamos en la siguiente sección, ya que se omite el operador de cruce al utilizar cruce en un punto. En este sentido, se podrían considerar otros operadores de cruce que eliminaran esta dificultad, como podrían ser cruce en un punto con cadenas binarias. Sin embargo se ha preferido utilizar esta particularidad para que exista homogeneidad entre los operadores utilizados para resolver el problema MaxA-p-Pv1(P), ($k = 1$), y el problema MaxA-p-Pvk(P, k) con $k > 1$ que estudiaremos en el capítulo siguiente.

Asimismo, es importante detacar que todas las experimentaciones para esta heurística AG han sido realizadas tomando como probabilidad de cruce $p_c = 0.8$ y como probabilidad de mutación $p_m = 0.5$. Como se mencionará en la sección de conclusiones uno de los trabajos pendientes es intentar ajustar más, mediante la experimentación, estas dos probabilidades.

7.4 Aproximación a MaxA-p-Pv1(P) con random search-RS

Proponemos en esta sección lo que podríamos considerar el algoritmo más sencillo, (pero no por ello menos eficiente), que soluciona de forma heurística el problema MaxA-p-Pv1(P). La idea del algoritmo es realizar una búsqueda aleatoria, (*random search*), sobre los puntos interiores al polígono P. Para ello se generará una nube $N = \{p_1, ..., p_m\}$ de m puntos interiores al polígono P y se realizará una búsqueda secuencial sobre N del punto $p_i \in N$ tal que

$$Área(V(P, p_i)) \geq Área(V(P, p_j)) \quad \forall j \neq i \quad i, j \in \{1, ..., m\} \tag{7.4.18}$$

siendo $V(p_i)$ el polígono de visibilidad del punto p_i interior a P.

Recordemos el enunciado del problema MaxA-p-Pv1(P):

búsqueda del punto de máxima iluminación interior a un polígono

MaxA-p-Pv1(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿En qué punto interior a P debo colocar una luz para que el área iluminada por dicha luz sea máxima?

En forma de pseudocódigo la función que genera la nube es la siguiente:

Función Generar-Nube

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$ y un entero m.
SALIDA: Un conjunto $N = \{p_1, ..., p_m\}$ de m puntos interiores a P.

```
[01]  i ←1;
[02]  N = {};
[03]  do
[04]    {do
[05]      {x ←rand()*1.0/RAND_MAX;
[06]       y ←rand()*1.0/RAND_MAX;
[07]      }while (p = (x, y) exterior a P);
[08]     p_i ← (x, y);
[09]     N ← N ∪ p_i;
[10]     i ← i + 1;
[11]    }while (i <= m);
[12]    return N;
```

Evidentemente el tamaño de la nube de puntos N debe estar relacionado con el número de vértices del polígono, de tal forma que para polígonos con mayor número de vértices el valor de m sea mayor. Sin embargo, se debe considerar un tamaño "razonable" para la nube pues en otro caso el tiempo de respuesta de la heurística sería demasiado elevado, ya que nuestro algoritmo debe calcular la zona iluminada y por tanto el polígono de visibilidad $V(P, p_i)$ para todo punto

p_i perteneciente a N. Hemos considerado un tamaño de nube $m = 50\ n$. Es importante destacar que si se utiliza un algoritmo lineal para construir cada polígono de visibilidad $V(P, p_i)$ [72], (que ha sido el utilizado en la implementaciones en C++ del Apéndice B), la complejidad de nuestro algoritmo resulta ser $O(mn)$.

En forma de pseudocódigo el algoritmo *RS* propuesto es el siguiente:

Algoritmo RS-MaxA-p-Pv1(P)

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Una *luz-punto* de máxima iluminación en P.

```
[01]  área ← 0;
[02]  N ← Generar_Nube(P, 50n);
[03]  for (p_i ∈ N)
[04]    {Calcular el polígono de visibilidad de p_i en P, V(P, p_i);
[05]     if (área < Área(V(P, p_i))
[06]       área ← Área(V(P, p_i));
[07]       p ← p_i;
[08]    }
[09]    return p;
```

Presentamos en la Figura 7.4 un ejemplo de ejecución de este algoritmo sobre el polígono P de 25 vértices, (generado aleatoriamente con nuestro generador *RPG*), de la Figura 7.2 y la Figura 7.3, junto con la nube analizada.

Nótese la diferencia entre la Figura 7.2(d) y la Figura 7.4, donde se muestran las nubes de puntos analizados que producen dichos algoritmos. En la primera, (producida por *SA*), se aprecia como la nube tiene mayor densidad entorno al óptimo, mientras en la segunda los puntos están uniformemente distribuidos.

Es importante apreciar en este ejemplo que aunque la nube generada para *RS* tiene sólamente 1250 puntos frente a los 20381 que se muestra en la Figura 7.2(d) para el algoritmo *SA*, la diferencia del área iluminada es de orden de $2 \cdot 10^{-2}$. Ello quiere decir que aunque *SA* produce mejor aproximación en la solución pero consume mayor tiempo de ejecución frente a *RS,* que produce una solución menos exacta. Por ello, si se desea obtener una solución muy exacta del problema, (jugando el tiempo un papel secundario), será adecuado la utilización del algoritmo *SA*, mientras que si el tiempo es un factor importante, pero la solución puede ser menos exacta será más adecuado el algoritmo *RS*. Si en *RS* se generase una nube de puntos de igual tamaño a la estudiada por *SA,* los resultados serán parecidos ya que aunque *SA* busca en torno al óptimo, también puede visitar puntos no deseados con una cierta probabilidad. Estas y otras conclusiones se analizarán en la Sección 7.6, en la que se realizará un estudio comparativo de todos los algoritmos expuestos.

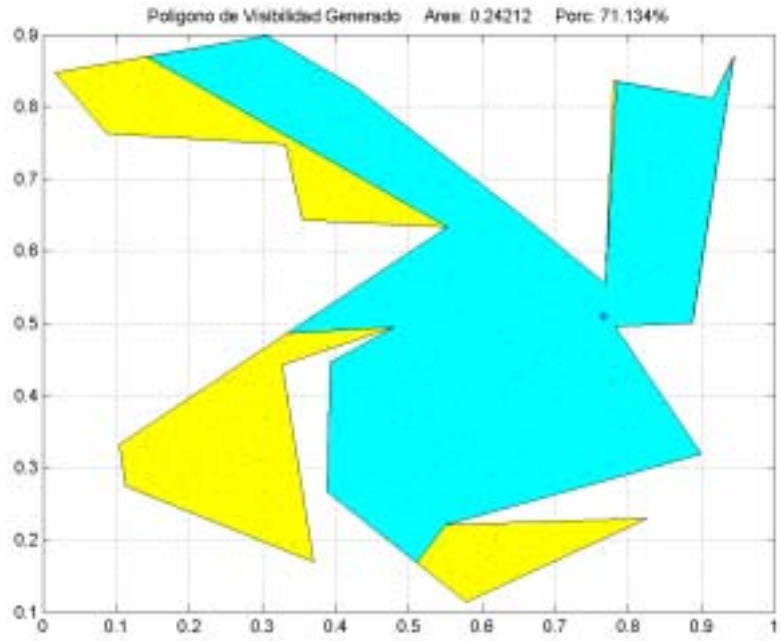


Figura 7.4: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con RS $m = 50 \cdot 25 = 1250$

```
ALGORITMO RANDOM SEARCH RS m=1250
 * Polígono de visibilidad calculado
 - Punto de área máxima........ (0.766808,0.509995)
 - Área máxima................. 0.2442116  71.1340%
```

7.5 Un nuevo enfoque. El gradiente-GRAD

Presentamos en esta sección la última heurística diseñada para solucionar el problema MaxA-p-Pv1(P) y que no está basada en ninguna técnica heurística general como pueden ser *simulated annealing* ó los *algoritmos genéticos*. La idea fundamental consiste en considerar que todo punto p interior a un polígono P de n vértices, que no sea un máximo local, se puede desplazar según un "*gradiente positivo*" de área iluminada.

Así, dado un punto p interior al polígono P consideremos un conjunto $\{p_1, ..., p_v\}$ de v puntos vecinos de p, que disten β de él y de tal forma que los vectores que unen dichos puntos con p formen secuencialmente y en sentido positivo un ángulo $2\pi/v$. Es decir, podemos estudiar un conjunto v de puntos alrededor de p que estén a una distancia β de él y tomar aquel punto que más aumente el área iluminada, moviendo p a dicho punto para analizar la siguiente iteración. En la Figura 7.5 podemos ver el conjunto de puntos vecinos a analizar tomando $v = 8$.

Diseñamos a continuación una función que determina el mejor vecino de entre los v posibles, al que debemos mover el punto p para que el área iluminada aumente el máximo, es decir

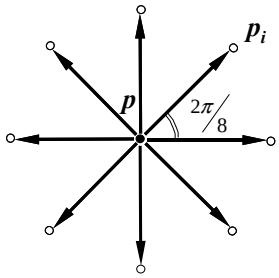


Figura 7.5: Vecinos a analizar para un punto p interior a P $v = 8$

movemos p según un gradiente positivo de área iluminada:

Función Generar-Candidato

ENTRADA: Un polígono P de n vértices, un punto $p = (x, y)$ interior o en el borde de él, un entero v de vecinos a analizar y un factor de separación β.
SALIDA: Un punto q tal que $\acute{A}rea(V(P, q)) \geq \acute{A}rea(V(P, p))$.

```
[01]  a ← Área(V(p));
[02]  α ← 0.0;
[03]  i ← 1.0;
[04]  q ← p;
[05]  while (i <=v)
[06]    {x* ← x + (β * cos(α));
[07]     y* ← y + (β * sin(α));
[08]     cand ← (x*, y*);
[09]     if(Área(V(P, cand)) ≥ Área(P, q))
[10]       q ←cand;
[11]     ++i;
[12]     α ← α+2π/v;
[13]    }
[14]    return q;
```

Evidentemente se conseguirá un mejor ajuste cuanto mayor sea el número v de vecinos a analizar y menor sea el factor de separación de dichos vecinos, respecto de p, es decir, cuanto menor sea β. En la implementación realizada en el Apéndice B para esta heurística se ha considerado $v = 16$ y $\beta = 0.001$.

Sin embargo, nos preguntamos ahora si será suficiente colocar aleatoriamente un punto p en el interior del polígono y hacer que este punto se mueva según un gradiente positivo hasta que se sitúe en un punto tal que todos los vecinos a él tengan menor área. Evidentemente, esto producirá frecuentemente que p converja a un máximo local. Para evitar esta situación se ha

considerado que tenemos inicialmente una luz en todos los vértices del polígono P, haciendo que cada una de ellas converja a un máximo local. A continuación se elige como solución del problema MaxA-p-Pv1(P) el mejor de estos máximos locales.

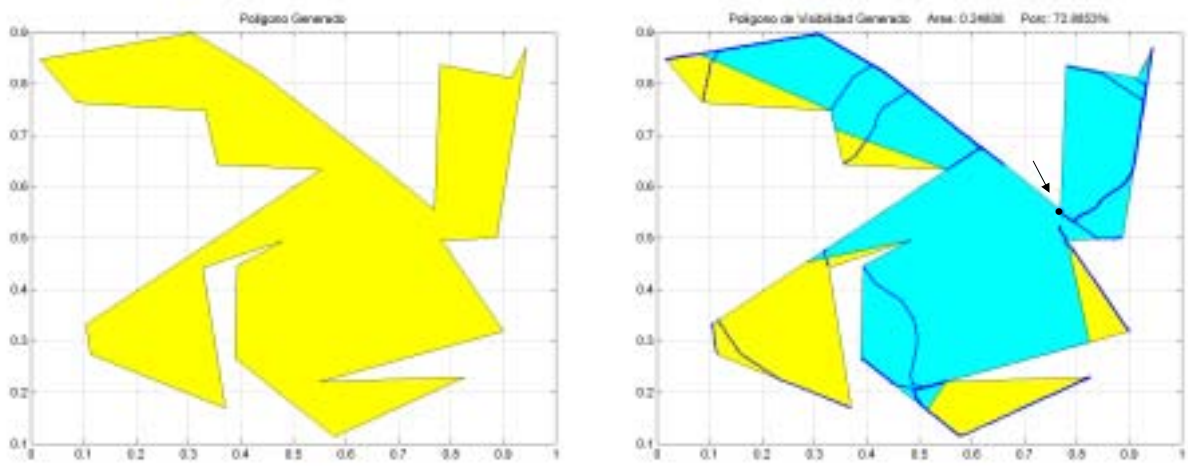
La pregunta que podemos hacernos ahora es la siguiente: ¿estamos seguros que la solución a nuestros problema se encuentra entre el conjunto de máximos locales generados? Las experimentaciones llevadas a cabo indican que la respuesta a esta pregunta es afirmativa. En forma de pseudocódigo el algoritmo *gradiente-GRAD* se puede representar de la siguiente manera:

Algoritmo GRAD-MaxA-p-Pv1(P)

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$..
SALIDA: Una *luz-punto* de máxima iluminación en P.

```
[01]  cambios ← 1;
[02]  max ← 1;
[03]  for (v_i)                          /*Hacemos una copia de los vértices de P*/
[04]    q_i ← v_i;
[05]  while(cambios ≠ 0)
[06]    {cambios ← 0;
[07]     for (q_i)
[08]       {c_i ← Generar_Candidato q_i;
[09]        ar ← Área(V(P, c_i));        /*Movemos cada q_i según el gradiente positivo*/
[10]        if(ar > Área(V(P, q_i))
[11]          {q_i ← c_i;
[12]           cambios ←cambios+1;
[13]           if(ar > Área(V(P, q_max))
[14]             max ← i;
[15]          }
[16]       }
[17]    }                                /*Devolvemos el punto máximo*/
[18]    return q_max;
```

Para finalizar mostramos en la Figura 7.6 la solución aportada *GRAD* al ejemplo que hemos mostrado al finalizar todos las heurísticas. La Figura 7.6(b) muestra la solución aportada, (está indicada con una flecha y un punto más oscuro), y los caminos que recorren los vértices cuando se mueven en el sentido positivo del gradiente de área. Obsérvese que cada conjunto de vértices tienden a un óptimo local y que uno de ellos es la solución que aporta el algoritmo. En este ejemplo la solución aportada por *GRAD* mejora las heurísticas anteriores. Este hecho será importante en los resultados presentados en la Sección 7.6, donde estudiaremos también otros parámetros como es el tiempo empleado por cada método aproximado.



(a) (b)

Figura 7.6: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con GRAD $v = 16$ $\beta = 0.001$

```
ALGORITMO GRADIENTE GRAD v=16 β=0.001
 * Polígono de visibilidad calculado
 - Punto de área máxima........ (0.769406,0.554380)
 - Área máxima................. 0.248077 72.88530%
```

7.6 Resultados. Tests comparativos

Según se puede desprender de las secciones anteriores existe una gran cantidad de combinaciones de los parámetros que influyen sobre nuestras heurísticas y en especial sobre *simulated annealing,* (SA). Evidentemente la comparación de los resultados que se puedan obtener no se puede hacer sobre un único polígono P, pues ello no representaría ningún dato comparativo. Con la finalidad de obtener datos que nos permitan determinar conclusiones se ha realizado un estudio comparativo aplicando cada posible combinación de parámetros de la heurística SA, a cuatro conjuntos de 50 polígonos cada uno de ellos. El primero de estos conjuntos estará formado por 50 polígonos con 50 vértices generados aleatoriamente con RPG; el segundo por 50 polígonos de 100 vértices y el tercero y el cuarto por 50 polígonos de 150 y 200 vértices respectivamente. Para cada polígono se ha obtenido el porcentaje de área iluminada por la *luz-punto* de máxima iluminación, (que soluciona por tanto el problema MaxA-p-Pv1(P)), el tiempo empleado para obtener dicha solución y el número de iteraciones empleadas por el algoritmo. Ello nos permitirá concluir cual es la mejor combinación de los parámetros que influyen sobre cada una de la heurísticas expuestas: SA, GA, RS y $GRAD$.

7.6.1 Análisis de parámetros para SA

Según la Sección 7.2 los parámetros que determinan la bondad de la heurística SA los hemos divididos en dos grupos como mostramos de forma resumida en el siguiente esquema:

- **Adaptación del problema**
  - Conjunto S de configuraciones.
  - Función C de coste.
  - Vecindad de cada configuración.
  - Configuración inicial.
- **Estrategias de templado**
  - Temperatura inicial T_0.
  - Disminución de la temperatura en cada iteración.
  - Número de iteraciones en cada temperatura $N(t)$.
  - Criterio de parada.

La mayoría de estos elementos se encuentran ya fijados y descritos en la Sección 7.2, pero existen 2 parámetros que deben ser analizados para encontrar la combinación de su valores que mejor se adapta a nuestro problema. Estos son, la elección de la tempea inicial T_0 y la disminución de la temperatura en cada iteración. En este sentido, según lo expuesto en la Sección 7.2 tenemos nueve posibles combinaciones en la elección de T_0 y el decremento de la temperatura. Estas combinaciones producirán nueve **Casos** que compararemos realizando un estudio heurístico y generando para ello, (utilizando nuestro generador aleatorio de polígonos RPG, descrito en el Apéndice A y cuya implementación se encuentra en el Apéndice B), un conjunto de 50 polígonos de 50, 100, 150 y 200 vértices y comparando el área iluminada, el tiempo empleado y el número de iteraciones realizadas por la heurística. Debe hacerse notar que en todos los casos se ha impuesto una condición para la temperatura final de parada $T_f \leq 0.005$. Si este valor se disminuye la solución se acercará más al óptimo, pero los tiempos de respuesta aumentarán ya que el número de configuraciones evaluadas será mayor. Los resultados obtenidos para cada caso, indicando la temperatura inicial T_0 y el tipo de decremento son los siguientes:

▶ **Caso 1:** $T_0 = C(S_0) \quad T_k = \frac{T_0}{1+k}\ (FSA)$

Presentaremos los resultados obtenidos en este primer caso en la siguiente tabla.

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	50.3236	14.44	4621.82
100	33.4460	22.30	2322.84
150	23.7555	29.70	1638.48
200	17.4808	28.18	1076.30

Tabla 7.1: Resultados Caso 1: $T_0 = C(S_0) \quad T_k = \frac{T_0}{1+k}\ (FSA)$

En esta tabla se expone, como se puede observar, la media del procentaje de área iluminada, la media del tiempo empleado en segundos y la media del número de iteraciones de llamada

del algoritmo. Análogamente presentamos en las Tablas 7.2, 7.3 los resultados obtenidos en los Casos 2 y 3.

► **Caso 2:** $T_0 = C(S_0) \quad T_k = \frac{T_0}{e^k}\ (VFSA)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	28.6057	0.14	37.84
100	14.5109	0.10	18.96
150	9.1404	0.12	13.24
200	6.8468	0.20	12.18

Tabla 7.2: Resultados Caso 1: $T_0 = C(S_0) \ \ T_k = \frac{T_0}{e^k}\ (VFSA)$

► **Caso 3:** $T_0 = C(S_0) \quad T_k = \alpha\ T_{k-1} \quad (\alpha = 0.9)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	40.2871	0.74	270.36
100	20.3790	0.88	129.48
150	13.4328	1.44	98.60
200	11.4124	2.06	99.88

Tabla 7.3: Resultados Caso 1: $T_0 = C(S_0) \ \ T_k = \alpha\ T_{k-1} \quad (\alpha = 0.9)$

Como se puede observar en estos tres primeros casos, la elección de una temperatura inicial T_0 dependiente de la función de costes C sobre la configuración inicial S_0, provoca que el número de iteraciones en la búsqueda sea demasiado pequeño, si el decremento de la temperatura es un decremento rápido, como sucede con $VFSA$. Además, si el número de vértices del polígono es grande el valor de $C(S_0)$, (que corresponde al área iluminada por el punto que determina la configuración inicial), tiende a ser pequeño, con lo cual el valor de la temperatura inicial $T_0 = C(S_0)$ será pequeño y muy cercano a la condición de parada, y por ello la solución obtenida se alejará del óptimo. Todos estos resultados se pueden observar en las Tablas 7.1, 7.2 y 7.3, en las cuales vemos como para estos tres primeros casos la mejor solución aportada corresponde a un decremento de temperatura lento como es FSA, con un número mayor de iteraciones y un tiempo de respuesta superior, es decir, la mejor solución aportada en estos tres primeros casos es la que produce el Caso 1.

En los tres siguientes casos haremos depender la temperatura inicial de la entrada del problema, es decir, del número de vértices n del polígono P. Se ha considerado una temperatura inicial lineal en el número de vértices $T_0 = n$, pudiendo ser motivo de futuras investigaciones el estudio del comportamiento de la heurística para funciones no lineales en la temperatura inicial.

Los resultados obtenidos se muestran en la Tablas 7.4, 7.5 y 7.6 para los tres decrecimientos de temperatura considerados respectivamente.

► **Caso 4:** $T_0 = n \quad T_k = \frac{T_0}{1+k}$ (FSA)

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	51.4632	29.52	10156
100	34.7576	218.08	20381
150	24.7619	509.46	30629
200	19.7551	1333.46	40897

Tabla 7.4: Resultados Caso 1: $T_0 = n \quad T_k = \frac{T_0}{1+k}$ (FSA)

► **Caso 5:** $T_0 = n \quad T_k = \frac{T_0}{e^k}$ $(VFSA)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	26.2933	0.18	82
100	16.1801	1.16	159
150	10.4725	3.38	240
200	7.6786	7.92	318

Tabla 7.5: Resultados Caso 1: $T_0 = n \quad T_k = \frac{T_0}{e^k}$ $(VFSA)$

► **Caso 6:** $T_0 = n \quad T_k = \alpha \; T_{k-1} \quad (\alpha = 0.9)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	37.9920	1.22	523
100	20.8935	8.78	1020
150	15.5067	25.68	1519
200	12.7956	55.42	2018

Tabla 7.6: Resultados Caso 1: $T_0 = n \quad T_k = \alpha \; T_{k-1} \quad (\alpha = 0.9)$

Comparando estos tres últimos casos con los tres primeros para los mismo tipos de decremento de la temperatura, es decir, el Caso1 con el Caso 4, el Caso 2 con el Caso 5 y el Caso 3 con el Caso 6, vemos claramente como si la temperatura inicial depende de n, la solución aportada por el algoritmo mejora en todos los casos aunque el tiempo y el número de iteraciones del algoritmo también aumenta, sobre todo cuando el decrecimiento es más lento como sucede en el Caso 4. Sin embargo, si el decrecimiento de temperatura es FSA, (Casos 1 y 4), las soluciones aportadas no difieren sustancialmente como vemos comparado las Tablas 7.1 y 7.4. Por tanto, podemos concluir que en general si se busca una solución más cercana al óptimo es más adecuado elegir una temperatura inicial dependiente del número de vértices del polígono P, aunque si el decrecimiento de la temperatura es lento la elección de la temperatura inicial no resulta influyente.

Analizados estos seis primeros casos podemos obtener las siguientes conclusiones:

- Mejora en general la solución aportada por SA cuando la temperatura inicial depende del número de vértices de P.
- Si el decrecimiento es lento como FSA, la elección de la temperatura inicial, ($T_0 = C(S_0)$ ó $T_0 = n$), no resulta determinante. Un decrecimiento geométrico $T_k = \alpha\ T_{k-1}\ \ (\alpha = 0.9)$ produce mejores soluciones que un decrecimiento más rápido como $VFSA$.

Analizamos en los tres siguientes casos como se comportan los tres decrecimientos si la temperatura inicial es un valor constante. Como el número de vértices de los polígonos analizados es 50, 100, 150 y 200, se ha elegido un valor constante $T_0 = 100.0$. Los resultados obtenidos se muestran en las tres siguientes tablas.

► **Caso 7:** $T_0 = 100.0 \quad T_k = \frac{T_0}{1+k}\ (FSA)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	51.7463	69.46	20381
100	34.0813	219.12	20381
150	23.5467	458.84	20381
200	19.5442	781.06	20.381

Tabla 7.7: Resultados Caso 1: $T_0 = 100.0\ \ T_k = \frac{T_0}{1+k}\ (FSA)$

► **Caso 8:** $T_0 = 100.0 \quad T_k = \frac{T_0}{e^k}\ (VFSA)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	27.5832	0.46	159
100	13.8940	1.30	159
150	10.7000	2.56	159
200	7.7105	4.26	159

Tabla 7.8: Resultados Caso 1: $T_0 = 100.0\ \ T_k = \frac{T_0}{e^k}\ (VFSA)$

► **Caso 9:** $T_0 = 100.0 \quad T_k = \alpha\ T_{k-1}\ \ (\alpha = 0.9)$

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	38.9342	2.88	1020
100	21.3399	9.90	1020
150	14.6113	19.44	1020
200	11.4176	31.24	1020

Tabla 7.9: Resultados Caso 1: $T_0 = 100.0\ \ T_k = \alpha\ T_{k-1}\ \ (\alpha = 0.9)$

Observando estos tres últimos casos comprobamos sorprendentemente como para una temperatura inicial constante $T_0 = 100.0$ los resultados mejoran cuando el decrecimiento de la temperatura es lento, como se puede observar comparando las Tablas 7.4 y 7.7. Sin embargo, este crecimiento no se puede observar si el decrecimiento de la temperatura es más rápido como sucede en los Casos 8 y 9.

Debemos recordar que hemos tomado una temperatura de parada $T_f \leq 0.005$. Evidentemente, si esta temperatura se disminuye los resultados de las heurística mejoran y en mayor medida aquellos casos que se encuentran más lejos de encontrar el óptimo, es decir para decrecimientos de temperatura más rápidos: $VFSA$ y geométrico. Por tanto, si se desea una solución aceptable y rápida se podrá obtener reduciendo la temperatura de parada y utilizando un decrecimiento rápido como $T_k = \alpha\ T_{k-1}\ \ (\alpha = 0.9)$,(geométrico).

Es importante observar que el conjunto de variantes respecto a los parámetros de SA que se podrían estudiar es casi infinita. Hemos pretendido en esta memoria encontrar referencias generales para dichos parámetros, haciendo notar que un estudio más exhaustivo en futuras investigaciones podrían mejorar los resultados obtenidos.

Así, las conclusiones generales que podemos deducir del estudio realizado para SA respecto al problema MaxA-p-Pv1(P) son las siguientes:

- Decrecimientos lentos de temperatura mejoran la solución, aunque el tiempo de respuesta del algoritmo es mayor.

- En general las mejores soluciones se consiguen tomando una temperatura inicial dependiente del número de vértices n del polígono P, aunque para polígonos con un número de vértices pequeño, ($n \leq 200$), una temperatura inicial constante produce mejores resultados.

- Los decrecimientos rápidos son útiles cuando se desean peores soluciones pero rápidas. En estos casos una temperatura de parada menor a 0.005 mejora las soluciones.

- En general los mejores resultados respecto a porcentaje de área iluminada se obtienen en el Caso 4: $T_0 = n\ T_k = \frac{T_0}{1+k}\ (FSA)$. Este será el caso que utilizaremos para realizar la comparativa con el resto de heurísticas diseñadas.

Presentamos en la Figura 7.7 un diagrama de tallos para mostrar de forma resumida los resultados de los casos anteriores. La coordenada x representa el Caso x, la coordenada y el número de vértices del polígono donde se ha aplicado la heurística SA y la coordenada z el porcentaje medio iluminado. Los puntos representan los elementos de cada una de las tablas.

7.6.2 Comparativa con GA

Exponemos en la siguiente tabla los resultados de aplicar la heurística *algoritmos genéticos-GA* al mismo conjunto de polígonos aleatorios generados con RPG y utilizados para analizar SA. Recuérdese que se exponen el porcentaje medio de área iluminada, el tiempo empleado por el método y el número de iteraciones. Para cada número de vértices se ha aplicado la heurística a 50 polígonos, considerando $p_m = 0.8$, $p_c = 0.05$.

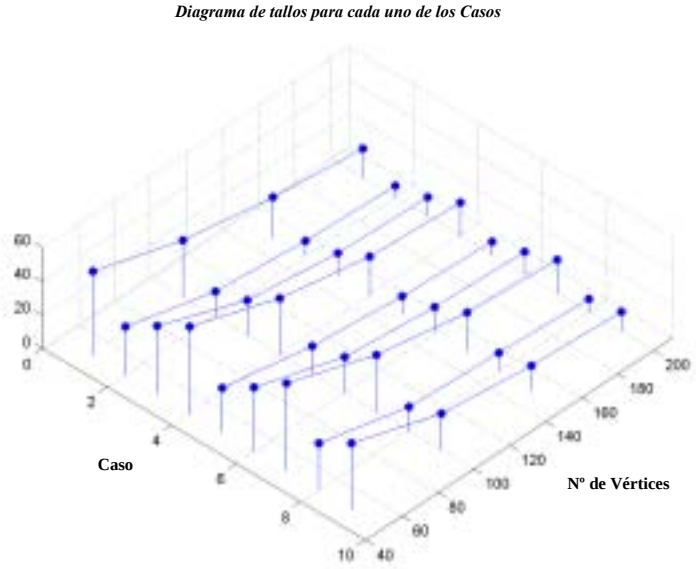


Figura 7.7: Resultados obtenidos para cada Caso en SA

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	50.2310	24.16	5426
100	31.4112	215.94	12304
150	20.0216	604.23	18694
200	17.9814	1903.23	32711

Tabla 7.10: Resultados para la heurística GA $p_c = 0.8$ $p_m = 0.05$

Como podemos apreciar el porcentaje de área iluminada disminuye para todos los vértices respecto a los datos de la Tabla 7.4, (Caso 4 SA). Además el tiempo de respuesta de GA es mayor que el aportado SA, creciendo para mayor número de vértices.

7.6.3 Comparativa con RS

Analizamos en este apartado los resultados obtenidos al aplicar la heurística *random search (RS),* al mismo conjunto de polígonos utilizados para realizar la comparativa de casos con SA. Debe recordarse que se generaron, (utilizando nuestro generador RPG), aleatoriamente cuatros conjuntos de 50 polígonos de 50, 100,150 y 200 vértices respectivamente, sobre los que se aplicó la heurística. En la siguiente tabla se exponen las medias de porcentajes de área iluminada, tiempo empleado y número de iteraciones empleadas por el algoritmo para cada conjunto de polígonos agrupados por el número de vértices. Es importante destacar que en RS el número de iteraciones está prefijado y de forma constante en la entrada del algoritmo al generar la nube de puntos a analizar.

Comparando los datos de la Tabla 7.11 con los mejores resultados obtenidos por SA, que

son los expuestos en la Tabla 7.4 de la página 129, observamos que RS mejora cada una de las variables analizadas: porcentaje de área iluminada, tiempo de ejecución y número de iteraciones.

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	51.8114	5.52	2500
100	34.9657	38.74	5000
150	24.5818	114.86	7500
200	21.5381	215.48	10000

Tabla 7.11: Resultados para la heurística RS $k = 50$

Este resultado se debe a que SA escanea regiones alejadas del óptimo con determinada probabilidad. Si se toma en SA una temperatura de parada inferior a la elegida para realizar los análisis, $T_f < 0.005$, los resultados obtenidos por SA mejoran a los obtenidos por RS, pero el tiempo de respuesta es mucho mayor. Por tanto, podemos concluir que SA es un algoritmo que produce buenos resultados frente a otras técnicas heurísticas, pero en contrapartida obtenemos una lenta convergencia frente a RS. En este sentido RS proporciona buenas soluciones con un tiempo de respuesta rápido.

Debemos destacar que RS estaba basado en la generación aleatoria de un número kn puntos interiores al polígono P de n vértices, (en las experimentaciones se ha elegido $k = 50$). La elección de este parámetro k está directamente relacionado con la rapidez y la solución obtenida. Cuando mayor sea su valor más próximo al óptimo estará la solución obtenida y mayor será, por tanto, el tiempo de respuesta del algoritmo. Presentamos en la Figura 7.9 los resultados obtenidos en la Tabla 7.11 en forma de gráfica relacionando el número de vértices con el porcentaje de área iluminada, comparado con los resultados obtenidos por el resto de heurísticas.

7.6.4 Comparativa con GRAD

La heurística $GRAD$ está basada en la idea de que toda *luz-punto* interior a un polígono P puede moverse en el interior de P, según un gradiente positivo de área iluminada. Así, si colocamos una luz en cada vértice de P y hacemos converger dichos puntos en el sentido del gradiente, cada luz tenderá a un máximo local. El método convergería más rápidamente si colocásemos menor número de puntos iniciales, es decir, si no colocásemos una luz en cada vértice de P. Sin embargo, como se muestra en la Figura 7.8, el número de máximos locales puede llegar a ser $O(n)$. Por ello se debe inicializar $GRAD$ colocando una luz en cada vértice del polígono P.

La experimentación realizada permite deducir que el máximo global buscado se encuentra entre uno de estos máximos locales, pudiendo ser más de un máximo global, (por ejemplo si P tiene núcleo). Como se puede comprobar en la Tabla 7.12 los resultados obtenidos por $GRAD$ son sorprendentemente buenos comparando con todas las heurísticas anteriores. $GRAD$ mejora los porcentajes de área iluminada para los cuatro conjuntos de polígonos estudiados, (con 50, 100, 150 y 200 vértices), en un tiempo aunque superior a RS, pero aceptable. Debe hacerse notar que el número de iteraciones de $GRAD$ aumenta respecto a las heurísticas anteriores. Esto se debe a que de los v vecinos analizados para mover cada *luz-punto* en el sentido positivo del

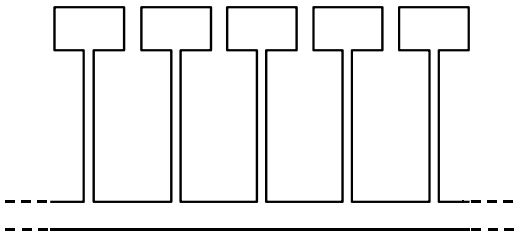


Figura 7.8: Un polígono con $O(n)$ máximos locales para el problema MaxA-p-Pv1(P)

gradiente muchos estarán fuera de P y por tanto no debe construirse su polígono de visibilidad y así calcular su región iluminada, con lo que el tiempo en estas iteraciones disminuye.

Los resultados que se muestran en la siguiente tabla han sido también obtenidos al ser aplicada $GRAD$ sobre el conjunto de polígonos generado aleatoriamente con RPG, (utilizado también en las anteriores heurísticas), mostrando porcentaje de área iluminada, tiempo de ejecución y número de iteraciones se muestran en la siguiente tabla:

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	53.3707	18.68	28880
100	35.1292	101.54	65792
150	25.5142	240.42	106464
200	21.9597	458.62	142336

Tabla 7.12: Resultados para la heurística GRAD

En la Figura 7.9 mostramos en forma de gráfica los datos de las Tablas 7.4, 7.10, 7.11 y 7.12, que relacionan el número de vértices de P con el porcentaje medio de área iluminada. Podemos comprobar como $GRAD$ obtiene mejores resultados para todos los vértices analizados.

7.6.5 Análisis y conclusiones

Presentamos en este apartado un análisis de los datos expuestos en las tablas anteriores. En primer lugar analizamos si dichos datos son significativamente diferentes para cada heurística y después presentamos a modo de ejemplo, las *curvas de crecimiento* sobre la *función objetivo* que produce la ejecución de cada heurística. Además presentamos las *curvas de crecimiento medio*, obtenidas realizando la media, iteración a iteración, de todas las *curvas de crecimiento* resultado de aplicar cada heurística a 50 polígonos con 100 vértices.

Contraste de hipótesis

La comparativa entre los resultados obtenidos por las diferentes heurísticas tiene sentido, si son significativamente diferentes, es decir, si podemos asegurar que las distribuciones que siguen dichos resultados son distintas. Para comprobar esta situación se han realizado contraste T de hipótesis, (utilizando el software matemático MatLab), con un nivel de significación del 95%,

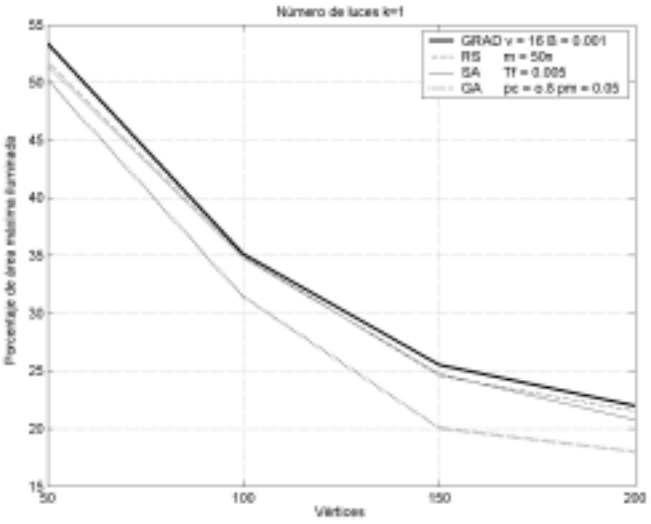


Figura 7.9: Resultados para GRAD con $v = 16$ $\beta = 0.001$ frente a RS, SA y GA

usando los resultados obtenidos sobre los 50 polígonos de 50, 100, 150 y 200 vértices cada uno de ellos, empleados para obtener las medias de las tablas de datos resumen expuestas, respecto al porcentaje iluminado. Las respuestas aportadas por los contrastes realizados se presentan en la siguiente tabla, indicando con el signo '+' si el contraste es significativo y con '–' si no lo es:

1. Contrastes para $n = 50$:

Algoritmo	*SA*	*GA*	*RS*	*GRAD*
SA	•	+	+	+
GA	+	•	+	+
RS	+	+	•	+
GRAD	+	+	+	•

Tabla 7.13: Tabla de contrastes realizados para los datos de polígonos con 50 vértices

2. Contrastes para $n = 100$:

Algoritmo	*SA*	*GA*	*RS*	*GRAD*
SA	•	+	+	+
GA	+	•	+	+
RS	+	+	•	−
GRAD	+	+	−	•

Tabla 7.14: Tabla de contrastes realizados para los datos de polígonos con 100 vértices

Por tanto *RS* y *GRAD* no son significativamente diferentes para 100 vértices.

3. Contrastes para $n = 150$:

Algoritmo	SA	GA	RS	$GRAD$
SA	•	+	+	+
GA	+	•	+	+
RS	+	+	•	+
$GRAD$	+	+	+	•

Tabla 7.15: Tabla de contrastes realizados para los datos de polígonos con 150 vértices

4. Contrastes para $n = 200$:

Algoritmo	SA	GA	RS	$GRAD$
SA	•	+	+	+
GA	+	•	+	+
RS	+	+	•	−
$GRAD$	+	+	−	•

Tabla 7.16: Tabla de contrastes realizados para los datos de polígonos con 200 vértices

Así de los contraste realizados podemos deducir que RS y $GRAD$ no obtienen resultados significativamente diferentes, aunque en media observamos que $GRAD$ obtiene mejor resultado en porcentaje de área iluminada. El resto de heurísticas generan resultados significativamente diferentes.

Curvas de crecimiento

Llamamos *curva de crecimiento* a la curva que relaciona la iteración $i-ésima$ de un algoritmo de optimización con el valor de la *función objetivo* en dicha iteración. Presentamos en este apartado las *curvas de crecimiento* que se genera al aplicar cada una de nuestras heurísticas al polígono de la Figura 7.1. Además para GA presentamos dos curvas: una que muestra la evolución de la *función objetivo* en relación con el número de generaciones y otra en relación con el número de iteraciones. Para la comparativa con el resto de métodos se ha utiliza esta segunda. Finalmente presentamos las *curvas de crecimiento medio* para polígonos con 100 vértices.

1. Mostramos en la Figura 8.15 esta curva para el problema MaxA-p-Pv1(P) aproximado con *simulated annealing* SA, (debe hacerse notar que se visualizan para todas las heurísticas el mismo número de iteraciones, lo que permitirá una comparativa más clara).

   Podemos comprobar como la convergencia es rápida. Además según aumentan las iteraciones de la heurística y por tanto disminuye la temperatura T, disminuye la función de aceptación $e^{\left(\frac{-\delta}{T}\right)}$ y por tanto disminuye la probabilidad de obtener soluciones peores que la actual acercándonos al óptimo buscado. Presentamos acontinuación la *curva de crecimiento* aportada por la heurística *algoritmos genéticos GA.*

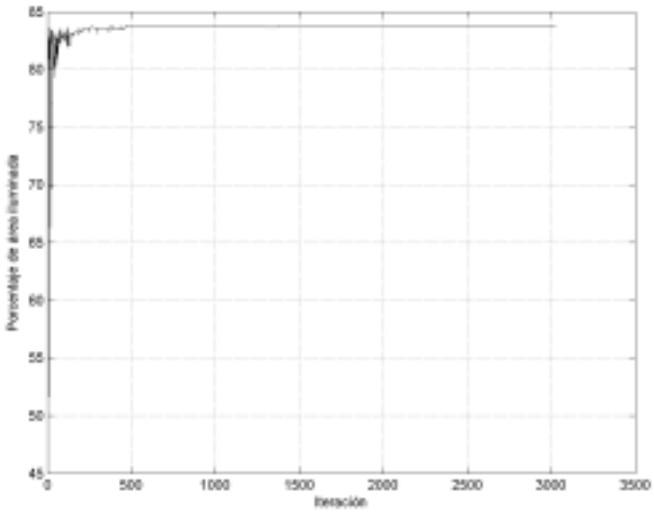
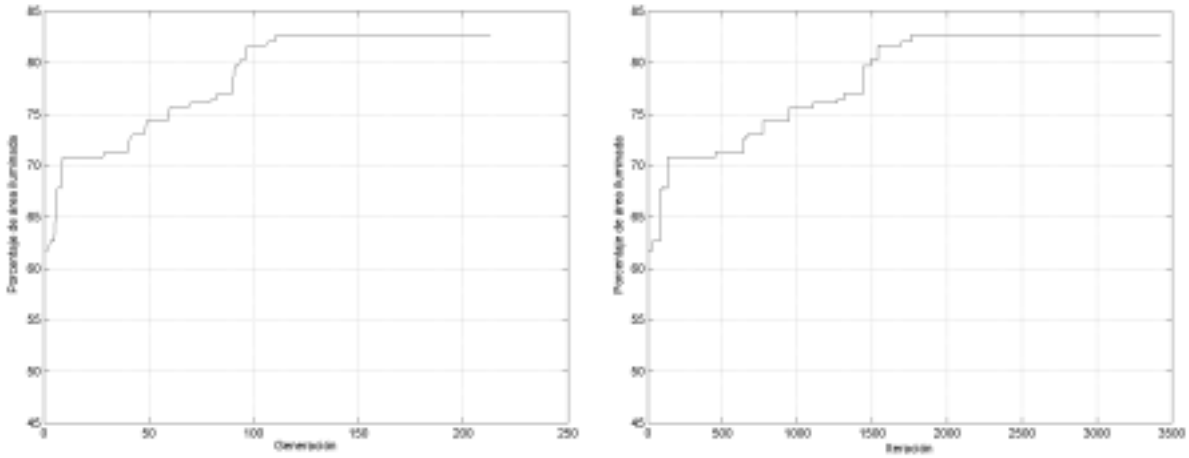


Figura 7.10: Curva de crecimiento para MaxA-p-Pv1(P) con SA

2. Para la heurística GA mostramos en la Figura 8.16 (a), la *curva de crecimiento* respecto a cada generación y en la Figura 8.16 (b), respecto al número de iteraciones, también tomando como polígono inicial el de la Figura 7.1:



(a) (b)

Figura 7.11: Curvas de crecimiento para el problema MaxA-p-P1k(P) con GA

Como podemos comprobar, los operadores diseñados permiten hacer tender la *función objetivo* hacia la solución, estableciendo zonas en las que dicha función permanece estable. Sin embargo la respuesta de la heurística es peor a SA.

3. Mostramos en la siguiente figura, la *curva de crecimiento* que produce RS, al aplicar la heurística al polígono de la Figura 7.1. Se ha considerado un número de puntos interiores

$m = 3500$. Podemos comprobar como la convergencia es también rápida e incluso mejora a SA.

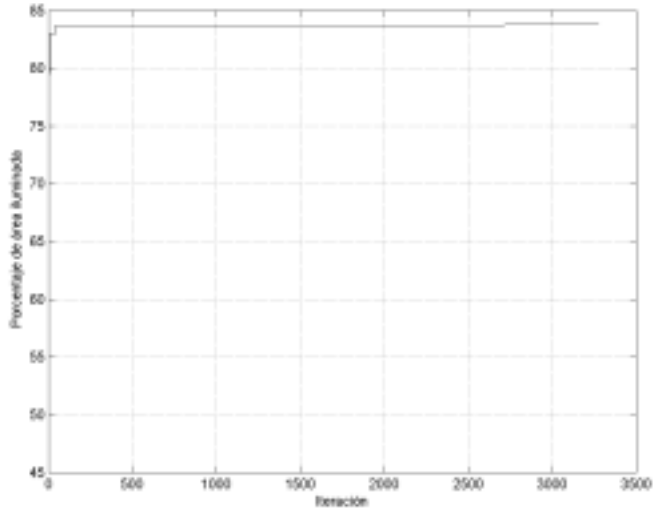


Figura 7.12: Curva de crecimiento para MaxA-p-Pv1(P) con RS

4. La *curva de crecimiento* que se obtiene con la aplicación de $GRAD$ al polígono de la Figura 7.1 se muestra a continuación.

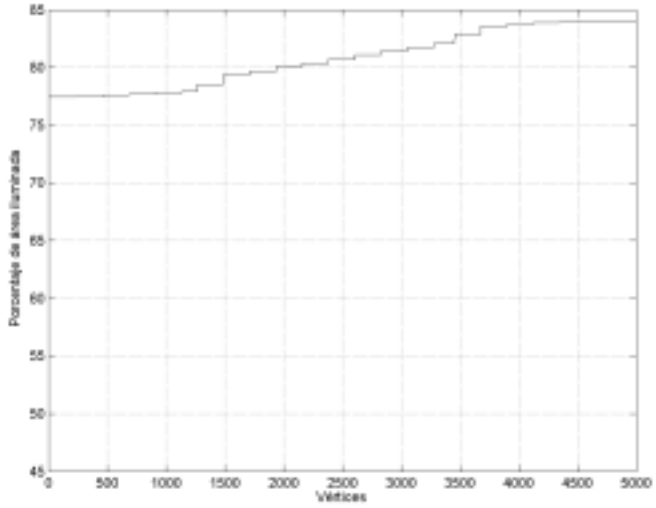


Figura 7.13: Curva de crecimiento para MaxA-p-Pv1(P) con GRAD

Esta curva se ha construido tomando en cada iteración el punto que maximice el área iluminada al mover cada vértice de P según un gradiente positivo de área iluminada. Se ha ejecutado la heurística tomando $v = 16$, (número de puntos que se estudian alrededor de uno dado) y $\beta = 0.001$, (distancia al punto). Aunque en este ejemplo la convergencia

de $GRAD$ es más lenta que SA, mostraremos en la Figura 7.15, como en media $GRAD$ converge más rápidamente que SA, según el número de iteraciones.

Mostramos en la Figura 7.14 todas las *curvas de crecimiento* de este ejemplo conjuntamente. Como podemos comprobar el porcenteje de área iluminada aportado por RS, SA y $GRAD$, son prácticamente iguales. Sin embargo la mejor heurística respecto a la convergencia es RS. Los peores resultados son aportados por GA.

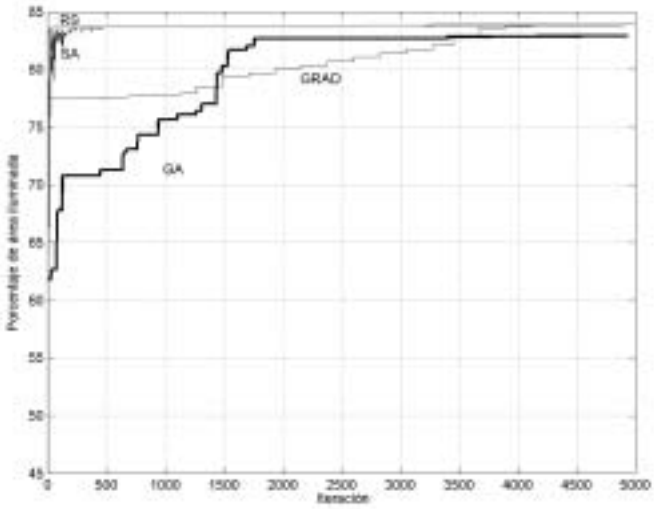


Figura 7.14: Comparativa de curvas de crecimiento para el problema MaxA-p-Pv1(P)

No obstante, estas *curvas de crecimiento* están referidas al polígono de la Figura 7.1. Para poder obtener conclusiones más claras, se ha construido la *curva de crecimiento* medio para cada heurística con polígonos de 100 vértices. Para ello se ha aplicado cada heurística a 50 polígonos de 100 vértices, obteniendo la *curva de crecimiento* para cada polígono y calculando finalmente la media de todas ellas en cada iteración. En la Figura 7.15 se muestran las curvas resultantes.

Así, de las Tablas 7.4, 7.10, 7.11 y 7.12 y de la Figura 7.15, podemos obtener las siguientes conclusiones que agrupamos en tres aspectos: mejor porcentaje de área ilumina por el punto solución, rapidez de convergencia respecto al número de iteraciones y mejor relación *iteraciones/tiempo*.

- **Respecto al porcentaje de área iluminada** por el punto solución, $GRAD$ proporciona los mejores resultados en todos los números de vértices analizados. Sin embargo, el número de iteraciones del método es elevado respecto a las otras heurísticas. Esto es debido a que escaneamos un conjunto v de vecinos de cada punto, cuando lo movemos según un gradiente positivo de área, para focos situados en los vértices del polígono. Así, escanemos rutas de visibilidad que aunque se alejan mucho del óptimo son visitadas.

  La heurística RS obtiene buenos resultados en porcentaje de área iluminada. Aunque sensiblemente superiores a la solución de SA, son muy similares a los aportados por $GRAD$. Además el tiempo y el número de iteraciones de RS es mucho más pequeño que el aportado

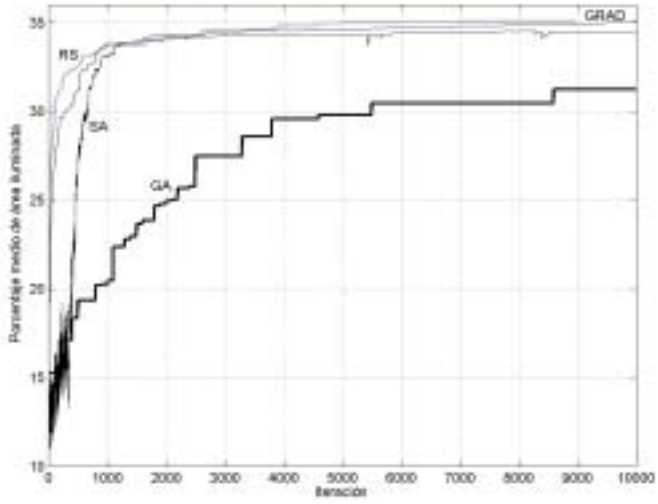


Figura 7.15: Curvas de crecimiento medio para MaxA-p-Pv1(P) con $n = 100$

tanto por $GRAD$, como por SA, para las cuales el tiempo de respuesta es similar. Los peores resultados los proporciona GA. Esto es debido a que para $k = 1$ el operador de cruce diseñado se anula, (ya que cada individuo está formado por un único gen) y por tanto GA se transforma en búsqueda pesudo-aleatoria.

Además de los contrastes realizados se deduce que RS y $GRAD$ no obtienen porcentajes de área iluminada significativamente diferentes.

- **Respecto a la convergencia de las heurísticas**, podemos observar en la Figura 7.15, el mejor método es RS, seguido de $GRAD$ y SA. Por tanto, RS se puede utilizar cuando se desee obtener resultados buenos con una rápida convergencia y sin embargo, la heurística $GRAD$ es recomendable cuando se desea optimizar la solución de MaxA-p-Pv1(P).

- Finalmente analizamos la **relación** *iteraciones/tiempo*. Esta relación puede medir el tiempo empleado por cada técnica en realizar operaciones propias de la heurística, como por ejemplo en GA el cruce o la mutación. En este sentido el mejor método es de nuevo $GRAD$, seguido de RS. Aunque ambos métodos se limitan a analizar puntos en el interior del polígono, RS necesita generar una nube alatoria de puntos a analizar, mientras que $GRAD$ no necesita dicha generación, ya que los puntos a analizar son los vecinos de cada punto.

Con estas conclusiones que acabamos de exponer, cabe hacerse una pregunta. Si una búsqueda aleatoria para el problema MaxA-p-Pv1(P), obtiene resultados más que aceptables, ¿será posible encontrar un algoritmo exacto que solucione dicho problema?. En la Sección 7.8, analizamos esta posibilidad.

7.7 Sobre el porcentaje de área iluminada

Una vez solucionado el problema MaxA-p-Pv1(P), nos podemos hacer la siguiente pregunta: *"¿existe alguna relación entre el número de vértices n del polígono P y el área iluminada por la luz-punto de máxima iluminación?"*. Evidentemente para un polígono concreto P la respuesta a esta pregunta no tiene ningún interés científico, pues existirán polígonos con 100 vértices por ejemplo, que se iluminarán totalmente, (si P es un polígono estrellado) y otros polígonos con el mismo número de vértices donde el porcentaje iluminado cambiará sustancialmente. Así, realmente la pregunta a la que queremos dar respuesta en esta sección es la siguiente:

"Si tomásemos un número suficientemente grande de polígonos P con n vértices y calculásemos la luz-punto de máxima iluminación, ¿cuál sería en media el porcentaje de área iluminada?, ¿existe en media alguna relación entre n y el porcentaje de área iluminada?"

Este problema que hemos denotado como PorA-p-Pv1(P) en la página 108, podemos enunciarlo de la siguiente manera:

porcentaje de área máxima iluminada por un punto interior a un polígono PorA-p-Pv1(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cual es el porcentaje medio máximo de área iluminada por 1 *luz* interior a P?

Para dar respuesta a esta pregunta se ha realizado un estudio empírico usando para ello dos de las heurísticas diseñadas para solucionar el problema MaxA-p-Pv1(P), como son *simulated annealing* (SA) y *random search* (RS), descritas en las Secciones 7.2 y 7.4 respectivamente. Utilizando el generador aleatorio de polígonos, (RPG), descrito en el Apéndice A, cuya implementación se encuentra en el Apéndice B de listados, se han generado 50 polígonos de n vértices para todo n con $5 \leq n \leq 300$ y múltiplo de 5. Es decir, se han generado aleatoriamente 50 polígonos con $5, 10, 15, 20, ..., 295, 300$ vértices y a cada polígono se le ha aplicado SA y RS, para obtener la *luz-punto* p de máxima iluminación, calculando a continuación el porcentaje de área iluminada por este punto p con respecto al área total de P. Es importante destacar que no se ha utilizado en esta sección la mejor heurística respecto a la aproximación para MaxA-p-Pv1(P), (como podría ser $GRAD$), ni se ha utilizado la mejor combinación de parámetros de SA, (como podrían ser los descritos en el Caso 4). Esto ha sido debido a que el conjunto de polígonos sobre el que se han realizado las experimentaciones es muy grande, (3000 polígonos aleatorios), y la utilización de una heurística de convergencia muy lenta haría interminable la obtención de conclusiones. Por ello, se han utilizado tanto para RS como para SA parámetros que permitan obtener conclusiones aceptables en tiempos también aceptables. Como comprobaremos más adelante la igualdad entre los resultados obtenidos por RS y SA nos permiten concluir que las respuestas obtenidas para PorA-p-Pv1(P) son coherentes.

Una vez obtenidas las correspondientes tablas de datos, se ha realizado un ajuste de datos exponencial, linealizando los datos a una función de la forma $f(x) = Ce^{Ax}$ y obteniendo el ajuste por mínimos cuadrados, en el intento de buscar una relación para el problema PorA-p-Pv1(P).

De forma esquemática el proceso seguido se puede enumerar de la siguiente manera:

1. Generar aleatoriamente 50 polígonos de n vértices, $\forall n \in N$, con $N = \{5m \ / \ 1 \leq m \leq 60\}$.
2. Para cada polígono generado obtener dos soluciones para MaxA-p-Pv1(P). Una mediante el algoritmo heurístico basado en SA, tomando con $T_0 = 100.0$ y decrecimiento de temperatura FSA y otra mediante RS generando en el interior del polígono una nube de puntos de cardinal constante e igual a 1000.
3. Para cada $n \in N$, obtener la media de los resultados obtenidos.
4. Buscar el ajuste por mínimos cuadrados más adecuado.

Vért. \ Pol.	1	2	3	4	5	6	7	8	9	10
20	38.15	88.33	87.76	80.44	89.53	93.38	87.93	49.82	50.86	94.82
40	29.09	49.57	63.71	46.07	67.39	71.36	55.91	41.46	56.95	61.95
60	28.22	31.95	47.51	41.98	56.16	37.02	35.45	35.68	42.92	51.89
80	27.40	23.53	27.61	47.52	34.49	38.03	41.68	33.68	23.85	36.91
100	25.68	34.85	29.40	18.99	37.07	30.78	38.26	30.75	20.93	32.76
120	35.21	39.01	31.39	33.71	18.55	23.19	24.78	28.02	24.74	35.44
140	16.88	29.70	12.59	21.39	34.36	30.80	21.57	24.69	15.73	17.58
160	14.45	19.68	20.45	30.09	22.47	18.83	34.61	18.76	31.06	26.67
180	21.23	11.79	17.64	35.82	19.42	12.83	20.03	14.51	24.34	24.81
200	6.43	9.67	14.23	17.42	13.53	6.21	13.71	32.08	6.16	16.54

Tabla 7.17: Resumen de resultados obtenidos para PorA-p-Pv1(P) con SA

Dado que las tablas de resultados obtenidas son excesivamente extensas, presentamos en la Tabla 7.17 y en la Tabla 7.18 los resultados obtenidos de forma resumida para un número reducido de vértices y de polígonos.

Vért.\ Pol.	1	2	3	4	5	6	7	8	9	10
20	82.89	80.54	60.07	86.71	76.62	94.39	78.95	63.88	81.79	91.03
40	60.53	63.05	53.96	81.23	65.50	66.11	61.69	43.68	69.58	69.10
60	57.29	48.58	40.50	54.07	48.22	27.06	27.71	33.65	32.42	31.74
80	36.85	53.25	29.35	33.21	33.28	28.70	30.06	44.87	36.20	42.30
100	32.03	28.61	36.72	25.92	25.84	25.66	31.08	21.58	31.30	28.68
120	37.69	19.90	45.24	21.85	18.87	23.03	35.73	22.96	23.00	21.07
140	25.14	21.88	22.89	32.33	25.44	25.03	28.20	27.11	35.05	23.36
160	26.00	22.19	18.82	27.86	22.01	29.07	24.94	23.58	33.95	22.37
180	17.68	19.72	19.97	15.94	15.01	18.13	21.47	19.47	17.96	16.10
200	29.27	17.97	16.98	23.44	14.07	24.09	18.67	15.24	19.59	18.28

Tabla 7.18: Resumen de resultados obtenidos para PorA-p-Pv1(P) con RS

Los resultados expuestos pertenecen a los 10 primeros polígonos estudiados para cada número de vértices entre 5 y 200, ($n \in T$ con $T = \{10m \ / \ 2 \leq m \leq 20\}$). En la primera tabla presentamos los resultados obtenidos con SA y a continuación igualmente un subconjunto de los resultados obtenidos con RS se exponen en la Tabla 7.18.

Una vez obtenidos los porcentajes de iluminación, se ha realizado $\forall n \in N$, donde $N = \{5m \ / \ 1 \leq m \leq 60\}$ la media aritmética del porcentaje de área iluminada en cada polígono. Estos resultados en relación con el número de vértices se presenta en la Tabla 7.19, para los datos obtenidos con *SA.*

Vért.	Porc.	Vért.	Porc.	Vért.	Porc.	Vért.	Porc.	Vért.	Porc.
5	98.6566	65	44.2580	125	27.8563	185	19.3340	245	13.2700
10	94.9616	70	41.2938	130	26.7448	190	17.3852	250	11.6692
15	90.3411	75	36.0209	135	26.3963	195	17.2018	255	13.0079
20	79.5730	80	36.2998	140	23.3431	200	17.4670	260	12.8916
25	71.1568	85	35.5531	145	25.6746	205	12.9919	265	12.6103
30	66.4919	90	33.5678	150	22.5237	210	14.8448	270	10.8111
35	58.9272	95	33.1926	155	23.2230	215	13.7625	275	11.2890
40	55.3656	100	33.1054	160	24.0753	220	13.8101	280	12.6866
45	56.2684	105	29.4391	165	21.5645	225	14.0747	285	11.5929
50	51.0337	110	28.9052	170	19.7577	230	14.9315	290	11.3349
55	48.4308	115	30.1182	175	20.8267	235	12.7207	295	11.8130
60	44.3851	120	29.0504	180	19.4689	240	13.8738	300	9.8661

Tabla 7.19: Media de procentajes obtenidos para PorA-p-Pv1(P) con SA

La Tabla 7.20 muestra los porcentajes medios de área iluminada obtenidos con *random search*, (RS). Cabe destacar en este caso que los resultados medios obtenidos son mejores que los obtenidos para el algoritmo *SA,* (como se concluía en la Sección 7.6).

Vért.	Porc.	Vért.	Porc.	Vért.	Porc.	Vért.	Porc.	Vért.	Porc.
5	99.9963	65	40.6310	125	27.5746	185	19.2084	245	16.2833
10	95.5174	70	43.9509	130	26.9230	190	18.9315	250	15.6145
15	86.6315	75	40.2370	135	26.9035	195	19.2759	255	15.8374
20	80.2708	80	37.9494	140	26.4575	200	19.4373	260	15.4738
25	78.8411	85	34.9265	145	23.5207	205	18.7137	265	15.6835
30	67.0270	90	35.0069	150	25.2084	210	18.5222	270	14.7700
35	65.1300	95	34.5968	155	21.8473	215	17.6543	275	13.8955
40	59.7022	100	33.7020	160	22.8641	220	17.1431	280	15.4472
45	52.8365	105	32.4159	165	24.3561	225	16.3656	285	15.0835
50	51.5249	110	31.4142	170	22.1526	230	17.7643	290	14.3062
55	51.3046	115	29.8624	175	22.1181	235	16.5838	295	14.7489
60	45.8039	120	28.5371	180	20.4027	240	16.4227	300	12.5990

Tabla 7.20: Media de procentajes obtenidos para PorA-p-Pv1(P) con RS

Se ha realizado a continuación un ajuste por mínimos cuadrados para los puntos que relacionan el número de vértices con el porcentaje medio iluminado. Exponemos a continuación el mecanismo de linealización seguido.

Linealización de datos a la función $y = Ce^{Ax}$

Dado un conjunto de m puntos con las abcisas distintas $\{(x_i, y_i)\}_{i=1}^{m}$ hemos aproximado por una curva exponencial de la forma

$$y = Ce^{Ax} \tag{7.7.19}$$

Tomando logaritmos en ambos lados de la igualdad tenemos:

$$\ln(y) = Ax + \ln(C) \tag{7.7.20}$$

y realizando el cambio de variables $Y = \ln(y)$, $X = x$ y $B = \ln(C)$, resulta la realción lineal con las nuevas variables X e Y:

$$Y = AX + B \tag{7.7.21}$$

Los puntos originales (x_i, y_i) $(1 \leq i \leq m)$ se han transformado con el cambio de variables en $(X_i, Y_i) = (x_i, \ln(y_i))$. Este proceso se denomina *método de linealización de datos*. Las ecuaciones normales de Gauss para la ecuación $Y = AX + B$ serán las de la recta de regresión en mínimos cuadrados de los puntos $\{(X_i, Y_i)\}_{i=1}^{m}$:

$$\begin{cases} A\left(\sum_{i=1}^{m} X_i^2\right) + B\left(\sum_{i=1}^{m} X_i\right) = \sum_{i=1}^{m} X_i Y_i \\ A\left(\sum_{i=1}^{m} X_i\right) + mB = \sum_{i=1}^{m} Y_i \end{cases} \tag{7.7.22}$$

Conocidos los valores de los coeficientes A y B, se halla el valor del parámetro $C = e^B$. Aplicando este mecanismo a los datos expuesto en las Tablas 7.19 y 7.20 se ha obtenido un ajuste exponencial, que se analiza a continuación y que muestra como la disminución de área sigue un decrecimiento exponencial en función de los vértices del polígono.

Ajuste de datos obtenido

La función exponencial obtenida utilizando este mecanismo de ajuste por *mínimos cuadrados* para ajustar los datos de la Tabla 7.19 es:

$$f_{SA}(x) = 90.775448\ e^{-0.00938x} \tag{7.7.23}$$

En la Figura 7.16 presentamos los resultados de la Tabla 7.19 y el ajuste exponencial obtenido. Como se puede observar la curva de datos se ajusta satisfactoriamente a $f_{SA}(x)$, de donde podemos concluir que el porcentaje de área iluminado por la *luz-punto* de máxima iluminación disminuye de forma exponencial. Analógamente en la Figura 7.17 presentamos los resultados del mismo estudio pero realizados con el algoritmo de *RS*, y que ajustarán por tanto los resultados presentados en la Tabla 7.20. Como se puede observar también la curva de datos se aproxima satisfactoriamente a la curva obtenida para el ajuste exponencial, que es en este caso:

$$f_{RS}(x) = 89.717914\ e^{-0.008633x} \tag{7.7.24}$$

Presentamos en la Figura 7.18 la comparativa de las gráficas obtenidas para el ajuste exponencial para el algoritmo *SA* como *RS*, que muestra evidentemente como los resultados obtenidos por ambas extrategias aportan la misma conclusión aproximada para el problema MaxA-p-Pv1(P), esto es, que existe en media una relación Ce^{-Ax} entre el número de vértices de un polígono P y el porcentaje iluminado por la *luz-punto* de máxima iluminación interior a P.

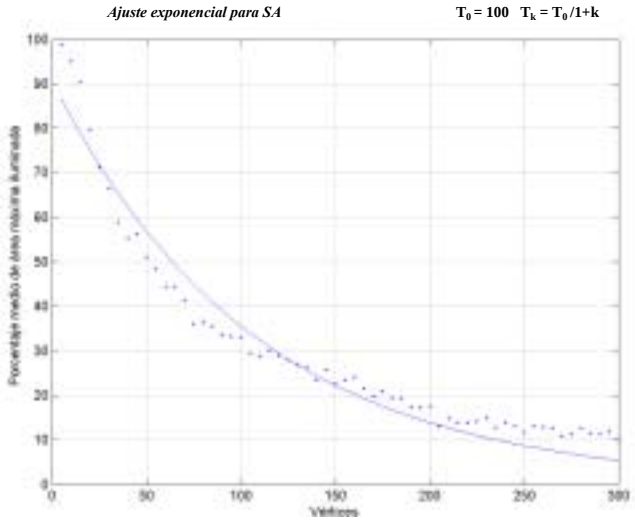


Figura 7.16: Ajustes por mínimos cuadrados para MaxA-p-Pv1(P) con SA

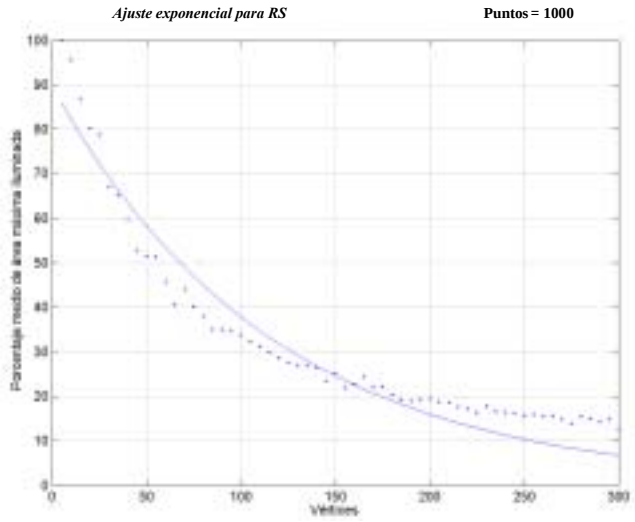


Figura 7.17: Ajustes por mínimos cuadrados para MaxA-p-Pv1(P) con RS

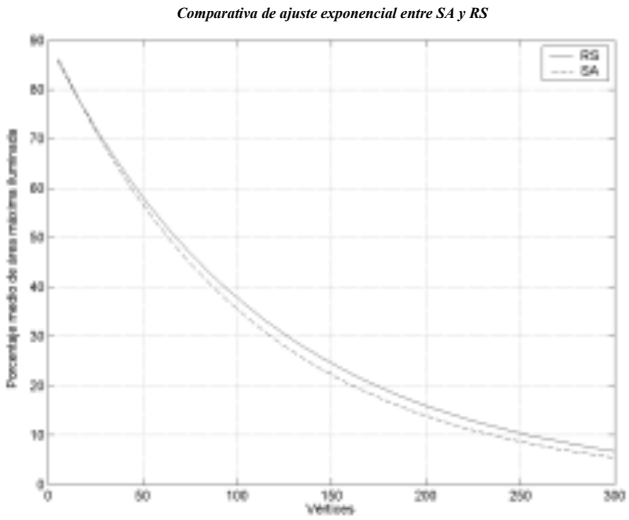


Figura 7.18: Comparativa de ajuste cuadrático con SA y RS

7.8 Aproximación a un algoritmo exacto para MaxA-p-Pv1(P)

Las soluciones aportadas en las secciones anteriores permiten encontrar de forma aproximada el punto de máxima iluminación interior a un polígono P de n vértices. Podemos preguntarnos ahora si existirá un algoritmo exacto que solucione el problema MaxA-p-Pv1(P). Como se mencionó anteriormente en las referencias bibliográficas en las que se estudia este problema [23], se diseñan también algoritmos aproximados para la búsqueda del punto de máxima iluminación.

Presentamos en esta sección una prueba pseudo-experimental para un algoritmo exacto de complejidad polinomial $O(n^5)$, que soluciona nuestro problema.

7.8.1 Superficie de áreas

Utilizando las implementaciones diseñadas en esta memoria para la búsqueda del punto de máxima iluminación, podemos construir lo que llamaremos la *superficie de áreas* de un polígono de n vértices P. Si a cada punto interior a P se le asocia su porcentaje de área iluminada respecto al área total de P, podemos construir una superficie que mostrará el comportamiento de la función de áreas de los puntos interiores al polígono. Observando estas superficies podremos comprobar si el punto de máxima iluminación se sitúa en alguna región específica del polígono que tenga características especiales respecto a la forma de P.

El método utilizado para generar dicha superficie ha consistido en tomar una nube de puntos interiores al polígono P. Para esta nube se ha construido la triangulación de Delaunay, [82]. Si el cardinal de la nube es suficientemente grande y a cada punto se le asocia su porcentaje de área iluminada, podemos construir un mallado manteniendo la estructura plana de la triangulación de Delaunay que representa nuestra superficie de áreas. En forma esquemática general los pasos seguidos para la construcción de la superficie han sido los siguientes, (recuérdese que los polígonos que utilizamos en nuestras experimentaciones ha sido generados por RPG en el cuadrado

unidad).

1. Generar en el cuadrado unidad, $((0,1) \times (0,1))$, un mallado rectilíneo M donde cada región del mallado es un cuadrado de lado α, con α suficientemente pequeño. En nuestras experimentaciones se ha considerado $\alpha = 0.005$.

2. Construir una nube de puntos N formada por todos los vértices del mallado M interiores al polígono P.

3. Generar la triangulación de Delaunay de puntos de N.

4. Para cada punto $p \in N$ construir su *polígono de visibilidad*, (se ha utilizado un algoritmo lineal [72]), y calcular su área.

5. Obtener la superficie que se genera asociando a cada punto $p \in N$ su área iluminada manteniendo las aristas de la triangulación de Delaunay.

Las estructuras de datos, así como los códigos y detalles de las implementaciones que permiten construir la superficie de áreas y su visualización mediante el paquete matemático MatLab, se encuentran en el Apéndice B de listados.

Si generamos aleatoriamente un polígono P y construimos su superficie de áreas obtenemos un resultado como el que se muestra en la Figura 7.19, donde se representa el polígono P y su superficie de áreas asociada.

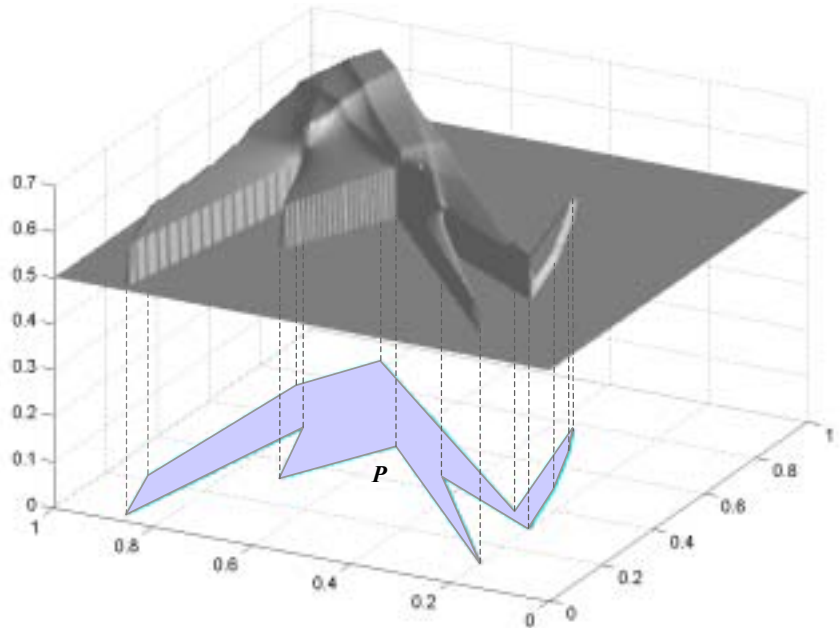


Figura 7.19: Superficie de Áreas para un polígono P con 15 vértices

Como se puede apreciar en la figura anterior la superficie de áreas produce una descomposición, (que denotaremos con $\mathcal{S}$), de P en regiones de visibilidad. Estas regiones están generadas

por el arreglo de semirrectas que se obtienen al prolongar los lados de P que inciden en los vértices cóncavos. En la Figura 7.20 se muestra la planta de la superficie de áreas, donde se puede apreciar con mayor claridad la descomposición.

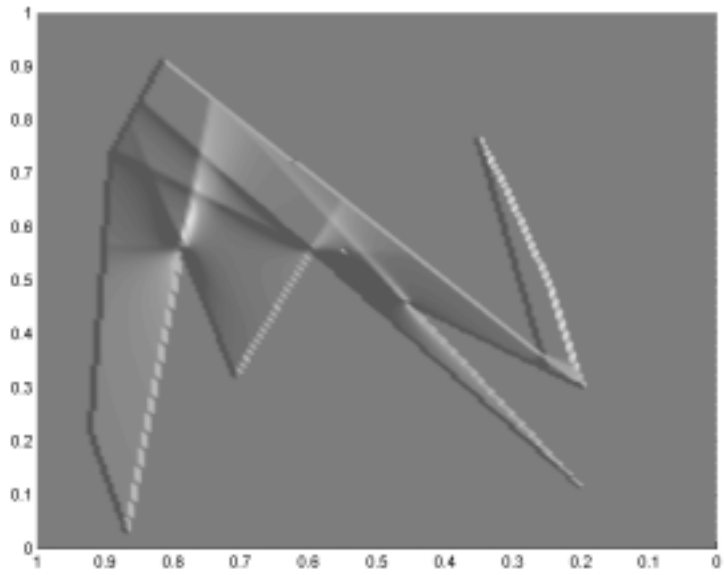


Figura 7.20: Planta de la superficie de áreas para el polígono de la Figura 7.19

Esta descomposición está contenida en la descomposición presentada en 1992 por Bose y otros [15], (que denotaremos con $\mathcal{T}$), de un polígono P en regiones de visibilidad, para realizar un estudio sobre vigilancia de vértices. Resumimos en el siguiente apartado los elementos más importantes de la descomposición analizada por Bose.

7.8.2 Descomposición de un polígono P en regiones de visibilidad

Un polígono P de n vértices se puede descomponer en un conjunto $\mathcal{T} = \{R_1, R_2, ..., R_m\}$ de *regiones de visibilidad*, (ver Figura 7.21). Esta descomposición se describe en [15] y es la descomposición inducida por los segmentos obtenidos al unir cada vértice de P con los vértices cóncavos visibles desde él, (llamaremos W_i al conjunto de segmentos generados por cada vértice v_i). Realmente es una particularización para los vértices de P, del concepto de *ventana* para un punto q interior a P, que se presenta en este mismo artículo.

En [15] se demuestra que estas regiones R_i tienen características importantes con relación a la visibilidad entre las que destacamos las siguientes:

- Toda *región de visibilidad* R_i en un polígono simple P es convexa.

- Desde todos los puntos de una *región de visibilidad* R_i se ve el mismo conjunto de vértices de P. Lamaremos a este conjunto, *conjunto de visibilidad.*

- Los cardinales de los *conjuntos de visibilidad* asociados a dos regiones R_i y R_j que compartan una arista, se diferencian en una unidad. Es decir desde regiones adyacentes se ve un vértice más o un vértice menos de P.

- Hay $O(n^3)$ *regiones de visibilidad* en un polígono simple con n vértices y existen polígonos con n vértices que tienen $\Theta(n^3)$ *regiones de visibilidad.*

- Al conjunto $\mathcal{T} = \{R_1, R_2, ..., R_m\}$ de *regiones de visibilidad* se le puede asociar un digrafo dual G. Este digrafo contendrá una arista a_{ij} entre los nodos n_i y n_j, asociados a las regiones de visibilidad R_i y R_j de $\mathcal{T}$ si desde R_i se ve un vértice de P más que desde R_j. Así este digrafo G tendrá fuentes y sumideros. Si P es un polígono con núcleo, G tendrá una única fuente asociada precisamente al núcleo de P.

- El digrafo G inducido por una descomposición $\mathcal{T}$ de un polígono P con n vértices tiene $O(n^2)$ sumideros y existen polígonos P con n vértices cuyos digrafos asociados tienen $\Theta(n^2)$ sumideros.

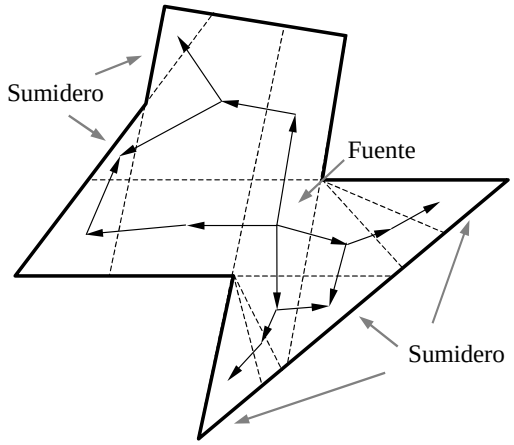


Figura 7.21: Descomposición $\mathcal{T}$ de un polígono en regiones de visibilidad y su digrafo dual

Podemos preguntarnos ahora que relación existe entre esta descomposición del polígono P y la descomposición que aparece al generar nuestras superficies de áreas: la descomposición $\mathcal{S}$, está contenida en la descomposición de Bose $\mathcal{T}$ en el sentido que se indica en el siguiente lema.

Lema 7.8.1 *Si P es un polígono de n vértices, entonces el arreglo de segmentos que generan la descomposición $\mathcal{S}$ de P está contenido en el arreglo de segmentos que generan la descomposición $\mathcal{T}$.*

Demostración. Por definición, tanto la descomposición $\mathcal{S}$ como la descomposición $\mathcal{T}$ están generadas por las semirrectas que se obtienen al unir los vértices cóncavos de P con un conjunto de vértices del polígono. Si v es un vértice cóncavo aparecen alrededor de él en forma de abanico este conjunto de semirrectas, (ver Figura 7.21), de tal forma que las dos semirrectas más internas del

abanico corresponden con la prolongación de los lados de P incidentes en v. Por construcción en $\mathcal{S}$ sólamente aparecen estas semirrectas internas, mientras en $\mathcal{T}$ tendremos el abanico completo de semirrectas. ■

Presentamos en la Figura 7.26 la descomposición $\mathcal{S}$ del polígono de la Figura 7.21, donde se presenta la descomposición $\mathcal{T}$.

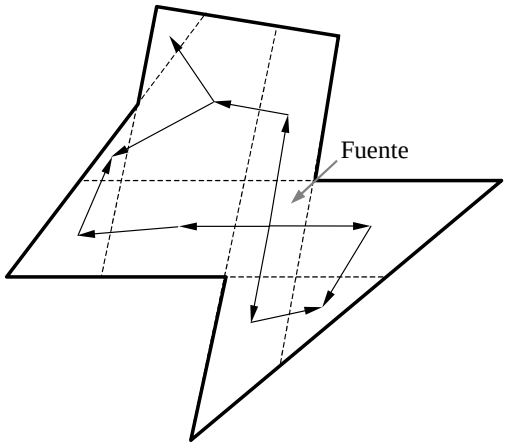


Figura 7.22: Descomposición $\mathcal{S}$ de un polígono en regiones de visibilidad y su digrafo dual

La descomposición $\mathcal{S}$ juega un papel esencial respecto a la búsqueda del punto de máxima iluminación que solucione el problema MaxA-p-Pv1(P). Las características de esta descomposición y su utilización para resolver nuestro problema se analizan en la siguiente sección.

7.8.3 Las descomposiciones $\mathcal{S}$ y el problema MaxA-p-Pv1(P)

Las descomposiciones $\mathcal{S}$ y $\mathcal{T}$ juegan un papel esencial para la búsqueda del punto de máxima iluminación interior a un polígono P. Una característica importante de $\mathcal{S}$ será la cantidad de regiones de visibilidad en que divide a P, que debe ser inferior a la cantidad de $O(n^3)$, que son las regiones en que $\mathcal{T}$ divide a P. Demostramos en el siguiente lema que la descomposición $\mathcal{S}$ divide a P en $O(n^2)$ *regiones de visibilidad.*

Lema 7.8.2 *Si P es un polígono simple de n vértices y $\mathcal{S} = \{R_1, R_2, ..., R_m\}$ la descomposición en regiones de visibilidad descrita, entonces hay $O(n^2)$ regiones de visibilidad y existen polígonos con n vértices que tienen $\Theta(n^2)$ regiones de visibilidad.*

Demostración. El conjunto de segmentos W que determina la descomposición $\mathcal{S}$, estará formado por la frontera de P, que denotamos con $bd(P)$, más el conjunto de dos segmentos W_i generados por cada vértice cóncavo v_i de P. Así, la subdivisión inducida por $W = bd(P) \cup \bigcup_{i=1}^{n} W_i$, es precisamente el conjunto de *regiones de visibilidad.* Además cada conjunto W_i se puede construir en tiempo constante ya que está formado por los segmentos que se obtienen al prolongar los dos lados incidentes en cada vértice cóncavo. Esto implica que la construcción del conjunto W tiene

un coste $O(n)$. Además el número de intersecciones de dos segmentos es a lo más uno, con lo cual el número máximo de intersecciones es $O(n^2)$.

Según la Fórmula de Euler, ([14]), si G es un grafo plano conexo $v - e + f = 2$, donde v es el número de vértices, e es el número de aristas y f es el de caras de G. También se sabe que $e \leq 3v - 6$. Por tanto

$$f \leq 2 - v + 3v - 6 = 2v - 4 \approx n^2 - 4 \tag{7.8.25}$$

de donde se deduce que el número de regiones es $O(n^2)$.

Por otra parte el polígono de la Figura 7.23 es un polígono con $\Theta(n^2)$ *regiones de visibilidad* en la descomposición $\mathcal{S}$.

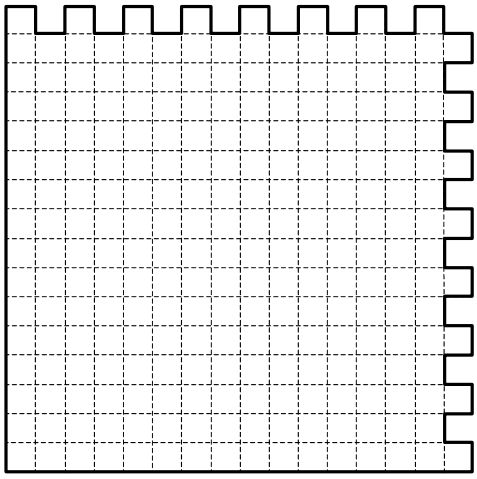


Figura 7.23: Un ejemplo con $O(n^2)$ regiones de visibilidad en la descomposición $\mathcal{S}$

En este ejemplo observamos como cada almena superior genera $\frac{n}{4}$ regiones de visibilidad por cada almena de la derecha. Esto producirá $\frac{n^2}{16}$ regiones de visibilidad, a las que es preciso sumar las regiones generadas por las propias almenas que son $\frac{n}{4}$. Con lo que queda probado la existencia de polígonos que tienen $O(n^2)$ regiones de visibilidad en su descomposición $\mathcal{S}$. ■

La cuestión fundamental ahora es determinar que relación existe entre las regiones de la descomposición $\mathcal{S}$ y el punto de máxima iluminación solución del problema MaxA-p-Pv1(P). Observando las *superficies de áreas*, podemos apreciar como el punto de máxima iluminación se encuentra sobre un vértice de una región $R_i \in \mathcal{S}$. Así enunciamos la siguiente conjetura:

Conjetura 7.8.3 *Dado un polígono P con n vértices, el punto de máxima iluminación se encuentra sobre un vértice de una región de visibilidad $R_i \in \mathcal{S}$.*

Para comprobar esta conjetura, (para la que no se ha encontrado una demostración exacta y que por tanto será estudio de futuras investigaciones), se ha realizado un estudio experimental sobre polígonos generados aleatoriamente con nuestro generador RPG. Para ello se ha implementado la descomposición $\mathcal{S}$ descrita anteriormente, (los códigos correspondientes se encuentran

en el Apéndice B), hallando los vértices de las regiones $R_i \in \mathcal{S}$ y para cada uno de ellos se ha calculado su área iluminada, tomando finalmente el mejor de ellos respecto al área iluminada.

En la siguiente tabla presentamos los resultados obtenidos con este algoritmo para 15 polígonos con vértices entre 30 y 100 vértices. Los resultados obtenidos tomando el mejor vértice de las regiones $R_i \in \mathcal{S}$ han sido comparados con los resultados que se obtendrían si aplicásemos a estos mismo polígonos la heurística $GRAD$, (que es la que mejor resultado ha obtenido según los estudios experimetales expuestos anteriormente). Como se puede observar el vértice de las regiones $R_i \in \mathcal{S}$ siempre mejora el área iluminada y $GRAD$ hace tender la solución hacia ese vértice. Por tanto, parece claro que el punto de máxima iluminación está en uno de los vértices de las regiones R_i pertenecientes a la descomposición $\mathcal{S}$ de P en *regiones de visibilidad*. Como hemos demostrado en el Lema 7.8.2 el número de estas regiones es $O(n^2)$ siendo n el número de vértices del polígono P, lo que nos permitirá diseñar un algoritmo de complejidad $O(n^3)$ para solucionar el problema MaxA-p-Pv1(P). Es importante destacar que la utilización de la descomposición $\mathcal{S}$ para solucionar el problema de la búsqueda del punto de máxima iluminación se presenta en esta memoria por primera vez. Además es la primera vez también que se presenta, (aunque sea de forma experimental), un algoritmo exacto para solucionar el problema MaxA-p-Pv1(P).

Polígono	Vértices	Área $GRAD$	Punto $GRAD$	Área $R_i \in \mathcal{S}$	Punto $R_i \in \mathcal{S}$
1	30	0.200991	(0.1223, 0.6804)	0.200995	(0.1225, 0.6801)
2	35	0.259864	(0.2372, 0.0208)	0.260086	(0.2368, 0.0208)
3	40	0.239474	(0.8948, 0.3907)	0.239565	(0.8950, 0.3913)
4	45	0.397329	(0.0363, 0.7863)	0.400391	(0.0364, 0.8054)
5	50	0.335408	(0.1109, 0.2543)	0.335433	(0.1109, 0.2542)
6	55	0.212166	(0.1106, 0.3482)	0.212238	(0.1105, 0.3483)
7	60	0.234215	(0.9736, 0.2101)	0.236457	(0.9747, 0.2227)
8	65	0.164105	(0.9208, 0.3368)	0.164116	(0.9209, 0.3372)
9	70	0.163398	(0.0566, 0.4257)	0.163622	(0.0551, 0.4190)
10	75	0.159547	(0.8432, 0.0998)	0.159550	(0.8429, 0.0998)
11	80	0.167627	(0.0552, 0.5607)	0.167630	(0.0529, 0.5608)
12	85	0.160772	(0.0009, 0.6471)	0.160779	(0.0007, 0.6472)
13	90	0.129282	(0.9651, 0.6029)	0.129295	(0.9651, 0.6030)
14	95	0.159435	(0.0071, 0.7953)	0.159459	(0.0070, 0.7951)
15	100	0.131729	(0.0727, 0.7946)	0.131771	(0.0726, 0.7949)

Tabla 7.21: Comparativa tomando el mejor vértice de $R_i \in \mathcal{S}$ frente a $GRAD$

Sin embargo hasta ahora solamente hemos presentado ejemplos que certifican la certeza de la Conjetura 7.8.3. Necesitamos realizar un estudio más amplio que garantice que el punto de máxima iluminación y que soluciona por tanto el problema MaxA-p-Pv1(P), se encuentra sobre un vértice de una región $R_i \in \mathcal{S}$. Para ello, se ha aplicado esta técnica sobre un conjunto de polígonos con un número de vértices entre 5 y 100 y múltiplo de 5, generando 50 polígonos para cada número de vértices. El resultado obtenido ha sido de nuevo comparado con el resultado que se obtendría si aplicásemos a estos mismo polígonos la heurística $GRAD$.

En la Figura 7.24 mostramos la gráfica de las medias de porcentaje obtenida mediante estas dos técnicas. La curva superior representa los porcentaje obtenidos tomando el mejor vértice

de las regiones $R_i \in \mathcal{S}$ y la inferior los obtenidos mediante $GRAD$. Para poder comparar con más claridad ambas gráficas presentamos en la Figura 7.25 cuatro ampliaciones de esta gráfica, tomando las porciones de gráficas correspondientes a polígonos con un número de vértices entre 5 y 25 en la figura (a), entre 25 y 50 en la figura (b), entre 50 y 75 en la figura (c) y entre 75 y 100 en la figura (d). Como se puede observar el porcentaje mejora en todos los casos tomando el mejor vértice de las regiones $R_i \in \mathcal{S}$, aunque prácticamente resultan idénticos sobre todo para polígonos con menos de 50 vértices.

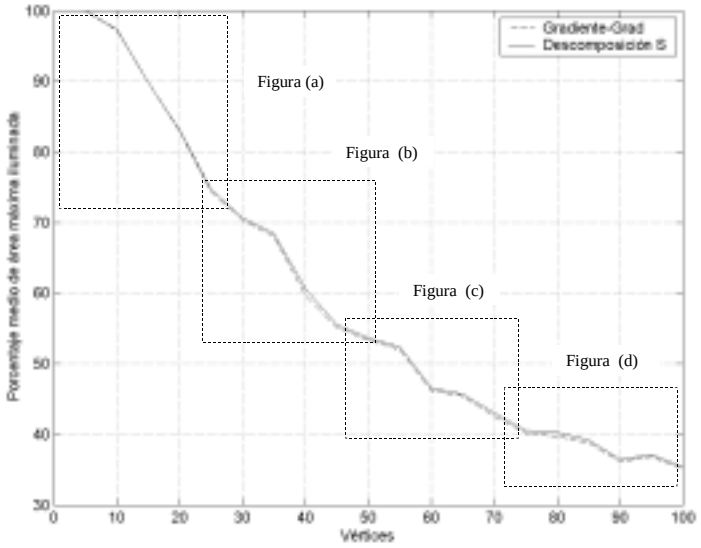


Figura 7.24: Resultados obtenidos al tomar el mejor vértice de la descomposición $\mathcal{S}$ frente a Grad

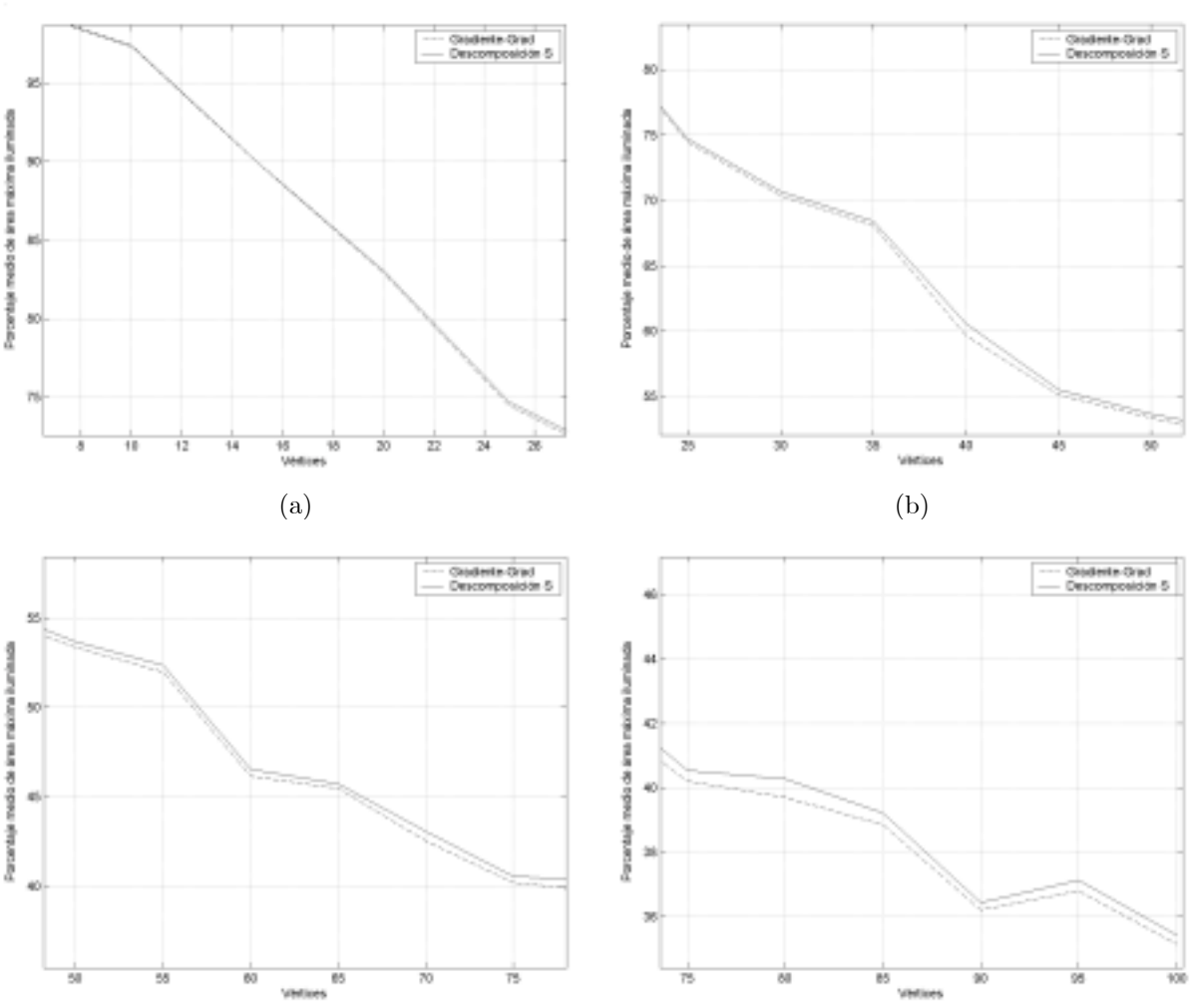
En consecuencia tenemos un conjunto de polígonos, (los de la Tabla 7.21), en los que el área iluminada aumenta tomando el vértice que mayor área ilumina de las regiones $R_i \in \mathcal{S}$, respecto a la técnica heurística que mejores resultados proporcionaba para el problema MaxA-p-Pv1(P), como era *gradiente-GRAD*. Además en media el porcentaje de área iluminada en el primer caso también es mayor que el obtenido por $GRAD$. Podemos deducir por tanto, (siempre de forma experimental), que dado un polígono P de n vértices, el punto que soluciona el problema MaxA-p-Pv1(P) se encuentra en un vértice de una región R_i perteneciente a la descomposición $\mathcal{S}$ de P, en *regiones de visibilidad* descrita anteriormente.

Por tanto, hemos presentado una prueba experimental de la Conjetura 7.8.3, que nos permitirá como veremos a continuación diseñar un algoritmo exacto polinomial para solucionar el problema MaxA-p-Pv1(P). Este algoritmo calculará el conjunto de *regiones de visibilidad* $\mathcal{S} = \{R_1, R_2, ..., R_m\}$ y los *polígonos de visibilidad* de los vértices de las regiones de $\mathcal{S}$, tomando como solución aquel vértice que ilumine mayor área, es decir, aquel vértice cuyo *polígono de visibilidad* dentro de P tenga mayor área. Jugarán un papel esencial el número de vértices que puedan llegar a tener las regiones de la descomposición $\mathcal{S}$, tanto para el diseño de un algoritmo

que solucione el problema MaxA-p-Pv1(P), como para el estudio de la complejidad del mismo. En el siguiente lema demostramos que este número es lineal.

Lema 7.8.4 *Si P es un polígono con n vértices y $\mathcal{S} = \{R_1, R_2, ..., R_m\}$ su descomposición en regiones de visibilidad, entonces el número de vértices de toda región de visibilidad R_i $\forall i = 1, ..., n$ es $O(n)$ y hay regiones con $\Theta(n)$ vértices.*

Demostración. Según la construcción de $\mathcal{S}$, cada vértice de P puede añadir a toda región de visibilidad dos lados como máximo, con lo cual el número de vértices es $O(n)$. Además todo polígono convexo P tiene una única región de visibilidad R, que coincide con él mismo y con su núcleo. Por tanto tendrá exactamente n vértices. ■



(a) (b)

(c) (d)

Figura 7.25: Ampliaciones de la Figura 7.24

Así, el algoritmo para solucionar el problema MaxA-p-Pv1(P), en forma de pseudocódigo lo podemos describir de la siguiente manera.

Algoritmo MaxA-p-Pv1(P)

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$..
SALIDA: Una *luz-punto* de máxima iluminación en P.

```
[01]  Construir el conjunto de regiones de visibilidad S = {R_1, R_2, ..., R_m};
[04]  q ←NULL;
[05]  area ← 0;
[06]  for (R_i ∈ S)
[07]   for (v_i ∈ R_i)
[08]    {Calcular el polígono de visibilidad de v_i en P,V(v_i);
[09]     if (area < Area(V(P, v_i))
[10]       area ← Area(V(P, v_i));q ← v_i;
[11]       q ← v_i;
[12]    }
[13]    return q;
```

Este algoritmo determina que el problema MaxA-p-Pv1(P) es un problema que se puede resolver en tiempo polinomial como se prueba a continuación.

Teorema 7.8.5 *Si P es un polígono con n vértices, el problema* MaxA-p-Pv1(P) *se puede resolver en tiempo polinomial $O(n^3)$.*

Demostración. Según se mencionó en el Lema 7.8.2 el conjunto de segmentos $W = bd(P) \cup \bigcup_{i=1}^{n} W_i$ tiene cardinal $O(n)$. Por otra parte, según Chazelle y Edelsbrunner [20], se puede buscar la subdivisión inducida por un conjunto de segmentos en posición general en $O(g \log g + i)$, donde g es el número de segmentos e i el número de intersecciones, lo que nos lleva a una complejidad $O(n \log n + n^2) \approx O(n^2)$.

Por otra parte existen $O(n^2)$ regiones con $O(n)$ vértices según el Lema 7.8.4. Para cada uno de estos vértices debemos calcular su *polígono de visibilidad* con un coste $O(n)$ [72]. Además el número de intersecciones de dos segmentos que generen $\mathcal{S}$ es a lo sumo uno, con lo cual el número máximo de vértices es $O(n^2)$. Por tanto el tiempo final será $O(n^3)$, lo que justifica que el problema MaxA-p-Pv1(P) se puede solucionar en tiempo polinomial. ■

Es importante destacar que si se utilizase la descomposición $\mathcal{T}$, el número de regiones de visibilidad sería $O(n^3)$ ya que en este caso a cada vértice cóncavos se le asocia un número lineal de regiones [15]. En este caso obtendríamos una complejidad también polinomial pero de un grado mayor $O(n^4)$.

Mostramos en la Figura 7.26 un ejemplo de la aplicación de este algoritmo basado en la descomposición $\mathcal{S}$. En la figura (a) mostramos el polígono P y su descomposición en regiones de visibilidad; en la figura (b) aparece además el vértice de esta descomposición de máxima

iluminación. Como podemos observar la solución es prácticamente la misma que las soluciones aportadas por las heurísticas *random search* (RS) y *gradiente* ($GRAD$), que se presentan en las figuras (c) y (d) respectivamente.

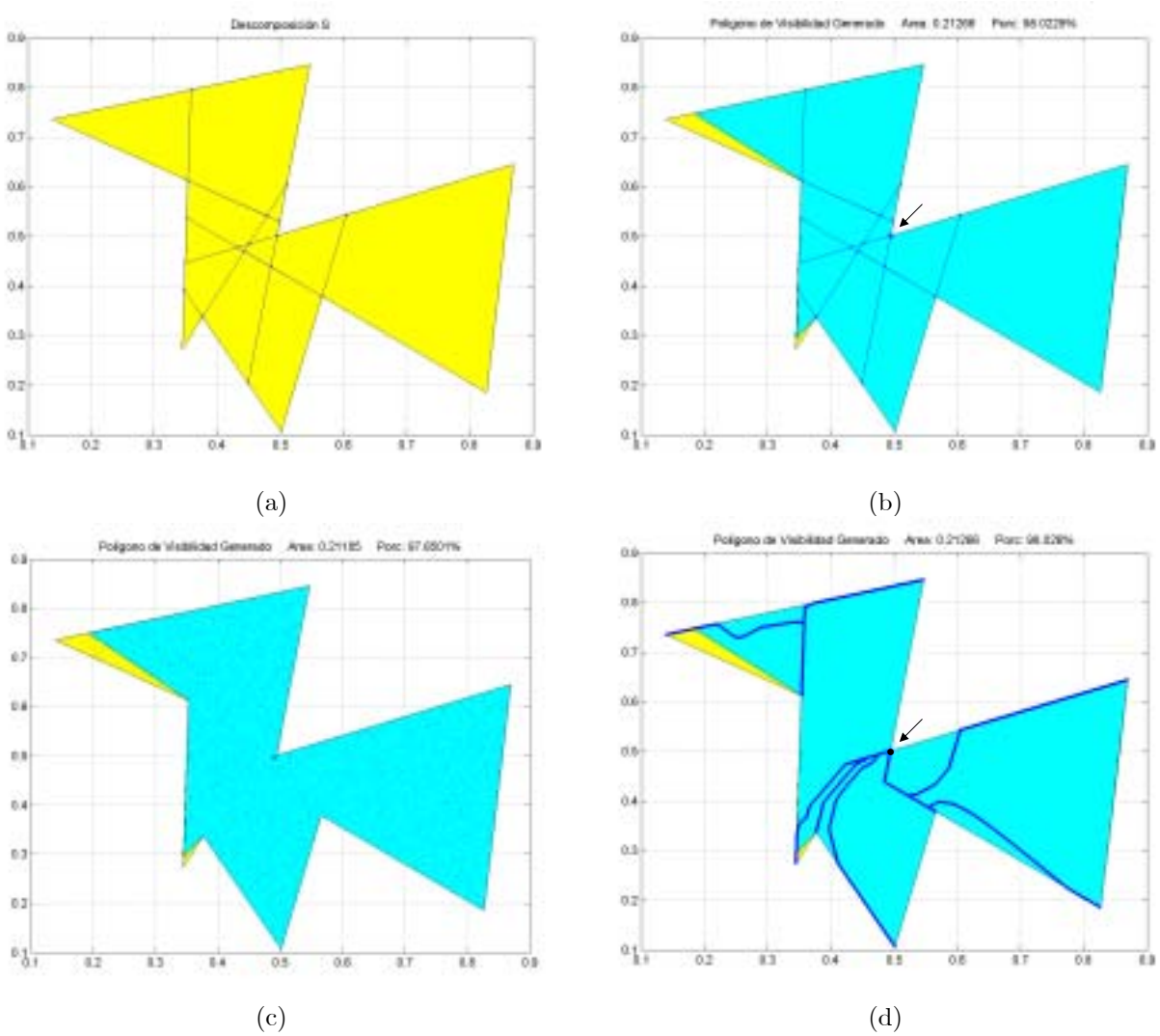


Figura 7.26: Un ejemplo de la aproximación del problema MaxA-p-Pv1(P) con la descomposición $\mathcal{S}$ en regiones de visibilidad

```
ALGORITMO DESCOMPOSICiÓN S
 * Polígono de visibilidad calculado
 - Punto de área máxima........ (0.494148,0.501424)
 - Área máxima.................. 0.212658 98.0229%
```

7.9 Conclusiones y trabajos futuros

Presentamos en este capítulo cuatro heurísticas para solucionar el problema de la búsqueda del punto de máxima iluminación interior a un polígono P de n vértices: *simulated-annealing SA, algoritmos genéticos GA, random search RS y método del gradiente GRAD.* En las Secciones 7.2, 7.3, 7.4 y 7.5 se han expuesto cada uno de los métodos y en la Sección 7.6 se han analizado los parámetros de cada una de las técnicas obteniendo resultados comparativos tanto en porcentaje de área iluminada, como en tiempo de respuesta de los métodos, concluyendo que $GRAD$ obtiene los mejores resultados medios. Sin embargo queda pendiente para futuras investigaciones un análisis pormenorizado de algunos parámetros que influyen sobre GA, como pueden ser la probabilidades de cruce y mutación.

En la Sección 7.7 se ha realizado un estudio experimental para encontrar relaciones matemáticas entre el número de vértices n de P y el porcentaje de área iluminada, concluyendo que esta relación sigue un decrecimiento exponencial de la forma $\alpha e^{-\beta n}$. Finalmente en la Sección 7.8 se ha analizado experimentalmente un algoritmo de complejidad polinomial para solucionar el problema MaxA-p-Pv1(P), utilizando para ello la descomposición de P en regiones de visibilidad y quedando pendiente la búsqueda de una desmostración exacta de la Conjetura 7.8.3.

A modo de resumen se presentan en la siguiente tabla los métodos heurísticos y los estudios realizados en relación con el problema MaxA-p-Pv1(P). Presentamos también los parámetros analizados para cada heurística.

Problema	*Solución / Complejidad*	*Parámetros*
MaxA-p-Pv1(P)	Aprox. *Simulated Annealing* $-$ *SA*	*Casos* 1 *a* 9 $-$ (Comparativa con 4) *Caso* 4: $T_0 = n$ $T_i = \frac{T_0}{1+i}(FSA)$ $T_f \leq 0.005$
	Aprox. *Algoritmos Genéticos* $-$ *GA*	*Selec.: proporcional / Cruce:* 1 *punto* $p_c = 0.8$ $p_m = 0.05$
	Aprox. *Random Search* $-$ *RS*	$m = 50n$
	Aprox. *Gradiente* $-$ *GRAD*	$v = 16$ $\beta = 0.001$
	$\approx O(n^3)$	–
PorA-p-Pv1(P)	$\alpha e^{-\beta n}$	–

Tabla 7.22: Técnicas Diseñadas para el problema Max-P-Pv1(P)

Según se indicó en el Capítulo 3 una restricción importante que se le puede imponer a la iluminación es la *limitación en el alcance*. Un estudio interesante que se podría realizar en futuras investigaciones es el comportamiento de estas métodos heurísticos cuando a la iluminación se le impone esta restricción. Asimismo, sería interesante realizar un estudio similar tomando el concepto de *buena* $t-$*iluminación* analizado en el Capítulo 4.

Capítulo 8

Minimización del número de luces que iluminan un polígono

Los métodos heurísticos presentados en el capítulo anterior para el problema MaxA-p-Pv1(P) y que buscan el punto interior a un polígono P de máxima iluminación, pueden ser generalizados para buscar el conjunto de k puntos interiores a P cuyo área conjunta iluminada sea máxima. Este último problema se ha denotado en el Capítulo 6 como MaxA-p-Pvk(P, k). Presentamos en este capítulo tres técnicas heurísticas para solucionar este problema: métodos basados en *simulated annealing* (SA), técnicas basadas en *algoritmos genéticos* (GA) y por último algoritmos de búsqueda aleatoria o *random search* (RS). Evidentemente la función objetivo utilizada en todas las técnicas del capítulo anterior cambia sustancialmente ahora, ya que necesitaremos calcular el área iluminada por k *luces*. Dada la necesidad de implementación de esta función objetivo, hemos utilizado para el cálculo de la unión de *polígonos de visibilidad* una pequeña modificación del algoritmo de recorte de *Weiler-Atherton* [56], que describimos también en este capítulo.

Finalmente presentamos técnicas de búsqueda binaria y secuencial para solucionar el problema $\mathcal{NP}$-duro que motivó esta parte de la memoria y que es el problema MinN-p-Pvk(P), que minimizará el número de *luces* que iluminan un polígono P de n vértices. Con todas las técnicas expuestas, (cuyas implementaciones se encuentran también en el Apéndice B), se han realizado estudios experimentales utilizando nuestro generador aleatorio de polígono RPG, que nos permiten obtener conclusiones tanto para el estudio comparativo de las técnicas heurísticas que solucionan MaxA-p-Pvk(P, k), como para averiguar si en media el número mínimo de *luces* que iluminan un polígono P de n vértices es inferior a $\lfloor n/3 \rfloor$.

8.1 Introducción

Para todo problema de naturaleza $\mathcal{NP}$-dura tiene sentido el estudio de técnicas heurísticas o aproximadas que le den respuesta ya que no es posible encontrar un algoritmo determinista

polinomial que lo solucione. Según demostraron Lee y Lin [71] el problema MinN-p-Pvk(P) que busca el mínimo número de luces interiores a un polígono P de n vértices para iluminarlo completamante, es de esta naturaleza $\mathcal{NP}$-dura. Formalmente este problema lo hemos enunciado de la siguiente manera:

minimización del numero de luces punto que iluminan un polígono
MinN-p-Pvk(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cuál es el número mínimo k de *luces punto* necesarias para iluminar el polígono P?

Para solucionar este problema es necesario previamente buscar para todo número entero k y todo polígono P, donde debo colocar k luces interiores a P, de forma que el área iluminada sea máxima. Este problema tiene formalmente el siguiente enunciado:

búsqueda de los k puntos de máxima iluminación interiores a un polígono
MaxA-p-Pvk(P, k):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde se deben colocar k *luces punto* en el interior del polígono P para que el área iluminada por dichas luces sea máxima?

Abordamos este problema mediante tres técnicas heurísticas que son una generalización de las diseñadas en el capítulo anterior para el problema MaxA-p-Pv1(P):

- *Simulated Annealing-SA*: En la Sección 8.3 presentamos las adaptaciones de MaxA-p-Pvk(P, k) a SA. Igual que se estudio en el capítulo anterior para el problema MaxA-p-Pv1(P) necesitamos determinar los siguientes elementos: conjunto de configuraciones S, vecindad, estrategias de templado y temperatura inicial. La mayor modificación está relacionada con el conjunto S de configuraciones y la vecindad de cada una de ellas. Hemos elegido en este caso la mejor combinación de los parámetros *temperatura inicial* y *disminución de la temperatura*, que se obtuvo en el análisis realizado en el capítulo anterior para el problema MaxA-p-Pv1(P). Esta combinación fue la analizada en el Caso 4 de la Sección 7.6.1, donde $T_0 = n$ y $T_k = \frac{T_0}{1+k}$ (FSA), siendo n el número de vértices del polígono P.

- *Algoritmos Genéticos-GA*: Como se mencionó en la Sección 7.3, las técnicas y operadores descritos allí para solucionar el problema MaxA-p-Pv1(P) son una particularización de los mecanismos que permitirán abordar el problema MaxA-p-Pvk(P, k) mediante estas técnicas genéticas. Presentamos en la Sección 8.4 los elementos que determinan el algoritmo genético: codificación, genoma, población, función objetivo y operadores de cruce y mutación. En la Sección 8.6 se analizan los resultados obtenidos y las comparativas con las otras dos heurísticas diseñadas: *simulated annealing-SA* y *random search-RS.* Las técnicas genéticas diseñadas para el problema MaxA-p-Pvk(P, k) deben considerarse técnicas generales que se adaptan a los distintos problemas estudiados en esta memoria. En este sentido, en el Capítulo 9 se expondrán las adaptaciones necesarias para abordar también

mediante estas técnicas genéticas el problema MaxA-p-Vor(N), que buscará el punto cuya *región de Voronoi* asociada tenga área máxima.

- *Random Search-RS*: Ampliando la técnica *random search-RS* diseñada en el capítulo anterior para solucionar el problema MaxA-p-Pv1(P) presentamos en la Sección 8.5 los mecanismos utilizados para solucionar el problema MaxA-p-Pvk(P, k) mediante una búsqueda aleatoria. Como se expondrá los resultados obtenidos en este caso son peores respecto RS en contraposición a los obtenidos por RS para el problema MaxA-p-Pv1(P).

La principal diferencia entre el estudio heurístico presentado para el problema MaxA-p-Pv1(P), (que busca el punto de máxima iluminación interior a un polígono P de n vértices), y el estudio que presentamos en este capítulo para el problema MaxA-p-Pvk(P, k), (que proporcionará el conjunto de k puntos o focos interiores al polígono P que maximice el área iluminada por al menos un foco), radica en la construcción de la *función objetivo* que utilizarán las técnicas aproximadas. En el primer caso, el conjunto de configuraciones o soluciones factibles eran todos los puntos interiores del polígono P, de tal forma que la *función objetivo* asociada a un punto t era igual al área de su *polígono de visibilidad* en el polígono P, $V(P,t)$. Sin embargo en el caso del problema MaxA-p-Pvk(P, k), el conjunto de soluciones factibles serán todos los posibles conjuntos $Q = \{t_1, ..., t_k\}$ con k puntos interiores al polígono P y por tanto, la función objetivo a evaluar en este caso, será igual al área del polígono *unión* de los *polígonos de visibilidad* de $t_i \in Q \ \forall i = 1, ..., k$, $(V(P, t_i))$.

El cálculo de la *unión* de k *polígonos de visibilidad* fue solucionado por Cheong y Oostrum [83], con un algoritmo de complejidad $O(((k(h+1)^2 + kn \log k) \log(k+n))$, siendo n el número de vértices y h el de agujeros del polígono P. Sin embargo, dada la necesidad de implementación del algoritmo, hemos utilizado una pequeña modificación del algoritmo de recorte de Weiler-Atherton [56] para la unión de *polígonos de visibilidad,* (que puede ser un polígono con agujeros). En la siguiente sección analizamos algunas características respecto a la *unión* de *polígonos de visibilidad* y exponemos el algoritmo utilizado para su implementación.

8.2 Unión de Polígonos de Visibilidad

Todo *polígono de visibilidad* es un polígono sin agujeros, pues una misma luz no puede iluminar la parte anterior y posterior de un agujero. Por tanto la *unión* de dos *polígonos de visibilidad* es un polígono sin agujeros. Por otra parte la *unión* de *polígonos de visibilidad* puede ser no conexa como mostramos en la siguiente proposición, cuya demostración resulta inmediata.

Proposición 8.2.1 *Sea P un polígono y dos puntos interiores a él t_1 y t_2, cuyos polígonos de visibilidad son $V(P,t_1)$ y $V(P,t_2)$ respectivamente. Entonces el polígono unión de $V(P,t_1)$ y $V(P,t_2)$ que denotaremos con $V(P,t_1,t_2)$ es conexo si y solo si la intersección es no vacía.*

Si un polígono P es un polígono con agujeros la unión de polígonos de visibilidad de puntos interiores a él, puede ser también un polígono con agujeros. Sin embargo, si unimos más de dos

polígonos de visibilidad de un polígono P sin agujeros, ¿seguirá siendo siempre un polígono sin agujeros? En la siguiente proposición demostramos que esta afirmación es falsa.

Proposición 8.2.2 *Sea un polígono P un polígono sin agujeros y k puntos interiores $t_1, t_2, ..., t_k$, cuyos polígonos de visibilidad son $V(P, t_1)$, $V(P, t_2)$,...,$V(P, t_k)$ respectivamente. El polígono unión $V(P, t_1, t_2, ..., t_k)$ puede ser un polígono con agujeros.*

Demostración. Sea el polígono de la Figura 8.1 y t_1, t_2, t_3 y t_4 cuatro puntos interiores al polígono cuyos *polígonos de visibilidad* son $V(P, t_1)$, $V(P, t_2)$, $V(P, t_3)$ y $V(P, t_4)$ respectivamente. Vemos cláramente que $V(P, t_1, t_2, t_3, t_4)$ es un polígono con agujeros. ■

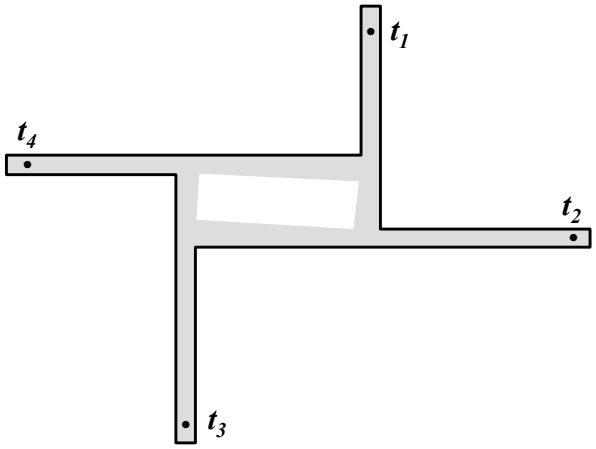


Figura 8.1: Unión de polígonos de visibilidad de un polígono

Llegados a este punto nos podemos preguntar si podemos encontrar un algoritmo para el cáclulo del polígono *unión* de *n polígonos de visiblidad*. En la siguiente subsección estudiamos el algoritmo de *Weiler-Atherton* para la unión de polígonos generales y describimos algunas pequeñas adaptaciones cuando se aplica sobre *polígonos de visibilidad* dadas las características especiales de éstos.

8.2.1 Algoritmo de Weiler-Atherton con Polígonos de Visibilidad

Existen algoritmos que permiten construir la unión de dos polígonos utilizando técnicas de recorte. En este sentido podemos mencionar los algoritmos de Liang-Barsky, Cyrus-Beck, Cohen-Sutherland, Nicholl-Lee-Nicholl y Weiler-Atherton, (ver [44, 56]). Todos ellos están relacionados con el recorte de segmentos y polígonos y permite construir el polígono *unión* de dos polígonos planos P_1 y P_2. Daremos una breve descripción del algoritmo de *Weiler-Atherton*, que nos permita determinar posibles adaptaciones de este algoritmo cuando los polígonos P_1 y P_2 son *polígonos de visibilidad* de dos puntos t_1 y t_2 interiores a un polígono P.

La idea básica en el algoritmo de recorte de *Weiler-Atherton* consiste en recorrer alternativamente el borde de cada polígono hasta encontrar un punto de intersección, construyendo como

veremos más adelante el polígono unión. Para simplificar la exposición denotaremos en lo que sigue a un polígono P sin agujeros de vértices $x_1, x_2, ..., x_n$ con $P(x_1, x_2, ..., x_n)$.

Según se ha mostrado en la Proposiciones 8.2.1 y 8.2.2 la *unión* de *polígonos de visibilidad* puede estar formada por varias componentes conexas, que a su vez pueden contener agujeros. En este sentido generalizaremos el concepto de polígono, diciendo que un polígono es un conjunto de componentes poligonales no conexas que pueden ser de dos tipos: componentes "principales" y componentes "agujeros", de tal forma que toda componente "agujero" esta contenida en una componente "principal". La diferencia esencial entre componentes "agujeros" y componentes "principales", impuesta por la aplicación del algoritmo, radica en que el recorrido de ambos tipos de polígonos es inverso, es decir, que las componentes principales se recorrerán en sentido positivo y los agujeros en sentido negativo.

Como todo *polígono de visibilidad* es un polígono sin agujeros y por tanto una componente "principal", para calcular la unión de k polígonos de visibilidad será suficiente ir uniendo cada polígono $V(P, t_i)$, con las componentes, (agujeros o principales), resultado de unir los *polígonos de visibilidad* de los puntos anteriores $V(P, t_1, ..., t_{i-1})$. Además como veremos más adelante la unión de dos componentes "principales" puede producir otras componentes "principales" o "agujeros", mientras que la unión de una componente "principal" con un "agujero" producirá un agujero recorte de éste, (si no está contenido en la componente "principal"), más la propia componente "principal". Exponemos en el Ejemplo 8.2.5 un caso particular para la unión de un polígono con un agujero, (una componente "principal" y una componente "agujero") y otro sin agujeros, (una única componente "principal"). Como se puede observar aparece una componente agujero nueva que queda orientada negativamente. En el Apéndice B se detallan las implementaciones y las estructuras de datos necesarias para la construcción de la *unión* de *polígonos de visibilidad.*

Descripción del algoritmo de Weiler-Atherton

Inicialmente el algoritmo parte de un vértice exterior de uno de los polígonos, avanzando en dicho polígono hasta encontrar un punto de intersección, para saltar en ese momento a un vértice del segundo polígono. Podemos describir el algoritmo con los siguientes pasos:

ENTRADA: Dos polígonos $P_1\,(a_1, a_2, .., a_t)$ y $P_2\,(b_1, b_2, .., b_t)$.
SALIDA: El polígono unión $P_1\,(a_1, a_2, .., a_t) \cup P_2\,(b_1, b_2, .., b_t)$.

- **Preparación**:

  1. Determinar los puntos de intersección de P_1 y P_2.
  2. Crear dos listas circulares L_1 y L_2 que se obtienen al recorrer los vértices de P_1 y P_2 respectivamente, e incluir los puntos de intersección encontrados entre cada par de vértices.

- **Cálculo**:

  1. Tomar el primer vértice a_1 de la lista L_1, exterior a P_2.

2. Seguir los nodos de la lista L_1 hasta encontrar un punto p_1 de intersección e incluir todos los nodos analizados, en la lista resultado, (que serán los vértices del polígono unión).
3. Buscar el punto p_1 en la lista L_2.
4. Seguir todos los nodos de la lista L_2 a partir de p_1, hasta encontrar un punto p_2 de intersección e incluir todos los nodos analizados en la lista resultado.
5. Volver a la lista L_1 y repetir el proceso hasta encontrar en el punto de partida a_1.

El algoritmo de *Weiler-Atherton* es sencillo en su implementación, pero teniendo en cuenta la naturaleza de los *polígonos de visibilidad*, es frecuente que se compartan vértices entre un par de polígonos que debemos unir. En este caso el algoritmo de *Weiler-Atherton* descrito necesitará añadir condiciones adicionales como se muestra en el siguiente ejemplo.

Ejemplo 8.2.3 *Sean los polígonos triangulares $P_1(a_1, a_2, a_3)$ y $P_2(b_1, b_2, b_3)$ de la Figura 8.2 y tal que $a_2 = b_2$. Evidentemente a_2 y b_2 son puntos de intersección.*

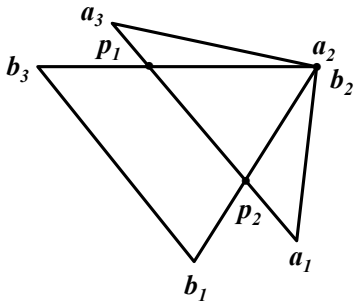


Figura 8.2: Un ejemplo del algoritmo de Weiler-Atherton

Si calculamos las listas circulares marcando los puntos con v para los vértices y c para los puntos de cortes tenemos:

- $L_1 = \{[a_1, v], [a_2, c], [a_3, v], [p_1, c], [p_2, c], [a_1, v]\}$
- $L_1 = \{[b_1, v], [p_2, c], [b_2, c], [p_1, c], [b_3, v], [b_1, v]\}$

Aplicando el algoritmo tendríamos el polígono suma determinado por los vértices:

$$P_1 \cup P_2 = \{a_1, a_2, p_1, p_2, b_2, p_1, p_2, b_2, ...\}$$

probando así que la construcción entra en bucle infinito, que no determina por consiguiente el polígono unión.

Por tanto, necesitamos añadir condiciones adicionales cuando utilizamos el algoritmo de *Weiler-Atherton* para unir *polígonos de visibilidad*. Analizamos éstas en el siguiente apartado.

Condiciones adicionales

Las dificultades del algoritmo de *Weiler-Atherton* aparecen cuando nos encontramos puntos de intersección que son vértices de alguno de nuestros polígonos $P_1\,(a_1, a_2, .., a_k)$ ó $P_2\,(b_1, b_2, .., b_t)$, como sucede en el caso $a_2 = b_2$. En este caso debemos distinguir si el siguiente vértice en el polígono en curso, (en nuestro caso a_3 de P_1), se encuentra en el interior o en el exterior del otro polígono, (en nuestro caso P_2). En el caso de que el siguiente vértice se encuentre exactamente sobre el borde la decisión será indistinta, como se analiza a continuación.

Supongamos que existe un punto de intersección que coincide con vértices de P_1 y P_2, sea $a_i = b_i$. Distinguimos tres casos que se representa en la Figura 8.3, según la posición de los vértices a_{i+1} y b_{i+1} de P_1 y P_2 respectivamente.

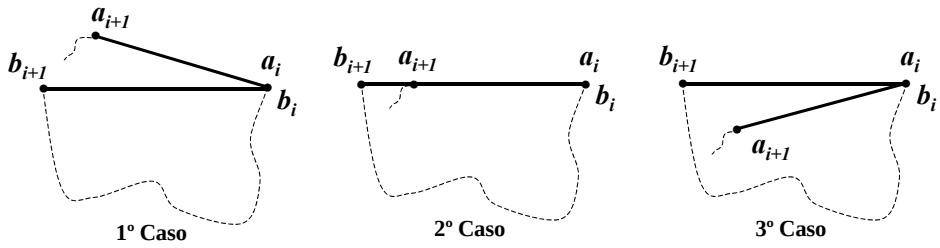


Figura 8.3: Puntos de Intersección sobre los vértices

Analizando cada uno de los casos se verifica que$\left(\overline{a_i a_{i+1}} \times \overline{b_i b_{i+1}}\right) > 0$ en el primer caso, $\left(\overline{a_i a_{i+1}} \times \overline{b_i b_{i+1}}\right) = 0$ en el segundo caso y $\left(\overline{a_i a_{i+1}} \times \overline{b_i b_{i+1}}\right) < 0$ en el útimo caso. Por tanto será necesario modificar el algoritmo de *Weiler-Atherton* cuando se aplica a *polígonos de visibilidad*, añadiendo una función de chequeo en el punto 3 del cálculo. Además si partimos de un punto vértice y no tenemos en cuenta si han sido visitados todos los puntos de corte, pueden quedar algunas componentes, (agujeros o principales), sin analizar, como se puede apreciar en el Ejemplo 8.2.5 si partimos del vértice a. Así, podríamos describir dicho cálculo para *polígonos de visibilidad* de la siguiente manera:

- **Cálculo:** Sea $C = \{p_1, ..., p_s\}$ la lista de todos los puntos de corte.

  1. Tomar el primer punto de corte p_1, eliminarlo de C y buscarlo en la lista L_1.
  2. Seguir los nodos de la lista L_1 hasta encontrar un punto p_2 de intersección, eliminar p_2 de C e incluir todos los nodos analizados, en la lista resultado, (que serán los vértices del polígono unión). Sea a_i el punto siguiente a p_2 en L_1.
  3. Buscar el punto p_2 en la lista L_2. Si llamamos b_i el punto siguiente a p_2 en L_2, pasar a 4 si $\left(\overline{p_2 a_i} \times \overline{p_2 b_i}\right) <= 0$. En caso contrario pasar a 5.
  4. Seguir todos los nodos de la lista L_2 a partir de p_2, hasta encontrar un punto p_3 de intersección e incluir todos los nodos analizados en la lista resultado.
  5. Volver a la lista L_1 y repetir el proceso hasta encontrar en el punto de partida p_1. En este caso se ha encontrado una componente conexa que puede ser "agujero", (si está orientada negativamente), o "principal", (si está orientada positivamente).

6. Si $C \neq \emptyset$ volver a 1.

Nota 8.2.4 *Evidentemente la modificación expuesta para el Algoritmo de Weiler-Atheton es aplicable también cuando el punto de intersección sea sólamente vértice de uno de los polígonos y no de los dos.*

Además las ampliaciones realizadas sobre el algoritmo de *Weiler-Atherton* no reducen la complejidad en el cálculo de los puntos de intersección de una arista $\overline{a_i a_{i+1}}$ con un polígono $P(x_1, x_2, ..., x_n)$, (que será $O(n^2)$), al tener que contrastar dicha arista con todos los vértice de P. La cota $O(n^2)$ se puede reducir aplicando una técnica de barrido, (ver [10]), para calcular las x intersecciones de n segmentos y que permite obtener una complejidad $O((n+x)\log n)$.

Presentamos a continuación un ejemplo para la unión de un polígono P_1 formado por una componente "principal" A y un "agujero" B, con otro polígono P_2 formado por una única componente "principal" C, (debe recordarse que todo *polígono de visibilidad* está formado por una única componente "principal", por tanto, el mecanismo seguido en este ejemplo es el seguido para unir el *polígono de visibilidad* de un punto t_i interior al polígono P, $V(P, t_i)$, con el conjunto de componentes, (agujeros y principales), resultantes de unir los *polígonos de visibilidad* de todos los puntos $t_1, ..., t_{i-1}$, $V(P, t_1, ..., t_{i-1})$.

Ejemplo 8.2.5 *Consideremos dos polígonos P_1 y P_2, tal que P_1 está formado por una componente "principal" A de vértices $A = \{a_1, a_2, a_3, a_4\}$ orientada en sentido positivo y una componente agujero B de vértices $B = \{b_1, b_2, b_3, b_4\}$ orientada en sentido negativo y P_2 está generada por una única componente principal C de vértices $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ orientada positivamente. Sean l_1, l_2, l_3, l_4, l_5 y l_6 los puntos de intersección de P_1 y P_2.*

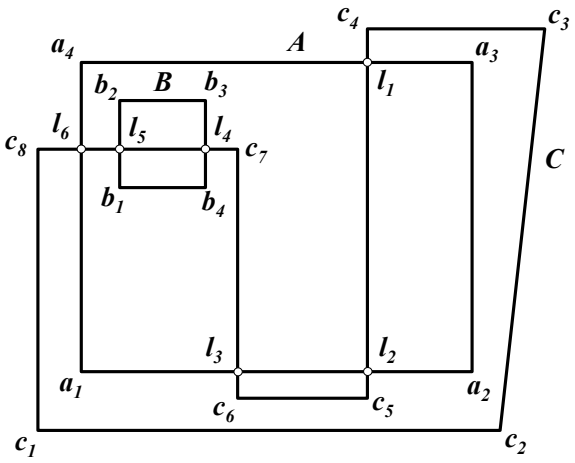


Figura 8.4: Un ejemplo de unión de polígonos con agujeros

Para calcular el polígono unión de P_1 y P_2 se debe calcular la unión de C, (que es la única componente de P_2), con cada componente de P_1.

1. *Calculando las listas circulares correspondientes y aplicando el algoritmo, se puede apreciar en la Figura 8.5, como la unión de las componentes principales A de P_1 y C de P_2, produce una nueva componente princial D de $P_1 \cup P_2$ con vértices $D = \{l_1, a_4, l_6, c_8, c_1, c_2, c_3, c_4\}$ y una componente agujero E, también de $P_1 \cup P_2$, de vértices $E = \{l_2, c_5, c_6, l_3\}$.*

2. *Por otra parte, la unión de la componente agujero B de P_1 y la componente principal C de P_2, produce una componente agujero F de $P_1 \cup P_2$, que es un recorte de B de vértice $F = \{l_4, l_5, b_2, b_3\}$.*

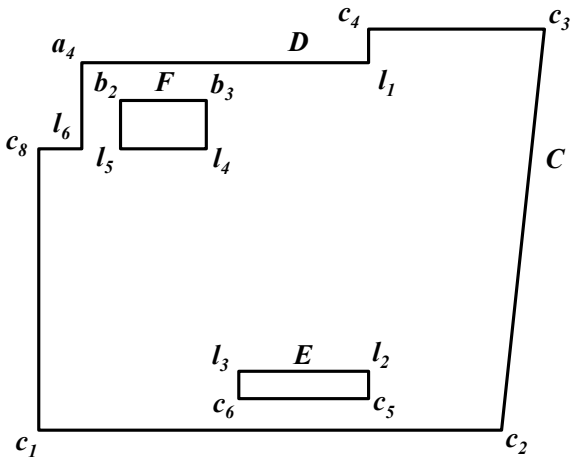


Figura 8.5: Resultado de la unión de los polígonos de la Figura 8.4

A modo de ejemplo presentamos en la Figura 8.6 el polígono producido por la implementación realizada en esta memoria del algoritmo de *Weiler-Atherton* para la *unión* de *polígonos de visibilidad*. En la figura (a) presentamos el polígono inicial; en la figura (b) visualizamos también un conjunto luces interiores al polígono; en la figura (c) presentamos el polígono resultante de unir todos los *polígonos de visibilidad* correspondientes a las luces interiores y por último en la figura (d) mostramos todos los elementos conjuntamente. Como se puede apreciar la *unión* de *polígonos de visibilidad* puede ser un polígono formado por varias componentes conexas con agujeros. Para el cálculo del área de este tipo de polígonos se han triangulado todos las componentes del polígono, (agujeros y principales), utilizando un algoritmo Scan de Graham para triangulación de polígonos simples [68], se ha calculado el área de todas las componentes, de tal forma que sumando el área de todas las componentes "principales" y restando el área de las componentes "agujeros" tenemos el área del polígono *unión*. Para comparar el porcentaje de área iluminado se ha repetido también el mismo proceso de triangulación sobre el polígono inicial P, como se mencionó en el capítulo anterior. Los códigos para la construcción de la *unión* de *polígonos de visibilidad* así como las programas para las visualización con el paquete matemáticos Matlab se puede encontrar en el Apéndice B de listados.

Una vez descrito el mecanismo para unir *polígonos de visibilidad* utilizado, podemos contruir ya la función de costes o función objetivo de nuestros métodos aproximados. Exponemos a continuación los elementos diseñados en las heurísticas que solucionan el problema MaxA-p-Pvk(P, k).

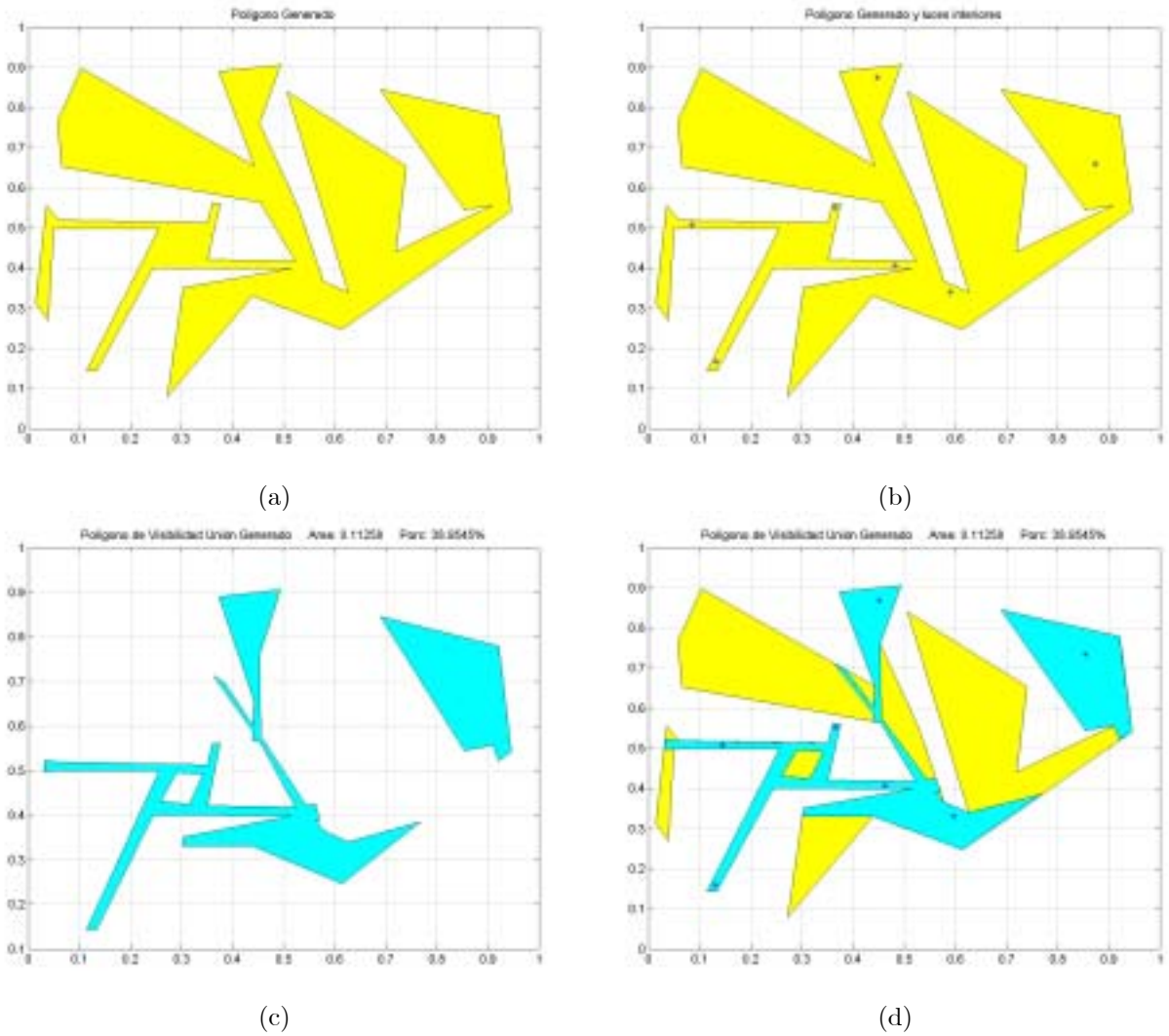


Figura 8.6: La unión de polígonos de visibilidad puede ser un polígono no conexo con agujeros.

8.3 El problema MaxA-p-Pvk(P,k) con simulated annealing-SA

La utilización de *simulated annealing SA* para solucionar de forma aproximada el problema MaxA-p-Pvk(P,k) comparte mucho elementos de configuración con los utilizados por esta misma técnica para abordar el problema MaxA-p-Pv1(P), adaptados evidentemente ahora a la búsqueda del conjunto, (y no de un solo punto), de k puntos cuyo área conjunta iluminada sea máxima. Por ello haremos un análisis más detallado en los elementos diferenciadores como son fundamentalmente los relacionados con la *funcion objetivo* o *función de costes*.

Recordemos el enunciado concreto del problema:

búsqueda de los k puntos de máxima iluminación interiores a un polígono MaxA-p-Pvk(P, k):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde debo colocar k *luces punto* en el interior del polígono P para que el área iluminada por dichas luces sea máxima?

Por tanto, la entrada del problema serán los n vértices, (dados en sentido positivo), del polígono P y la salida serán las coordenadas de k puntos $\{(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)\}$ interiores a P. Igual que se hizo para el caso $k = 1$, mostramos en la Figura 8.7 un ejemplo de la aplicación de la heurística SA para el problema MaxA-p-Pvk(P, k), sobre un polígono P de 19 vértices y con $k = 2$, (es decir, se buscan dos puntos tal que el área conjunta iluminada sea máxima).

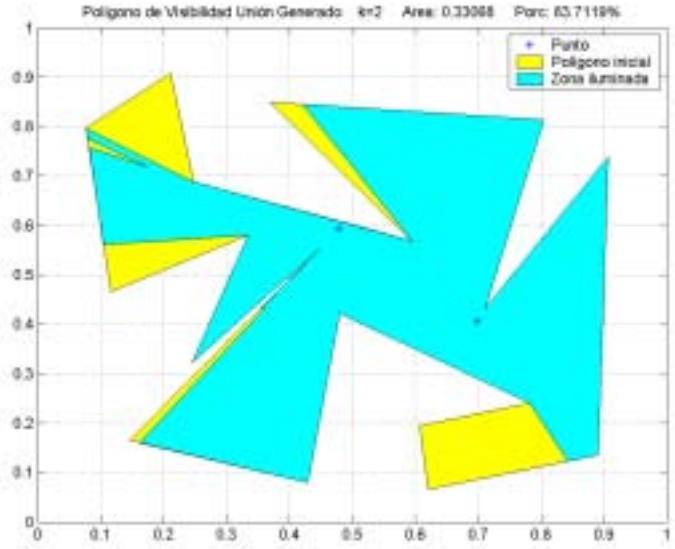


Figura 8.7: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con SA $k = 2$

Los elementos de la heurística adaptados a nuestro problema se pueden describir de la siguiente manera:

Conjunto S de configuraciones:

El conjunto de configuraciones factibles para el problema MaxA-p-Pvk(P, k) estará formado por todos los conjunto de k puntos interiores al polígono P. Cada punto vendrá determinado por sus coordenadas cartesianas (x, y). Así el conjunto S de configuraciones lo podemos representar:

$$S = \{S_1 = \{(x_1^1, y_1^1), ..., (x_k^1, y_k^1)\}, S_2 = \{(x_1^2, y_1^2), ..., (x_k^2, y_k^2)\}, ..., S_n = \{(x_1^n, y_1^n), ..., (x_k^n, y_k^n)\}, ...\}$$

Cada elemento de S representa una solución factible para nuestro problema, de tal forma que la solución S_i que proprocionará de forma aproximada la heurística será aquella tal que el área conjunta iluminada por los puntos que la forman sea máxima. Para calcular este área se utilizará

el algoritmo de *unión* de *polígonos de visibilidad* expuesto en el punto anterior y determinará el valor la función objetivo o función de costes para cada elemento de la configuración.

Función de coste C:

La función de costes, función objetivo o función fitness $C : S \rightarrow \mathbb{R}$ asignará a cada elemento de S un valor real. Así, dado un conjunto $S_i \in S$, de k puntos interiores al polígono P, $S_i = \{p_1^i = (x_1^i, y_1^i), ..., p_k^i = (x_k^i, y_k^i)\}$ la función de costes producirá un valor real que representa el área conjunta iluminada por todas las *luces-punto* de S_i. Si denotamos por $V(P, S_i)$ al polígono unión resultado de unir todos los polígonos de visibilidad de los puntos de S_i, $(V(P, p_1^i), ..., V(P, p_k^i))$, estará formado por un conjunto de t componentes "principales", cada una de las cuales denotaremos con CP_m^i $m = 1, .., t$ y un conjunto de l componentes "agujeros" que denotatemos con CA_h^i $h = 1, ..., l$. De esta forma $V(P, S_i)$ lo podemos expresar de la forma:

$$V(P, S_i) = \bigcup_{m=1}^{t} CP_m^i \setminus \bigcup_{h=1}^{l} CA_h^i \tag{8.3.1}$$

y la función de costes para $S_i \in S$ la definimos de la forma:

$$C(S_i) = \sum_{m=1}^{t} Area(CP_m^i) - \sum_{h=1}^{l} Area(CA_h^i) \tag{8.3.2}$$

recordando que para las pruebas experimentales de la heurística, el área de cada componente "principal" o "agujeros" se calcula mediante un mecanismo de triangulación de cada componente, ([68]).

La función de costes se debe aplicar a cada elemento S_i del conjunto de configuraciones. Además en cada iteración se debe obtener una solución factible $S_{i+1} \in S$ en función de la solución factible anterior $S_i \in S$, como se explica en la Sección 1.5.1. La función de vecindad utilizada en este caso para el problema MaxA-p-Pvk(P, k) es una ampliación de la utilizada para el problema MaxA-p-Pv1(P) en el capítulo anterior. En este caso a cada elemento de S_i, que es un punto $p_j^i = (x_j^i, y_j^i)$ se le asignará un vecino, sumando a cada coordenada un valor real que sigue la distribución $N(0, 1)$.

Vecindad en cada configuración

Según se indicó en el estudio de la heurística *simulated annealing SA*, para cada elemento $S_i \in S$, se debe obtener un elemento $S_{i+1} \in S$ en la vecindad de S_i y que será el elemento a analizar en la siguiente iteración de SA. En nuestro caso, dado $S_i = \{p_1^i = (x_1^i, y_1^i), ..., p_k^i = (x_k^i, y_k^i)\}$ la función de vecindad obtendrá cada elemento p_j^{i+1} $j = 1, ..., k$ de S_{i+1} sumando a cada coordenada de p_j^i $j = 1, ..., k$ un valor real que sigue la distribución $N(0, 1)$, es decir:

$$\begin{cases} x_j^{i+1} = x_j^i + N(0,1) & j = 1, ..., k \\ y_j^{i+1} = y_j^i + N(0,1) & j = 1, ..., k \end{cases} \tag{8.3.3}$$

Igual que se explicó para el caso $k = 1$ en el capítulo anterior, para obtener números aleatorios distribuidos uniformemente hemos utilizado la función `rand` de que dispone el lenguaje C y para

generar números aleatorios distribuidos normalmente se ha implementado el método Box-Muller [16], que permite obtener a partir de dos números u_1 y u_2 distribuidos uniformemente, otros dos números n_1 y n_2 normalmente distribuidos, mediante las fórmulas

$$\begin{cases} n_1 = \sqrt{-2 * \ln(u_1)} sen(2\ \pi\ u_2) \\ n_2 = \sqrt{-2 * \ln(u_1)} \cos(2\ \pi\ u_2) \end{cases} \tag{8.3.4}$$

Así si denotamos por `div` el *factor de vecindad* por el que se dividirán n_1 y n_2, como se explicó para el problema MaxA-p-Pv1(P) en la Sección 7.2, la función que genera el vecino S_{i+1} al elemento S_i, se puede expresar de la siguiente manera en forma de pseudocódigo:

Función Generar-Vecino

ENTRADA:

Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
Un elemento de S, $S_i = \{p_1^i = (x_1^i, y_1^i), ..., p_k^i = (x_k^i, y_k^i)\}$.
Un factor de vecindad `div`.

SALIDA:

Un conjunto de k puntos $S_{i+1} = \{p_1^{i+1} = (x_1^{i+1}, y_1^{i+1}), ..., p_k^{i+1} = (x_k^{i+1}, y_k^{i+1})\}$ interiores a P, que constituyen un vecino de S_i.

```
[01]  S_{i+1} ← ∅;
[02]  for(p_j^i = (x_j^i, y_j^i) ∈ S_i)
[03]    {u_1 ←rand()*1.0/RAND_MAX;
[04]     u_2 ←rand()*1.0/RAND_MAX;
[05]     do
[06]      {n_1 ←Sqrt(-2.0*ln(u_1))*sin(2 * π * u_2);
[07]       n_2 ←Sqrt(-2.0*ln(u_1))*cos(2 * π * u_2);
[08]       x_j^{i+1} ← x_j^i + n_1/div;
[09]       y_j^{i+1} ← y_j^i + n_2/div;
[10]      }while (p_j^{i+1} = (x_j^{i+1}, y_j^{i+1}) exterior a P);
[11]     S_{i+1} ← S_i ∪ {p_j^{i+1}};
[12]    }return S_{i+1};
```

La configuración inicial determina la situación de partida de *simulated annealing SA*. En nuestro caso dicha configuración estará formada por un conjunto de k puntos interiores al polígono P, ampliando así lo descrito para el problema MaxA-p-Pv1(P) en el Capítulo 7.

Configuración inicial:

La configuración inicial que necesita SA para aproximar el problema MaxA-p-Pvk(P, k) es un primer elemento S_0 del conjunto de configuraciones S, formado por un conjunto de k puntos

interiores al polígono P.

$$S_0 = \{p_1^0 = (x_1^0, y_1^0), ..., p_k^0 = (x_k^0, y_k^0)\}$$

Este primer conjunto de puntos se debe generar aleatoriamente siguiendo el mismo mecanismo que se utilizó para el caso $k = 1$. En forma de pseudocódigo, la función que generará esta primera configuración inicial se puede describir de la siguiente manera:

Función Generar-Configuracion Inicial

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Un conjunto $S_0 = \{p_1^0 = (x_1^0, y_1^0), ..., p_k^0 = (x_k^0, y_k^0)\}$ de k puntos interiores a P.

```
[01]  S_0 ← ∅;
[01]  for(i = 1 to i = k)
[01]    {do
[02]       {x_i^0 ←rand()*1.0/RAND_MAX;
[03]        y_i^0 ←rand()*1.0/RAND_MAX;
[04]       }while (p_i^0 = (x_i^0, y_i^0) exterior a P);
[09]     S_0 ← S_0 ∪ {p};
[09]    }return S_0;
```

Estrategias de templado y criterio de parada:

La heurística *simulated annealing SA* para la búsqueda de los k puntos interiores a un polígono P quedará totalmente determinada si conocemos además de la función de coste, la vecindad de cada elemento de la configuración y la generación de la configuración inicial, las estrategias de templado utilizadas: temperatura inicial y decrecimiento de la temperatura. En este sentido se realizó en el capítulo anterior un estudio pormenorizado de diferentes tipos de temperatura inicial y de disminución de esta, produciendo nueve Casos que fueron motivos de estudio. Para el problema MaxA-p-Pvk(P, k) que nos ocupa en este capítulo se ha utilizado la combinación de temperatura inicial y disminución de temperatura que mejores resultados obtuvieron para $k = 1$ en el capítulo anterior y que fueron los obtenidos en el Caso 4 : $T_0 = n$ para la temperatura inicial y $T(m) = \frac{T_0}{1+m}$ para la disminuación de la temperatura en cada iteración m. Asimismo el número de ejecuciones de la heurística para cada temperatura T será $N(T) = 1/T$ igual que en el caso $k = 1$ y el criterio de parada igualmente $T_f \leq 0.005$. De forma esquemática exponemos los criterios utilizados en la heurística:

- $T_0 = n$ (n número de vértices de P).
- $T_m = \frac{T_0}{1+m}$ (m número de iteración).
- $N(T) = \frac{1}{T}$.
- $T_f \leq 0.005$.

Presentamos en la Figura 8.8 un ejemplo del resultado obtenido por SA para un polígono con 19 vértices, $k = 2$, ejecutado con estos criterios.

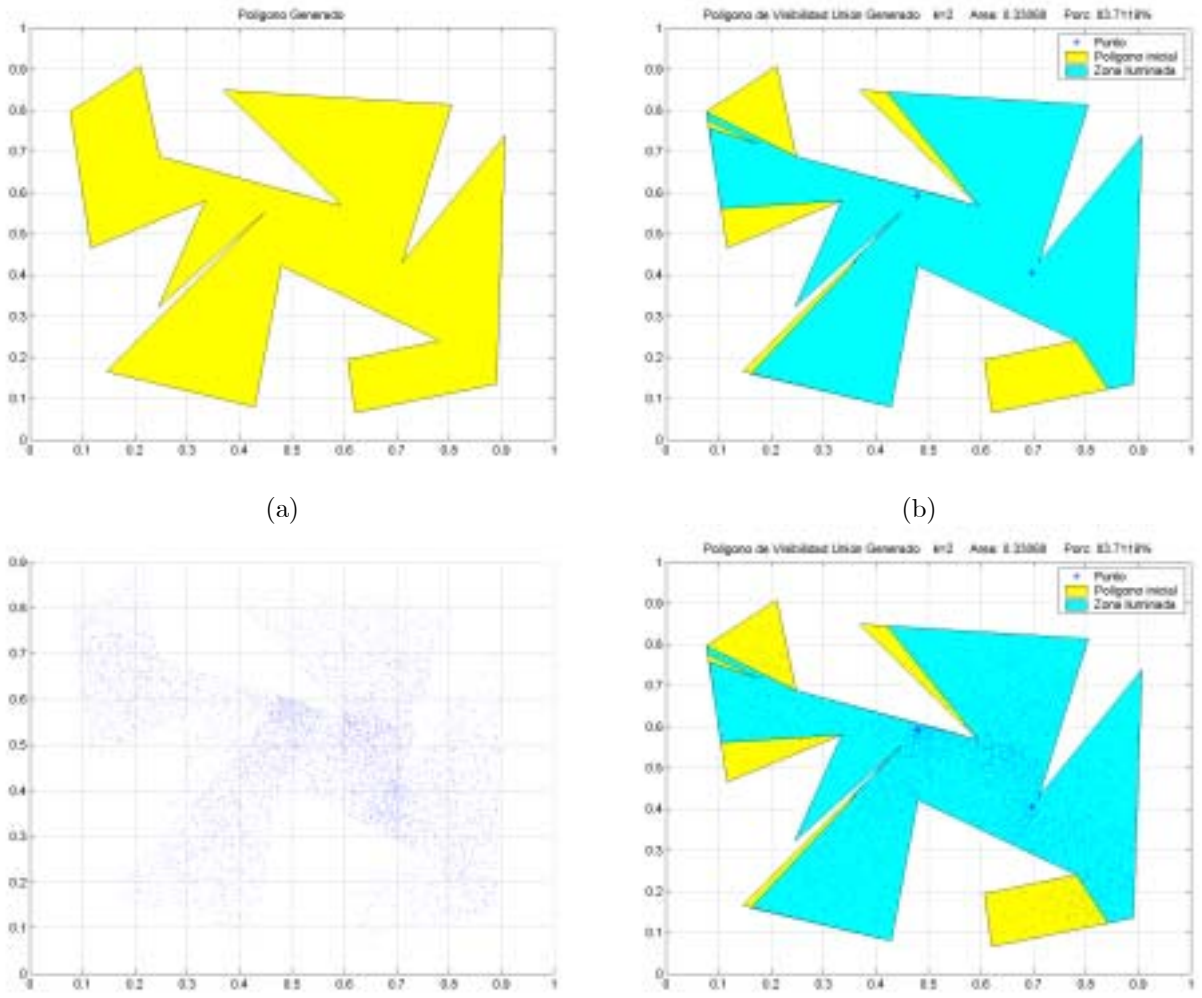


Figura 8.8: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con SA $k = 2$ $T_0 = n$ $T(k) = \frac{T_0}{1+k}$ $T_f = 0.005$ $N(T) = \frac{1}{T}$

```
ALGORITMO SIMULATED ANNEALING SA T0=19 K=2:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.494151,0.596570)
                                 p2=(0.690905,0.404886)
 - Área máxima.................. 0.330682 83.7119%
```

Además en la Figura 8.8 (c) se puede observar la nube utilizada por SA en la búsqueda y como se puede observar, la nube no es tan densa entorno a los puntos solución como ocurría en la Figura 7.2 para $k = 1$. Esta densidad de puntos es debida a que la búsqueda se realiza

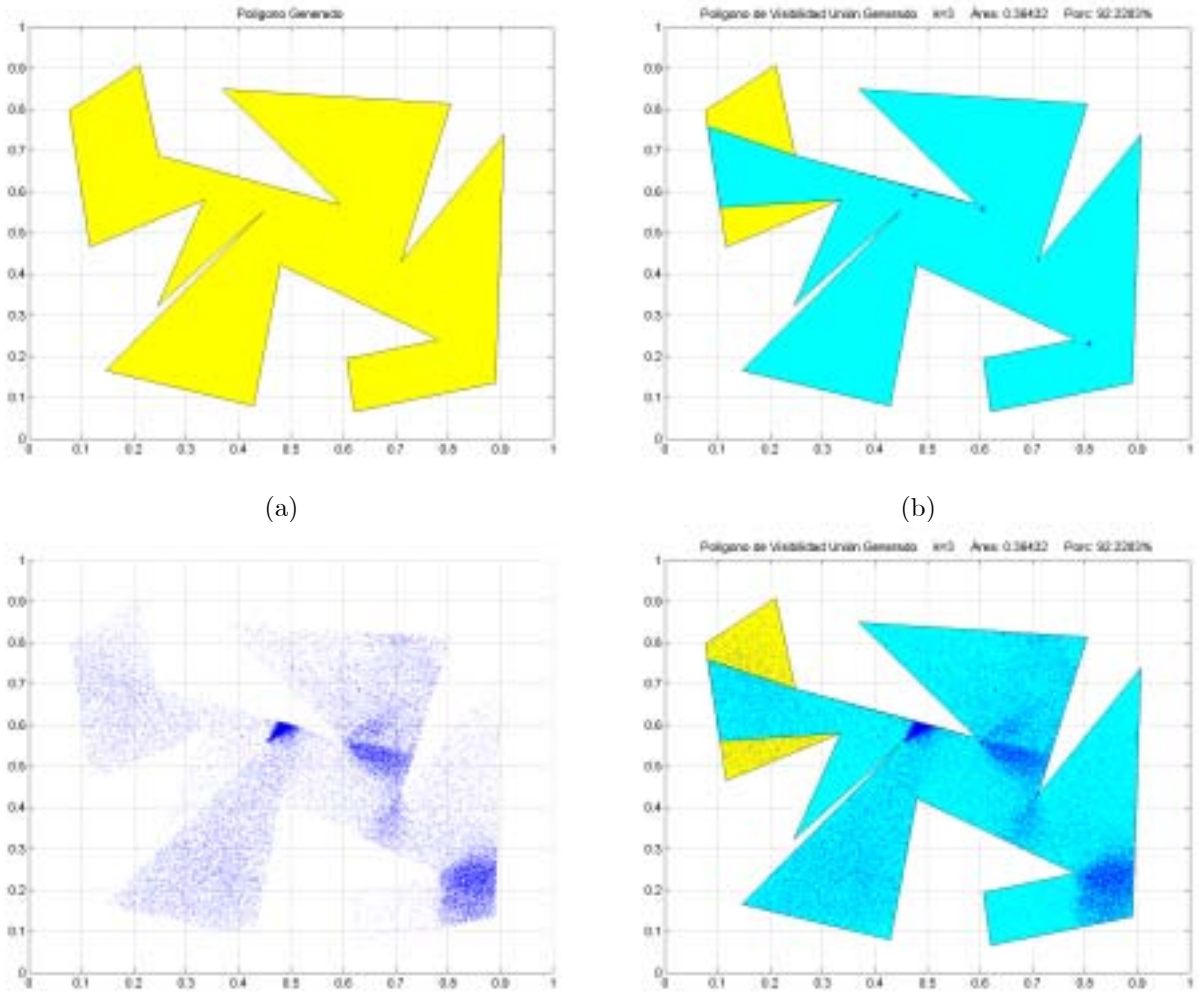


Figura 8.9: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con SA $k = 3$ $T_0 = n$ $T(k) = \frac{T_0}{1+k}$ $T_f = 0.00005$ $N(T) = \frac{1}{T}$

```
ALGORITMO SIMULATED ANNEALING SA T0=19 K=3:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.475441,0.592153)
                                 p2=(0.807935,0.230839)
                                 p3=(0.606423,0.557482)
 - Área máxima.................. 0.36432 92.2283%
```

conjuntamente para $k = 2$ y por tanto, se analizarán parejas de puntos con uno de los puntos muy alejado del óptimo, lo que produce que el otro punto también se desplace a posiciones más desfavorables y se genere una configuración más uniforme de los puntos interiores analizados. Si aplicamos SA al mismo polígono, pero con $k = 3$ y la condición de parada a $T_f \leq 0.00005$, obtenemos una nube de búsqueda como la que se muestra en la Figura 8.9 (c), en la que se puede observar como la nube generada determina también la *regiones de visibilidad* de la descomposición $\mathcal{S}$ estudiada en el capítulo anterior. Esta situación hace sospechar que la solución al problema MaxA-p-Pvk(P, k) también se va a encontrar entre los vértices de las *regiones de visibilidad* $R_i \in \mathcal{S}$. Además, tanto en la Figura 8.8, para $k = 2$, como en la Figura 8.9, para $k = 3$, se puede observar como las soluciones aportadas tienden hacia vértices de dicha descomposición, lo que será motivo de futuras investigaciones. Sin embargo, debe recordarse que teníamos $O(n^2)$ regiones $R_i \in \mathcal{S}$, con $O(n)$ vértices cada una de ellas, con lo cual tenemos $O(n^3)$ puntos para los cuales debemos estudiar el área iluminada por todo subconjunto de k puntos. Teniendo en cuenta esta situación y que el problema de minimización del número de *luces-punto* que iluminan un polígono P, (MinN-p-Pvk(P)), es $\mathcal{NP}$-duro, se justifica si cabe aún más la utilización de técnicas heurísticas para solucionar nuestro problema MaxA-p-Pvk(P, k) y por tanto el problema MinN-p-Pvk(P).

Pasamos a continuación a describir los elementos utilizados para la construcción de un algorítmo que solucione también el problema MaxA-p-Pvk(P, k), pero utilizando otra técnica heurística como son los *algoritmos genéticos*.

8.4 El problema MaxA-p-Pvk(P,k) con algoritmos genéticos-GA

Presentamos en esta sección las componentes de la heurística GA, diseñadas para solucionar el problema MaxA-p-Pvk(P, k) y que han sido utilizadas en la implementación del algoritmo correspondiente cuyo código se encuentra en el Apéndice B de listados.

Codificación, genoma y población

El problema MaxA-p-Pvk(P, k) consiste en la búsqueda de un conjunto de k luces interiores a un polígono P con n vértices, tal que el área conjunta iluminada por todas las luces sea máxima. Recordemos el enunciado formal del problema:

búsqueda de los k puntos de máxima iluminación interiores a un polígono
MaxA-p-Pvk(P, k):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Dónde debo colocar k *luces punto* en el interior del polígono P para que el área iluminada por dichas luces sea máxima?

De forma natural podemos ampliar la estructura de *individuo* $G = ((x, y))$, (formada por las coordenadas de un punto interior al polígono P, que se diseño para solucionar el problema MaxA-p-Pv1(P) en el capítulo anterior), a nuestro caso, en el que un individuo está codificado

por un genoma con k genes

$$G = ((x_1, y_1), (x_2, y_2), ..., (x_k, y_k)) \tag{8.4.5}$$

siendo k el número de luces buscadas.

El tamaño de población elegido está relacionado con la entrada del problema, es decir, con el número de vértices n del polígono P. Así, la población para la generación t de nuestra heurística vendrá representado por:

$$A(t) = \{G_1^t, G_2^t, ..., G_{kn}^t\} \tag{8.4.6}$$

donde k es el número de *luces-punto* buscadas y G_i^t representa a un individuo o genoma perteneciente a la población $A(t)$. Definimos en el siguiente apartado la *función objetivo* de nuestra heurítica, tanto para un *genoma* o *individuo* G, (que será equivalente a la *función de costes* definida para una configuración S en *simulated annealing* SA), como para una población $A(t)$.

Función objetivo

A cada individuo $G_i^t = ((x_1^{(i)}, y_1^{(i)}), (x_2^{(i)}, y_2^{(i)}), ..., (x_k^{(i)}, y_k^{(i)}))$ $i = 1, ..., n$ de una población $A(t)$, se le debe asociar un valor igual al área conjunta iluminada por todos los puntos $(x_j^{(i)}, y_j^{(i)})$ $j = 1, ..., k$, que lo componen. Si asociamos un individuo G_i^t con la definición de configuración factible S_i $i = 1, 2, ...$, realizada para la heurística *simulated annealing* SA, podemos definir la *función objetivo* para cada individuo G_i^t $i = 1, ..., n$ de una población $A(t)$ de la siguiente manera:

$$f(G_i^t) = \sum_{m=1}^{w} Area(CP_m^i) - \sum_{h=1}^{l} Area(CA_h^i) \tag{8.4.7}$$

donde CP_m^i $m = 1, ..., w$ y CA_h^i $h = 1, ..., l$ representan a las "componentes principales" y "componentes agujeros" respectivamente, resultado de unir, (en las implementaciones mediante el algoritmo de *Weiler-Atherton*), los *polígonos de visibilidad* en P de los puntos que componen G_i^t, $(V(P, (x_j^{(i)}, y_j^{(i)})$ $j = 1, ..., k)$. Este polígono unión lo podemos denotar por tanto

$$V(P, G_i^t) = \bigcup_{m=1}^{w} CP_m^i \setminus \bigcup_{h=1}^{l} CA_h^i \tag{8.4.8}$$

La función *fitness o función objetivo de una población* $A(t)$ se definirá como el valor máximo obtenido al aplicar la función objetivo a cada uno de los individuos que componen la población

$$F(A(t)) = \max_{i=1,...,kn} f(G_i^t) \tag{8.4.9}$$

Los operadores de selección, cruce y mutación elegidos son los expuestos en la Sección 7.3 para el problema MaxA-p-Pv1(P). Sin embargo, detallamos en el siguiente apartado alguno de sus elementos.

Selección, cruce y mutación

El operador de *selección* utilizado en la heurística *GA* ha sido el de *selección proporcional* implementado mediante el *método de la ruleta*, (ver Capítulo 1). Por tanto, los pasos seguidos en la implementación de este método son los mismo a los utilizados en la Sección 7.3 para solucionar el problema de la búsqueda del punto de máxima iluminación, (MaxA-p-Pv1(P)), y que repetimos a continuación:

(a) Determinar la suma S de las adaptaciones de toda la población.

(b) Relacionar uno a uno todos los individuos con segmentos contiguos de la recta real $[0, S)$, tales que cada segmento individual sea igual en su tamaño a su grado de adaptación.

(c) Generar un número aleatorio en $[0, S)$.

(d) Seleccionar el individuo cuyo segmento cubre el número aleatorio.

(e) Repetir el proceso hasta obtener el número deseado de muestras.

Así, si $\{G_1^t, ..., G_{kn}^t\}$ es la población en la generación t y $S = \sum_{i=1}^{kn} f(G_i^t)$, elegimos un número aleatorio r uniformemente distribuido entre 0 y S. Si $r \leq f(G_1^t)$ elegimos el individuo G_1^t, en otro caso elegimos el individuo G_i^t tal que $\sum_{j=1}^{i-1} f(G_j^t) < r \leq \sum_{j=1}^{i} f(G_j^t)$. Este mecanismo se repite dos veces para obtener los dos padres de nuestra generación t, G_i^t y G_j^t.

Una vez seleccionados los padres para la generación t, (G_i^t, G_j^t) pasamos a modificarlos mediante el operador de cruce como se mencionó en la Sección 1.5.2. Para ello generamos aleatoriamente un número r ($0 \leq r < 1$). Si r es mayor que la probabilidad de cruce p_c estos individuos pasarán sin ser modificados a la generación siguiente, en caso contrario serán sometidos al proceso de cruce. El mecanismo de cruce utilizado es el *cruce en un solo punto* descrito en la fórmula 1.5.19. Elegimos al azar un número aleatorio $1 \leq r \leq k$ e intercambiamos los genes de los padres a partir del gen que ocupa el lugar r. Así, si los genomas de los padres seleccionados vienen descritos por:

$$\left.\begin{array}{l} G_i^t = \left((x_1^{(i)}, y_1^{(i)}), ..., (x_r^{(i)}, y_r^{(i)}), (x_{r+1}^{(i)}, y_{r+1}^{(i)}), ..., (x_k^{(i)}, y_k^{(i)})\right) \\ G_j^t = \left((x_1^{(j)}, y_1^{(j)}), ..., (x_r^{(j)}, y_r^{(j)}), (x_{r+1}^{(j)}, y_{r+1}^{(j)}), ..., (x_k^{(j)}, y_k^{(j)})\right) \end{array}\right\} \tag{8.4.10}$$

tendremos tras el procedimiento de cruce los genomas

$$\left.\begin{array}{l} G_i^{*t} = \left((x_1^{(i)}, y_1^{(i)}), ..., (x_r^{(i)}, y_r^{(i)}), (x_{r+1}^{(j)}, y_{r+1}^{(j)}), ..., (x_k^{(j)}, y_k^{(j)})\right) \\ G_j^{*t} = \left((x_1^{(j)}, y_1^{(j)}), ..., (x_r^{(j)}, y_r^{(j)}), (x_{r+1}^{(i)}, y_{r+1}^{(i)}), ..., (x_k^{(i)}, y_k^{(i)})\right) \end{array}\right\} \tag{8.4.11}$$

Cruzados los padres G_i^t, G_j^t y obtenidos los hijos G_i^{*t}, G_j^{*t}, se les aplicará a éstos el operador de *mutación*. Este operador sumará a cada coordenada de los k puntos que componen el gen de

cada uno de los hijos, un número aleatorio que sigue la distribución $N(0,1)$, con una determinada probabilidad p_m. De esta manera para cada punto $((x,y))$ perteneciente a G_i^{*t} o G_j^{*t}, generamos un número aleatorio r $(0 < r \leq 1)$. Si r es mayor que la probabilidad de mutación p_m no será sometido a mutación, en caso contrario generaremos dos números aleatorio a y b que siguen la distribución $N(0,1)$ y sustituiremos el punto (x,y) por $(x + a/100, y + b/100)$.

La condición de parada para nuestra heurística ha sido la misma a la diseñada para solucionar el problema MaxA-p-Pv1(P). Por ello, se ha considerado en nuestro algoritmo genético como condición de parada que el *fitness* de la población $F(A(t))$ no varíe durante un número h de generaciones. En nuestro caso se ha considerado $h = 20$. Es decir AG para cuando

$$\exists t \in \mathbb{N} \ / \ F(A(t+h)) \leq F(A(t)) \quad \forall h = 1, ..., 20$$

y por tanto t representa la generación de parada.

A modo de ejemplo presentamos en las Figuras 8.10 y 8.11 el resultado de aplicar la heurística GA al polígono de la Figura 8.9, con $k = 2$ y $k = 3$. Los resultados se han obtenido tomando las probabilidades de cruce y mutación $p_c = 0.8$ y $p_m = 0.05$ respectivamente y como se puede comprobar son prácticamente iguales a los obtenidos con SA, con alguna variación siendo sensiblemente inferior el valor de la *función objetivo* en el caso $k = 2$ y sin embargo, mínimamente superior para $k = 3$.

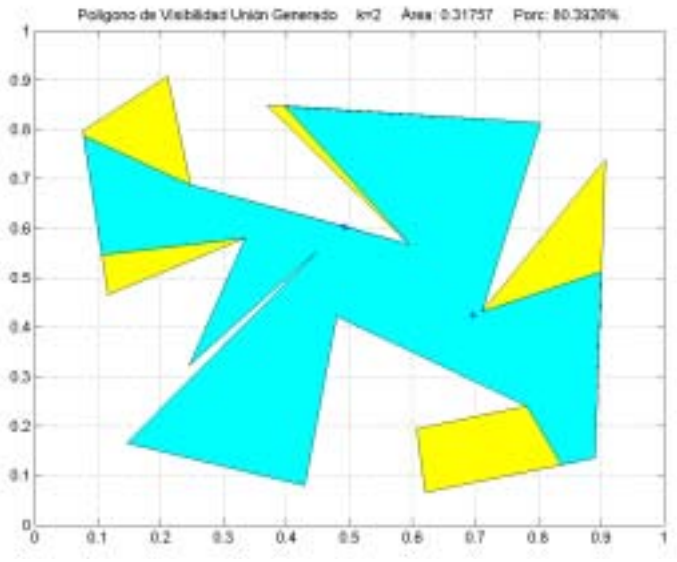


Figura 8.10: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P,k) con GA $k = 2$ $p_c = 0.8$ $p_m = 0.5$

```
ALGORITMO GENÉTICO GA pc=0.8 pm=0.05 K=2:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.492748,0.603879)
                                 p2=(0.697624,0.425612)
 - Área máxima.................. 0.31757 80.3926%
```

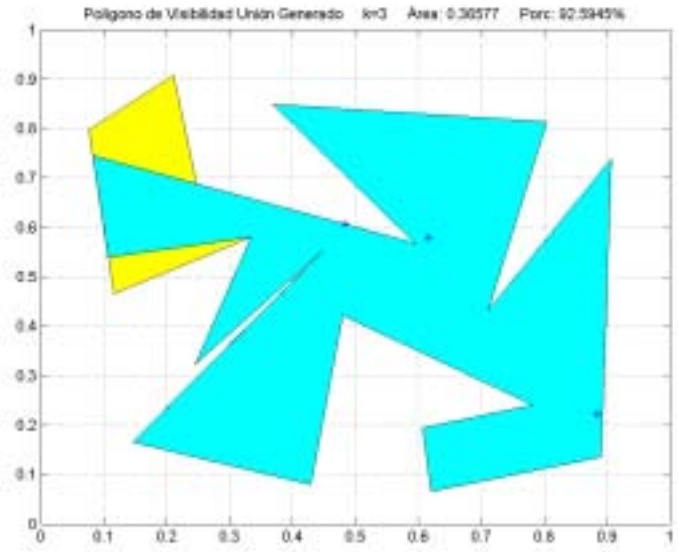


Figura 8.11: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con GA $k = 3$ $p_c = 0.8$ $p_m = 0.5$

```
ALGORITMO GENÉTICO GA pc=0.8 pm=0.05 K=3:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.483805,0.607203)
                                 p2=(0.617147,0.579044)
                                 p3=(0.883998,0.221208)
 - Área máxima.................. 0.36577 92.5945%
```

8.5 Aproximación a MaxA-p-Pvk(P,k) con random search-RS

Podemos utilizar para solucionar MaxA-p-Pvk(P, k) técnicas de búsqueda aleatoria sobre el conjunto de configuraciones o soluciones factibles de nuestro problema. Para ello, igual que se propuso en la Seccion 7.4 para solucionar el problema MaxA-p-Pv1(P), se generará una nube aleatoria $N = \{S_1, S_2, .., S_m\}$ pero ahora de subconjuntos de k puntos, es decir, de soluciones factibles de nuestro problema. Por tanto:

$$S_i = \{p_1^i = (x_1^i, y_1^i), ..., p_k^i = (x_k^i, y_k^i)\} \quad i = 1, ..., m$$

La solución que proporcionará la heurística se obtendrá realizando una búsqueda secuencial de la configuración $S_i \in N$ tal que el área conjunta iluminada en el interior de P, por sus k puntos sea máxima.

$$\acute{A}rea(V(P, S_i)) \geq \acute{A}rea(V(P, S_j)) \quad \forall j \neq i \quad i, j \in \{1, ..., m\} \tag{8.5.12}$$

Utilizando la función Genera-Nube descrita en la página 121 para generar una nube de m puntos, (en nuestro caso serán k), la función que generará la nube de configuraciones N la podemos describir de la siguiente manera:

Función Generar-Configuraciones

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$, y dos números enteros: m, (número de configuraciones), y k, (tamaño de la configuración).
SALIDA: Un conjunto $N = \{S_1, ..., S_m\}$ de m conjuntos con k puntos interiores a P cada uno de ellos.

```
[01]  i ←1;
[02]  N = {};
[03]  do
[04]    {S_i ← Generar − Nube(P, k);
[09]     N ← N ∪ S_i;
[10]     i ← i + 1;
[11]    }while (i <= m);
[12]    return N;
```

Como se mencionó en el Capítulo 7 el cardinal de la nube de configuraciones N juega un papel fundametal en el resultado de la heurística. En este sentido, se debe considerar un tamaño "razonable" para la nube N pues en otro caso el tiempo de respuesta de la heurística sería demasiado elevado, (para cada punto $p_j \in S_i$ $j = 1, ..., k$, siendo $S_i \in N$ una configuración de puntos, se debe calcular su *polígono de visibilidad* $V(P, p_i)$, calculando después la unión de estos *polígonos de visibilidad* mediante el algoritmo de *Weiler-Atherton* descrito y triangulando dicho polígono, (que pueden ser no conexo y con agujeros), para calcular su área). Se ha considerado que N tiene m elementos, es decir, m conjuntos de k puntos, con $m = 50kn$, siendo n el número de vértices del polígono P. De esta forma tanto el número n de vértices del polígono P, como el número k de *luces* buscadas, influyen en el cardinal de la nube de configuraciones. Debe recordarse que para $k = 1$, es decir, para resolver el problema MaxA-p-Pv1(P), se consideró $m = 50n$.

En forma de pseudocódigo el algoritmo RS propuesto para el problema MaxA-p-Pvk(P, k) es el siguiente:

Algoritmo RS-MaxA-p-Pvk(P, k)

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Un conjunto $S = \{p_1 = (x_1, y_1), ..., p_k = (x_k, y_k)\}$ de k *luces-punto*, cuyo área conjunta iluminada en P es máxima.

```
[01]  área ← 0;
[02]  N ← Generar_Configuraciones(P, 50kn, k);
```

```
[03]   for (S_i ∈ N)
[08]     {Calcular el polígono de visibilidad union de p_j ∈ S_i en P,V(P,S_i);
[09]      if (area < Área(V(P,S_i))
[10]        área ← Área(V(P,S_i));
[11]        S ← S_i;
[12]     }
[13]     return S;
```

Presentamos en las Figuras 8.12 y 8.13 el resultado de aplicar RS al polígono de la Figura 8.9, con $k = 2$ y $k = 3$, con las nubes de puntos empleadas.

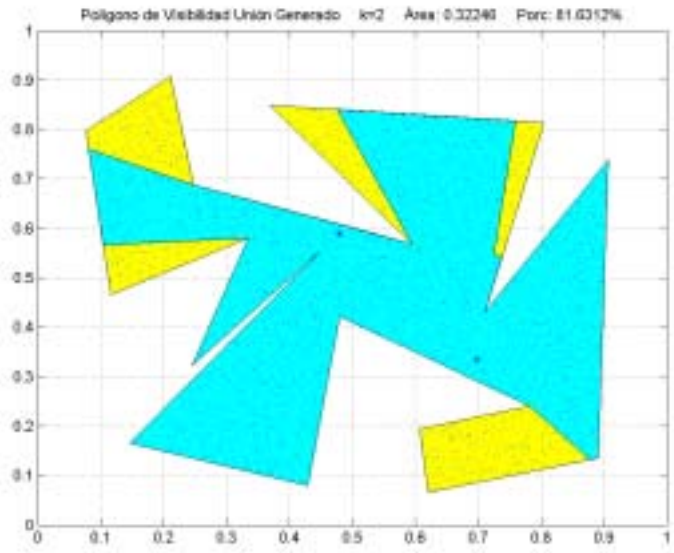


Figura 8.12: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con RS $k = 2$ $m = 1900$

```
ALGORITMO RANDOM SEARCH RS m=1900 K=2:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.489831,0.597144)
                                 p2=(0.691022,0.341987)
 - Área máxima.................. 0.322462 81.6312%
```

Como se puede observar los resultados obtenidos son peores a los mostrados por *simulated annealing SA*, cuya respuesta es incluso muy distinta en las coordenadas de uno de los puntos solución, ($p3 = (0.6064423, 0.557482)$ con SA y $p3 = (0.715221, 0.801295)$ con RS), para $k = 3$; sin embargo el porcentaje iluminado es muy similar para ambas heurísticas, (92.22% en SA y 91.79% con RS). Es importante destacar que para el problema MaxA-p-Pv1(P), es decir para $k = 1$, RS tenía mejor comportamiento que SA, tanto en tiempo de ejecución, como en porcentaje de área iluminada, como se analizó en el capítulo anterior.

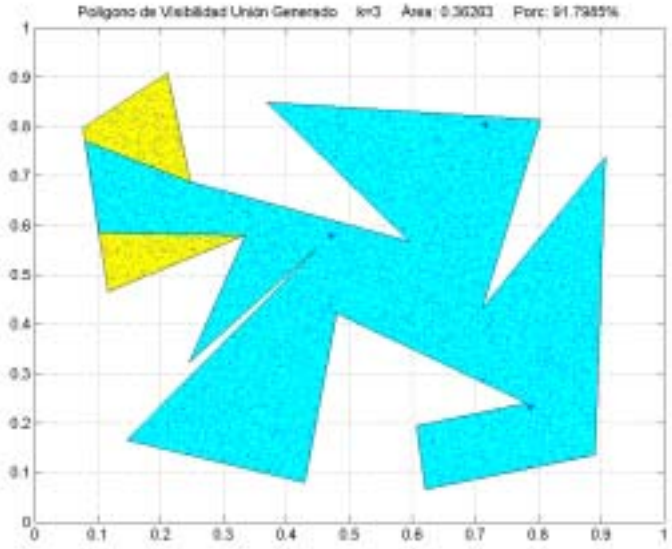


Figura 8.13: Un ejemplo de la aproximación del problema MaxA-p-Pvk(P, k) con RS $k = 2$ $m = 1900$

```
ALGORITMO RANDOM SEARCH RS m=2850 k=3:
 * Polígono de visibilidad unión calculado
 - Puntos de área máxima........ p1=(0.475843,0.593192)
                                 p2=(0.798341,0.230712)
                                 p3=(0.715221,0.801295)
 - Área máxima.................. 0.36263 91.79853%
```

8.6 Test comparativos

Para realizar un estudio comparativo de las tres heurísticas expuestas se han aplicado, (igual que se hizo en el capítulo anterior para el problema MaxA-p-Pv1(P)), sobre cuatro conjuntos de 50 polígonos con 50, 100, 150 y 200 vértices cada uno de ellos, considerando $k = 3$ y obteniendo las medias de porcentaje de área iluminada, tiempo empleado por la heurística y número de iteraciones. Los resultados obtenidos se presentan en las tres siguientes tablas:

► **Resultados** *simulated annealing-SA*

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	88.4000	376.12	10156
100	62.4080	916.68	20381
150	51.7010	2012.33	30629
200	40.5849	4491.36	40897

Tabla 8.1: Resultados simulated annealing-SA $k = 3$

► **Resultados** *algoritmos genéticos-GA.*

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	85.0311	778.8	12278
100	56.2565	1835.2	14504
150	45.9120	3625.2	20602
200	33.6129	5429.8	33793

Tabla 8.2: Resultados algoritmos genéticos-GA $k = 3$

► **Resultados** *random search-RS.*

Vértices	% Área visible	Tiempo en seg.	Iteraciones
50	88.4106	260.04	7500
100	58.6558	765.92	15000
150	47.7797	1598.40	22500
200	37.9243	3368.08	30000

Tabla 8.3: Resultados random search-RS $k = 3$

En la siguiente figura mostramos gráficamente los resultados de estas tres tablas. La coordenada x representa el número de vértices de los polígonos entrada y la coordenada y el porcentaje medio de área conjunta iluminada por $k = 3$ luces en el interior de cada polígono, respecto a su área, es decir, el área del polígono *unión* de tres *polígonos de visibilidad* correspondientes a puntos interiores al polígono.

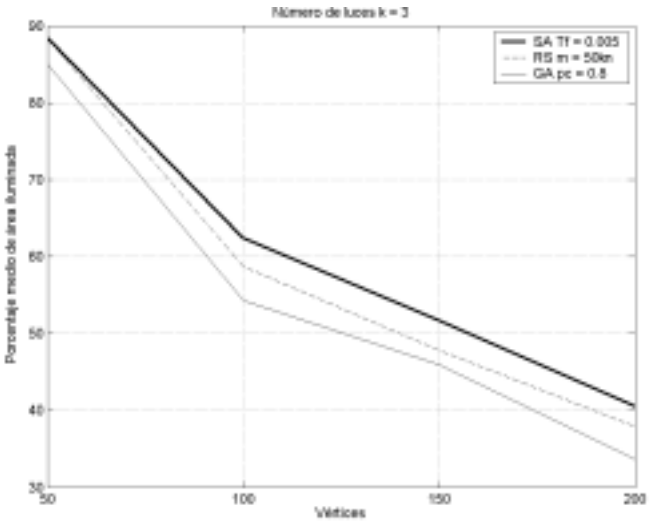


Figura 8.14: Resultados para SA $T_0 = n$ $T_i = \frac{T_0}{1+i}$ frente a RS y GA

Como podemos apreciar SA obtiene mejores resultados tanto en área iluminada, como en tiempo de ejecución y en número de iteraciones. Sin embargo, para un número pequeño de vértices RS obtiene también resultados aceptables, aunque el número de iteraciones es superior a SA.

Contraste de hipótesis

Para poder obtener conclusiones sobre los resultados obtenidos es necesarios que los datos producidos con cada heurística sean significativamente diferentes, es decir, que los porcentajes iluminados en cada caso no sigan la misma distribución. Por ello, se han realizado contrastes T, (utilizando el software matemático MatLab), con un nivel de significación del 95%, utilizando los resultados obtenidos sobre los 50 polígonos de 50 vértices, cuyos medias se encuentran en las primeras filas de las tablas anteriores. Las respuestas aportadas por los contrastes realizados se presentan en la siguiente tabla, indicando con el signo '+' si el contraste es significativo y con '–' si no lo es:

Algoritmo	*Simulated annealing-SA*	*Algorit. genéticos-GA*	*Random Search-RS*
Simulated annealing-SA	•	+	–
Algorit. genéticos-GA	+	•	+
Random Search-RS	–	+	•

Tabla 8.4: Tabla de contrastes realizados para los datos de polígonos con 50 vértices

Según los resultados obtenidos, podemos concluir que RS y SA no obtiene resultados significativamente diferentes para polígonos de 50 vértices. Debe observarse no obstante, que para polígonos con mayor número de vértices el área iluminada aplicando la segunda heurística mejora el obtenido por SA. En la siguiente tabla mostramos el resultado de los contrastes realizados sobre los resultados de polígonos con 100 vértices.

Algoritmo	*Simulated annealing-SA*	*Algorit. genéticos-GA*	*Random Search-RS*
Simulated annealing-SA	•	+	+
Algorit. genéticos-GA	+	•	+
Random Search-RS	+	+	•

Tabla 8.5: Tabla de contrastes realizados para los datos de polígonos con 100 vértices

Para 150 y 200 vértices el contraste sigue siendo positivo.

Curvas de crecimiento

Presentamos en esta sección a modo de ejemplo las curvas de crecimiento de los ejemplos de las Figuras 8.8 para la heurística SA y de la Figura 8.10 para la heurística GA:

1. Las *curvas de crecimiento* relacionan la iteración $i-ésima$ del algoritmo con el valor de la *función objetivo* en dicha iteración. Mostramos en la Figura 8.15 esta curva para el problema MaxA-p-Pvk(P,k), $k=2$, aproximado con *simulated annealing* SA, tomando

el polígono ejemplo de la Figura 8.8, (debe hacerse notar que para poder comparar mejor el comportamiento de las heurísticas se han considerado un número similar de iteraciones en todas ellas, aunque ya hemos estudiado anteriormente el número de iteraciones que utiliza cada método para los parámetros utilizados).

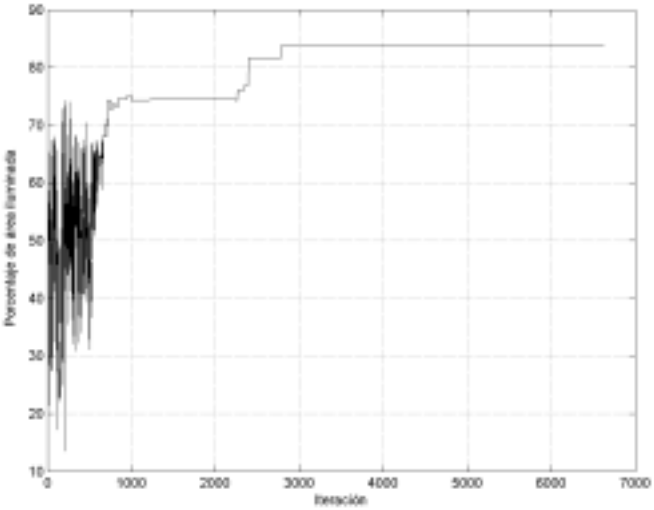
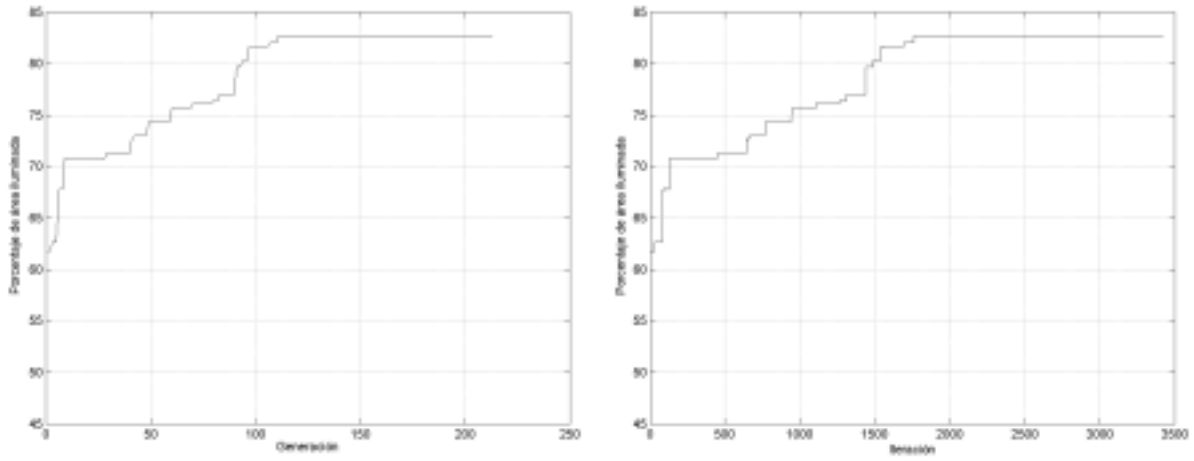


Figura 8.15: Curva de crecimiento para el problema MaxA-p-Pvk(P,k) con SA $k=2$

Comparando esta gráfica con la Figura 7.10, (curva de crecimiento de SA para $k=1$), observamos como cuando el número de luces aumenta las oscilaciones en la iteraciones iniciales también aumentan. Esto es debido a que en este caso el valor de δ, que representa la diferencia de la *función de coste* o *función fitness* C en una iteración i y en $i+1$, converge más lentamente. Ello produce que aumente la probabilidad de encontrar un valor aleatorio uniforme $U(0,1)$ tal que $U(0,1) < e^{\left(\frac{-\delta}{T}\right)}$ y por tanto la probabilidad de visitar configuraciones desfavorables.

2. En el caso de la heurística GA para el mismo polígono inicial, mostramos dos *curvas de crecimiento,* (Figura 8.16): la figura (a) representa en la coordenada x la generación de GA y en la coordenada y el porcentaje de área iluminada para cada generación; la segunda curva sin embargo, muestra en la coordenada x cada iteración o llamada a la *función objetivo*. Esta segunda curva nos permite establecer una comparativa más clara con la curva de la Figura 8.15, obtenida para la heurística *simulated annealing-SA*.

   Podemos observar como la *función fitness* o *función objetivo* a maximizar va aumentando a medida que avanza el *algoritmo genético*, presentando intervalos en las que el área de zona iluminada permanece estable. Por tanto, los operadores diseñados permiten tender a la solución aproximada del problema MaxA-p-Pvk(P,k). Además para GA la *curva de crecimiento* no presenta las oscilaciones que manifiesta la *curva de crecimiento* para SA, con lo cual sería más recomendable esta heurística si el número de iteraciones es bajo.



(a) (b)

Figura 8.16: Curvas de crecimiento para el problema MaxA-p-Pvk(P, k) con GA $k = 2$

3. Para la tercera heurística estudiada, *random search-RS*, la *curva de crecimiento* manifiesta una convergencia más rápida que las otras dos heurísticas estudiadas. Además el resultado final mejora a la heurística GA, lo que en media también se produce como hemos apreciado en la Tabla 8.3.

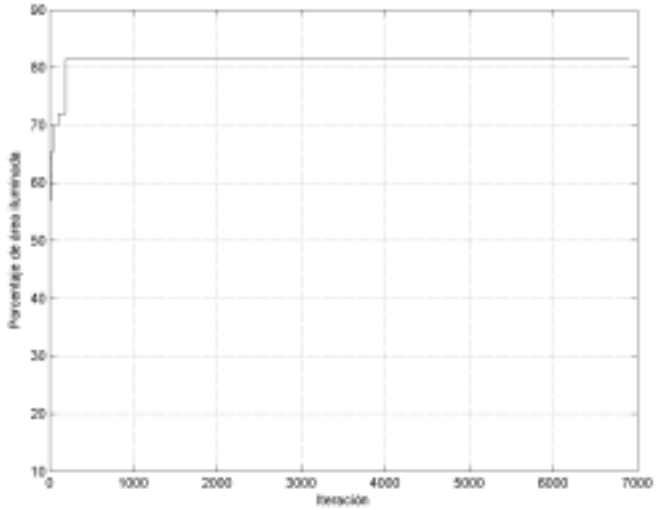


Figura 8.17: Curva de crecimiento para el problema MaxA-p-Pvk(P, k) con RS $k = 2$

La Figura 8.18 muestra las tres curvas de crecimiento conjuntamente. Como podemos observar en este ejemplo concreto, los mejores resultados son los obtenidos por RS, aunque su convergencia es más lenta. Esta misma idea se observa en las tablas de datos expuestas anterioremente.

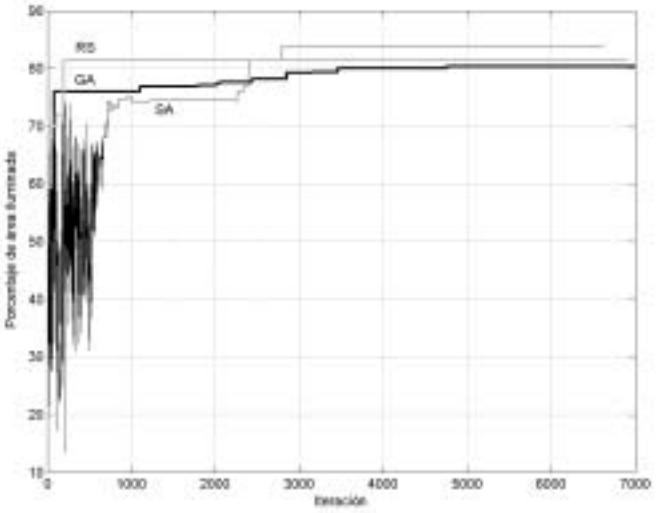


Figura 8.18: Comparativa de curvas de crecimiento para el problema MaxA-p-Pvk(P, k)

Sin embargo se necesita realizar un estudio detallado sobre un conjunto más numeroso de polígonos aleatorios para poder obtener conclusiones. En la Figura 8.19 mostramos las *curvas de crecimiento* medio de polígonos con 100 vértices. Para obtener estas curvas se ha aplicado la técnica *binaria* a 50 polígonos de 100 vértices, considerando $k = 3$ y obteniendo las *curvas de crecimiento* para cada una de ellas. Finalmente se ha realizado la media iteración a iteración de cada una de las curvas.

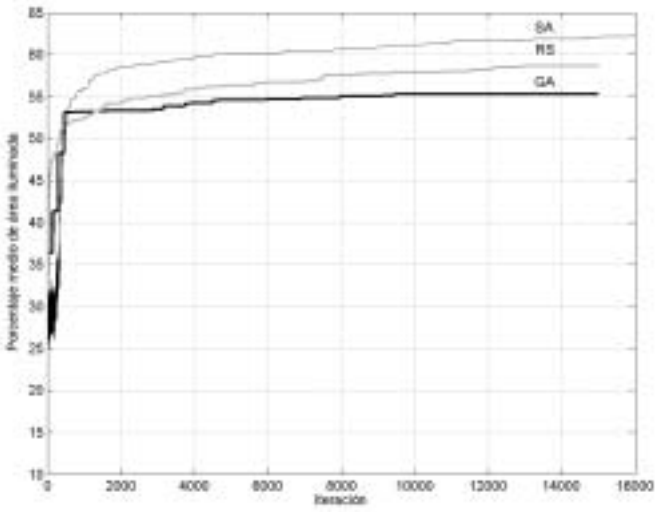


Figura 8.19: Curvas de crecimiento medio para MaxA-p-Pvk(P, k) con $n = 100$ y $k = 3$

Así, de la figura anterior y de las Tablas 8.1, 8.2 y 8.3, podemos determinar las siguientes conclusiones:

- En general *simulated annealing-SA* obtiene mejores resultados para el problema MaxA-p-Pvk(P, k), en porcentaje de área iluminada.
- Una búsqueda aleatoria o *random search-RS*, obtiene buenos resultados, aunque peores que SA. Además la convergencia de esta técnica es rápida con respecto a las otras dos estudiadas. Según los contrastes realizados SA y RS no son significativamente diferentes, para polígonos con pocos vértices, (el contraste se ha realizado para los datos obtenidos con 50 vértices).
- Los *algoritmos genéticos-GA*, obtienen en media peores resultados que RS y SA, además la convergencia de GA es similar a RS y SA, respecto al número de iteraciones, pero el tiempo medio de respuesta de la heurística es mayor, es decir, la relación *tiempo/iteración* es mayor.

Una vez analizado el problema MaxA-p-Pvk(P, k), estamos ya en condiciones de aportar una respuesta aproximada para MinN-p-Pvk(P). Debe recordarse que este problema, que es de naturalera $\mathcal{NP}$-dura [71], fué el que motivó este capítulo.

8.7 El problema MinN-p-Pvk(P)

Formalmente el problema MinN-p-Pvk(P), lo hemos enunciado de la siguiente manera:

minimización del numero de luces punto que iluminan un polígono
MinN-p-Pvk(P):
ENTRADA: Un polígono P de n vértices.
PREGUNTA: ¿Cuál es el número mínimo k de *luces punto* necesarios para iluminar el polígono P?

Proponemos dos estrategias para buscar el número mínimo de luces que iluminan completamente el polígono P: una estretegia secuencial que partiendo de $k = 1$ aumentará su valor de uno en uno hasta encontrar un k tal que el área conjunta iluminada por las luces que proporcione MaxA-p-Pvk(P, k), sea igual al área de P y una estrategia binaria, que partiendo del intervalo $\left[1, \left\lfloor \frac{n}{3} \right\rfloor\right]$, irá dividiendo sucesivamente dicho intervalo en función del área iluminada por sus extremos, hasta encontrar un intervalo de la forma $[a, a+1]$, proporcionando la solución $a+1$. Los detalles y los datos aportados por las experimentaciones realizadas se muestran en la siguiente sección.

8.7.1 Estrategia secuencial

Partiendo de $k = 1$, podemos ir aumentado dicho valor hasta encontrar k tal que el área conjunta iluminada por las k luces de máxima iluminación aportadas por MaxA-p-Pvk(P, k), sea tan cercana como queramos al área de P. En forma de pseudocódigo esta estrategia la podemos

representa de la siguiente manera:

Algoritmo estrategia secuencial

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Un entero k, que representa al número mínimo de luces que iluminan completamente P.

```
[01]  k ←0;
[03]  AP ← Área(P);
[03]  do
[04]    {k ← k+1;
[05]     AS ←Área MaxA-p-Pvk(P, k);
[08]    }while (AS < AP);
[10]    return k;
```

8.7.2 Estrategia binaria

Si represetamos también con Área MaxA-p-Pvk(P, k), al área iluminada por lás k luces de máxima iluminación aportada por alguna de la heurísticas construidas para este problema, un pseudocódigo que represente la estrategia secuencial se puede expresar de la siguiente manera:

Algoritmo estrategia binaria

ENTRADA: Un polígono P de n vértices, $\{v_1, ..., v_n\}$.
SALIDA: Un entero k, que representa al número mínimo de luces que iluminan completamente P.

```
[01]  a ←1;
[01]  b ← ⌊n/3⌋;
[03]  AP ← Área(P);
[03]  do
[04]    {AS ←Área MaxA-p-Pvk(P, ⌊(a+b)/2⌋);
[08]     if(AS < AP)
[08]       a ← ⌊(a+b)/2⌋
[08]     else
[08]       b ← ⌊(a+b)/2⌋
[08]    }while (b ≠ a + 1);
[10]    return b;
```

Los datos comparativos aportados por ambas estrategias para polígonos generados aleatoriamente con nuestro generador aleatorio de polígonos RPG se muestran en la siguiente sección:

8.7.3 Estudios comparativos

En la siguiente tabla mostramos el número de luces que se producen como solución y el tiempo utilizado por ambas estrategias para polígonos generados aleatoriamente con un número de vértices comprendidos entre 10 y 100. Para las llamadas al problema MaxA-p-Pvk(P, k) se ha utilizado la heurística *simulated annealing-SA*, descrita en la Sección 8.3 y que ha obtenido los mejores resultados en los tests comparativos.

Vértices	*Luces* **secuencial**	*Tiempo en seg.*	*Luces* **binaria**	*Tiempo en seg.*
10	2	32.0	2	18.0
20	3	99.0	3	77.0
30	3	118.0	3	199.0
40	4	353.0	4	591.0
50	7	1426.0	8	1839.0
60	9	2690.0	9	2565.0
70	11	3364.0	11	3321.0
80	15	4634.0	14	4167.0
90	19	5546.0	19	4987.0
100	23	7356.0	24	6285.0

Tabla 8.6: Comparativa de estrategias para MinN-p-Pvk(P)

De los datos de la Tabla 8.6 se deduce que la estrategia *binaria* es más recomendable a medida que aumenta el número de vértices del polígono, ya que en este caso el tiempo empleado por dicha estrategia es menor al empleado por la estrategia *secuencial*, siendo la solución aportada por ambas estrategias prácticamente igual. Además se observa como el número de luces es siempre inferior a $\lfloor \frac{n}{3} \rfloor$.

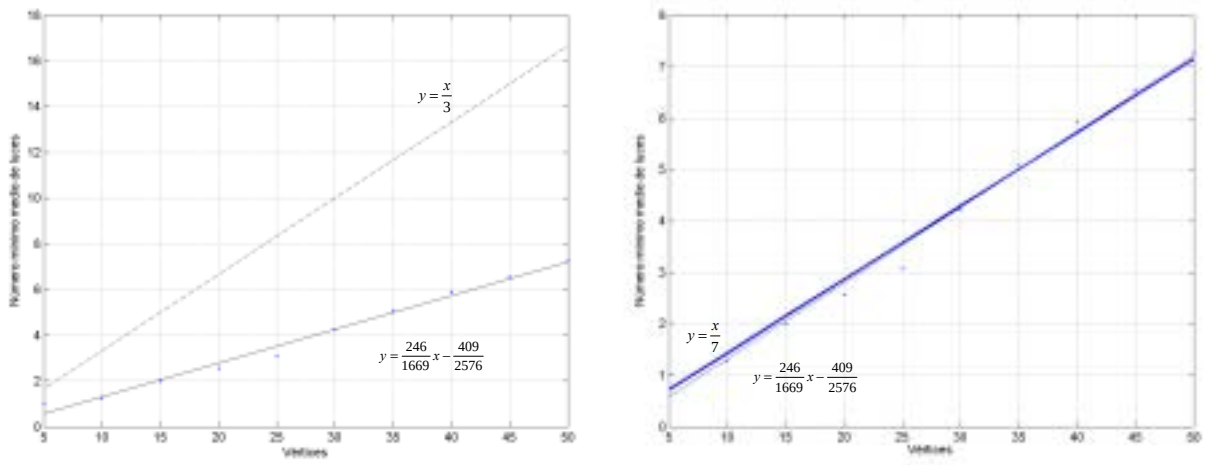
Vértices	*Solución media para* MinN-p-Pvk(P)
5	1.0000
10	1.2571
15	2.0000
20	2.5714
25	3.0857
30	4.2286
35	5.0857
40	5.9137
45	6.5423
50	7.2610

Tabla 8.7: Luces mínimas medias para iluminar P

En la Tabla 8.7 mostramos la solución media aportada por la estrategia *binaria* para polígonos con un número de vértices entre 5 y 50 y múltiplos de 5. Para cada número de vértices

se ha aplicado la estrategia a un total de 50 polígonos generados aleatoriamente con nuestro generador RPG, obteniendo posteriormente la media de la respuesta de dicha estrategia para MinN-p-Pvk(P).

Representamos estos datos en la Figura 8.20 (a). Dibujamos también la recta $y = \frac{x}{3}$, y el ajuste por mínimos cuadrados lineal que relaciona el número de vértices en la coordenada x, con el número mínimo de luces necesarias en media para iluminar un polígono con x vértices.



(a) (b)

Figura 8.20: Datos medios y su ajuste para el problema MinN-p-Pvk(P)

Como se puede comprobar en media el número mínimo de luces que iluminan un polígono P con n vértices es inferior a $\lfloor \frac{n}{3} \rfloor$, siendo el ajuste lineal que se obtiene

$$y = \frac{246}{1669}x - \frac{409}{2576} \approx \frac{x}{6.78} - 0.15 \approx \frac{x}{7} \tag{8.7.13}$$

donde x representa el número de vértices de P. En la Figura 8.20 (b) representamos el ajuste de datos obtenido y la curva $y = \frac{x}{7}$. Podemos deducir que en media el número mínimo de luces que ilumina un polígono P con n vértices es aproximadamente $\frac{n}{7}$.

8.8 Conclusiones y trabajos futuros

En este capítulo hemos presentado soluciones heurísticas para el problema de mínimización del número de luces que iluminan un polígono P con n vértices. Este problema los hemos denotado con MinN-p-Pvk(P) y para solucionarlo se han estudiado dos posibles estrategias: *secuencial* y *binaria*. Ambas estrategias necesitan el cálculo del conjunto de k puntos interiores a P, tal que el área conjunta iluminada por estas k luces sea máxima. Este problema lo hemos denotado con MaxA-p-Pvk(P, k) y se analizan tres método heurístico para su aproximación: *simulated annealing-SA*, *algoritmos genéticos-GA* y *random search-RS*.

Se ha realizado también un estudio comparativo sobre la bondad de cada uno de los métodos para el segundo problema y sobre cada una de las estrategias para el primero, utilizando polígonos generados aleatoriamente con *RPG*. Finalmente se presentan los resultados de un estudio experimental que muestra que en media el número mínimo de luces que iluminan un polígono completamente es aproximadamente $\frac{n}{7}$, siendo n el número de vértices del polígono. Para la realización de todos estos análisis se ha considerado que un polígono está completamente iluminado si lo está en un 99% de su área. De forma resumida presentamos en la siguiente tabla las heurísticas presentadas, así como los parámetros utilizados en cada una de ellas.

Problema Estudiado	*Solución Aportada*	*Parámetros*
MaxA-p-Pvk(P,k)	Aprox. *Simulated Annealing-SA*	$T_0 = n \;\; T_i = \frac{T_0}{1+i}$ (FSA) $\;\; T_f \leq 0.005$
	Aprox. *Algoritmos Genéticos-GA*	*Selec.: proporcional / Cruce:* 1 *punto* $p_c = 0.8 \;\; p_m = 0.05$
	Aprox. *Random Search-RS*	$m = 50kn$
MinN-p-Pvk(P)	Estrat. *Secuencial*	*según* MaxA-p-Pvk(P,k)
	Estrat. *Binaria*	

Tabla 8.8: Técnicas diseñadas par los problemas MaxA-p-Pvk(P,k) y MinN-p-Pvk(P)

Es importante notar que la estrategia del *gradiente-GRAD* presentada en el capítulo anterior para solucionar el problema MaxA-p-Pv1(P), no se utiliza en este capítulo, pues puede suceder como se muestra en la Figura 7.6 (b), que ni colocando luces en todos los puntos de convergencia de focos situados en los vértices de P, podamos iluminarlo completamente.

Se puede considerar también un abanico amplio de problemas que pueden ser estudiados en futuras investigaciones. En primer lugar se podría hacer un estudio más exhaustivo de los parámetros de cada una de las heurísticas, probando por ejemplo otros operadores de cruce o mutación en los *algoritmos genéticos* o otros decrecimientos de temperatura en *simulated annealing*. También sería interesante la construcción de técnicas heurísticas similares a las presentadas en este capítulo, pero donde el polígono inicial sea un polígono con agujeros o tomando variaciones de la iluminación clásica como puesden ser las *iluminación de alcance limitado* presentada en el Capítulo 3, o la $t-$*buena iluminación* presentada en el Capítulo 4.

Igualmente se podría estudiar la relación existente entre la descomposición $\mathcal{S}$ de P y la minimización de luces: ¿colocando luces en los vértices de $\mathcal{S}$ iluminaremos completamente P? Intutivamente la respuesta a esta pregunta parece ser afirmativa, aunque el coste computacional de este método sería demasiado elevado.

Capítulo 9

Maximización de la región de Voronoi

Dada una región del plano R y un conjunto N de n puntos en ellas, podemos encontran una subdivisión de dicha región, asociando a cada punto del conjunto los puntos de la región que se encuentran más cerca de dicho punto, que de cualquier otro del conjunto. Sabido es que dicha subdivisión se denomina *diagrama de Voronoi* y que está intrínsecamente relacionada con la *triangulación de Delaunay* [12].

Un problema para el que actualmente no se conocen cotas algorítmicas eficientes, (desconociendo incluso si podría llegar a ser un problema $\mathcal{NP}$-duro), es el problema de buscar un nuevo punto q en R, tal que la región que se le asocie a q en el *diagrama de Voronoi* del conjunto $N \cup \{q\}$, tenga área máxima. Presentamos en este capítulo dos técnicas heurísticas que abordan este problema. La primera de ellas es una técnica *simulated annealing-SA* y la segunda una técnica de búsqueda aleatoria o *random search-RS*.

9.1 Introducción

Dentro de la Geometría Computacional una de la estructuras más conocidas son los *Diagramas de Voronoi*, [12]. Según se ha mencionado anteriormente, si tenemos un conjunto $N = \{p_1, p_2, ..., p_n\}$ de n puntos en un región plana R, podemos asociar a cada punto $p_i \in N$, el conjunto de puntos de R que están más cerca de p_i que de cualquier otro punto $p_j \in N$ $i \neq j$. Esto permite obtener una subdivisión de R que se denomina *diagrama de Voronoi* del conjunto N en R y que denotaremos con $Vor(N, R)$. En la Figura 9.1 exponemos un ejemplo del *diagrama de Voronoi* de una nube de 150 puntos en el cuadrado unidad.

Formalmente una *región de Voronoi* y un *diagrama de Voronoi* se pueden definir de la siguiente manera:

Definición 9.1.1 *Dada una región R del plano euclídeo y un conjunto $N = \{p_1, p_2, ..., p_n\}$ de n puntos en R, se define la región de Voronoi de un punto $p_i \in N$ y se denota con $Rv(p_i, N, R)$*

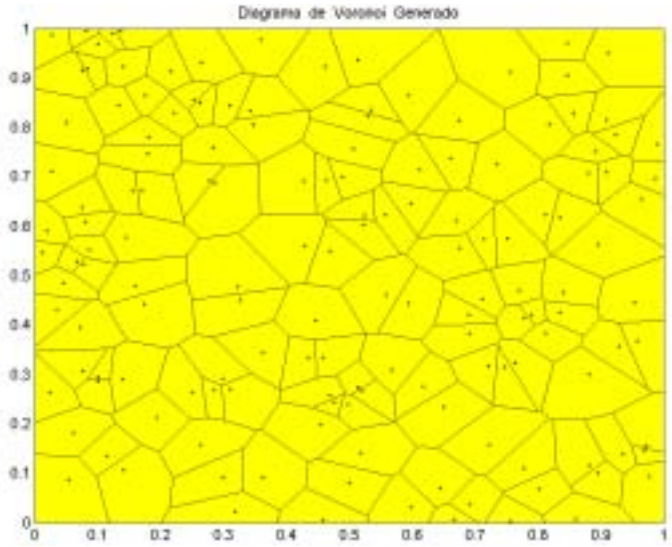


Figura 9.1: Una entrada para problema MaxA-p-Vor(N) con Card(N)$= 150$

como el conjunto de puntos de R que están más cerca de p_i, que de cualquier otro punto de N:

$$Rv(p_i, N, R) = \{x \in R \ / \ |p_i - x| \leq |p_j - x| \ \ \forall p_j \in N \ j \neq i\} \tag{9.1.1}$$

Por tanto,el conjunto de regiones asociadas a todos y cada uno de los $p_i \in N$ anteriores constituyen el *diagrama de Voronoi.*

Definición 9.1.2 *Dada una región R del plano euclídeo y un conjunto $N = \{p_1, p_2, ..., p_n\}$ de n puntos en R, el diagrama de Voronoi de N en R es el conjunto de todas las regiones de Voronoi de puntos del conjunto N:*

$$Vor(N, R) = \bigcup_{i=1}^{n} Rv(p_i, N, R) \tag{9.1.2}$$

El problema que estudiamos en este capítulo de forma heurística, es el de la búsqueda de un nuevo punto $q \in R$, tal que su *región de Voronoi* $Rv(q, N \cup \{q\}, R)$ tenga área máxima. Este problema que denotaremos con MaxA-p-Vor(N), lo podemos enunciar de la siguiente manera:

maximización de la región de voronoi asociada a un nuevo punto

MaxA-p-Vor(N):

ENTRADA: El *diagrama de Voronoi* de un conjunto $N = \{p_1, p_2, ..., p_n\}$ de n puntos en una región R del plano.

PREGUNTA: ¿Cuál es el punto $q \in R$, tal que el área de la *región de Voronoi* $Rv(q, N \cup \{q\}, R)$, que se asocia a q en el *diagrama de Voronoi* $Vor(N \cup \{q\}, R)$ sea máxima?

Consideraremos en todo lo que sigue que R es el cuadrado unidad, es decir, es el cuadrado de vértices $\{(0,0),(1,0),(1,1),(0,1)\}$. Así, todos los *diagramas de Voronoi* construidos aleatoriamente se encuentran restringidos a dicho cuadrado, por lo que realmente R es una región fija en los estudios de nuestras heurísticas.

No existen muchas referencias sobre el problema MaxA-p-Vor(N) en la bibliografía. Fundamentalmente ha sido referenciado en el contexto de juegos geométricos, donde cada jugador podrá mover sus puntos o introducir nuevos puntos de tal forma que se maximice el área de las *regiones de Voronoi* de cada jugador; ver las descripciones de Okabe y otros [79], Cheong y otros [22] y Ahn y otros [2]. Sin embargo, en ninguna de estas referencias se da un método explícito para maximizar la región de Voronoi de un nuevo punto. Sólo recientemente Dehne y otros [30], han estudiado este problema para el caso concreto de que los vecinos del nuevo punto q estén en posición convexa, demostrando en este caso que existe un máximo local.

Siguiendo la misma línea de los dos últimos capítulos estudiamos ahora dos técnicas heurísticas que abordan el problema MaxA-p-Vor(N). La primera de ellas está basada en la heurística *simulated annealing-SA* y la segunda sigue un mecanismo de búsqueda aleatoria o *random search-RS*. Se ha realizado también un estudio comparativo de ambos métodos sobre conjuntos de *diagramas de Voronoi* generados aleatoriamente, cuyos resultados se exponen en la Sección 9.4.

- *Simulated Annealing-SA*: La entrada para el problema MaxA-p-Vor(N) difiere sustancialmente de los datos de entrada de los problemas tratados hasta ahora, ya que en este caso el elemento de entrada es un *diagrama de Voronoi* de un conjunto N de puntos, mientras que en los problemas tratados anteriormente siempre contábamos con un polígono cuya región iluminada se deseaba maximizar de una u otra forma. Sin embargo, el dato de salida de todos los problemas es un punto $p = (x, y)$ del plano, que en el caso que nos ocupa maximice el área de su *región de Voronoi*, mientras que en los problemas anteriores maximizaba su región iluminada. Por tanto, debemos determinar los elementos de la heurística para el caso concreto del problema MaxA-p-Vor(N): *función de coste* C, conjunto S de configuraciones, vecindad, estrategias de templado y temperatura inicial. Como se muestra en la Sección 9.2 se ha considerado, como se hizo para el problema MaxA-p-Pvk(P, k), la mejor combinación de los parámetros *temperatura inicial* y *disminución de la temperatura*, que se obtuvo en el análisis realizado en el Capítulo 7 para el problema MaxA-p-Pv1(P). Esta combinación fue la analizada en el Caso 4 de la Sección 7.6.1, donde $T_0 = n$ y $T_k = \frac{T_0}{1+k}$ (FSA), siendo n ahora el cardinal del conjunto N.

- *Random Search-RS*: Dado el *diagrama de Voronoi* de un conjunto N de puntos, sobre una región R del plano, podemos realizar una búsqueda aleatoria sobre el conjunto de puntos de R. El tamaño de la nube de puntos sobre la que se realiza la búsqueda jugará un papel esencial en el método y que en nuestro caso estará relacionado con el cardinal del conjunto N. En la Sección 9.3 presentamos esta heurística y en la Sección 9.4 los resultados comparativos respecto a SA. A pesar de la sencillez de RS comprobaremos como aunque tiene una convergencia rápida, obtiene peores resultados en media que SA. Por tanto será útil cuando deseemos respuestas rápidas, pero poco óptimas.

La realización de estudios experimentales con los métodos heurísticos que exponemos, necesita de el cálculo de *diagramas de Voronoi* de nubes de puntos aleatorios. Exponemos a continuación de forma resumida el algoritmo incremental construido para el cálculo del *diagramas de Voronoi*, cuya implementación se puede encontrar en el Apéndice B de listados.

9.1.1 Algoritmo incremental para la construcción de un diagrama de Voronoi

Existen numerosos métodos para construir el *diagrama de Voronoi* de una nube de puntos y por consiguiente su dual, la *triangulación de Delaunay*. En este apartado únicamente daremos los detalles generales del algoritmo implementado para construir *diagramas de Voronoi* sobre nubes de puntos aleatorios, que necesitaremos en nuestras experimentaciones.

El método incremental es uno de los más importantes debido a su simplicidad. Este método comienza con un *diagrama de Voronoi* simple, con un punto y lo modifica añadiendo otros puntos uno a uno. En este proceso, cuando se añade un nuevo punto p, se detecta en primer lugar la *región de Voronoi* en la que se encuentra p, generada por el punto q. La inclusión de p en la región de q hace necesario el cálculo del bisector entre estos dos puntos. Este bisector cortará en dos puntos, w_1 y w_2 a la región de q. Estos puntos serán el origen para continuar construyendo bisectores entre p y el resto de sus nuevos vecinos hasta completar un nuevo polígono convexo, la nueva *región de Voronoi* de p.

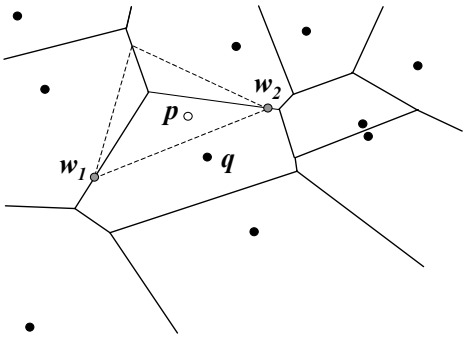


Figura 9.2: Construcción de la región de Voronoi de un nuevo punto p

Para que este algoritmo resulte realmente eficiente es preciso que la estructura de datos que acompañe al *diagrama de Voronoi* tenga información sobre las regiones adyacentes. De este modo, una vez localizado un punto q y construído el bisector, se puede pasar de una región a su contigua en tiempo constante.

La elección del algoritmo incremental ha sido debida a que la propia definición del problema MaxA-p-Vor(N) induce un proceso incremental al buscar un nuevo punto p tal que su *región de Voronoi*, (la región punteada en la figura anterior), dentro del *diagrama de Voronoi* que se da como entrada, tenga área máxima. Dicho área será el valor de la *función objetivo* o *función de coste* a maximizar de forma aproximada por las heurísticas *simulated annealing-SA* y *random search-RS*, cuyos elementos detallamos a continuación.

9.2 El problema MaxA-p-Vor(N) con simulated annealing-SA

Aunque la formulación de MaxA-p-Vor(N) es diferente a los problemas estudiados en los capítulos anteriores mediante *simulated annealing-SA*, el diseño de esta heurística comparte muchos de los elementos de configuración diseñados para los problemas MaxA-p-Pv1(P) y MaxA-p-Pvk(P, k). En el caso que nos ocupa la entrada del problema será el *diagrama de*

Voronoi de un conjunto $N = \{p_1, p_2, ..., p_n\}$ de puntos en el plano, como se muestra en la Figura 9.1 para un conjunto de 150 puntos. Así, los elementos de la heurística adaptados a nuestro problema se pueden describir de la siguiente manera:

9.2.1 Adaptación del problema

Según se explicó en el Capítulo 1, si $x \in S$ es la configuración inicial y T la temperatura en cada iteración, siendo $T_0 > 0$ la temperatura inicial, el esquema general del *simulated annealig*, es el siguiente:

```
[01]  do
[02]    {do
[03]       {Genera solución y ∈ Vecindad(x) ⊂ S;
[04]        Evalúa δ ← C(x) − C(y);
[05]        if (δ < 0) x ← y
[06]        else
[07]           if ((δ ≥ 0) ∧ (U(0,1) < e^(−δ/T))) x ← y;
[08]        n ← n + 1;
[09]       }while (n ≤ N(T));
[10]    Disminuir T;
[11]    }while (parada==false);
```

Por tanto, necesitamos detallar el conjunto S de configuraciones, la *función de coste* C, el criterio de vecindad, la configuración inicial elegida y las estrategias de templado.

Conjunto S de configuraciones

El conjunto de configuraciones o soluciones factibles para MaxA-p-Vor(N) serán todos los puntos $p = (x, y)$ que se encuentren en el interior de la región R, ya que queremos maximizar el área de la *región de Voronoi* $Rv(p, N \cup \{p\}, R)$. En nuestro caso, dicha región R es el cuadrado de vértices $\{(0,0), (1,0), (1,1), (0,1)\}$. Así, consideraremos que el conjunto de configuraciones es infinito y que cada elemento, viene determinado por sus coordenadas (x, y) en el plano.

$$S = \{p_1 = (x_1, y_1), p_2 = (x_2, y_2), ..., p_n = (x_n, y_n), ...\} \tag{9.2.3}$$

Función de coste C:

La función de coste $C : S \to \Re$ asignará a cada punto $p_i \in S$, $(p_i \in R)$, un valor real igual al área de la *región de Voronoi* de p_i en el *diagrama de Voronoi* del conjunto de puntos $N \cup \{p\}$, siendo N el conjunto de entrada del problema MaxA-p-Vor(N). Como toda *región de Voronoi* es convexa, el cálculo de su área es inmediato utilizando la triangulación que se obtiene al unir el punto p_i con los vértices de la región. Así:

$$C(p_i) = Area(Rv(p, N \cup \{p\}, R) \tag{9.2.4}$$

La vecindad elegida para cada elemento de la configuración analizado será la misma a la utilizada en los capítulos anteriores para los problemas de iluminación, teniendo en cuenta que ahora cada elemento de S sólo debe verificar que pertenezca al interior de R.

Vecindad de cada configuración:

La función de vecindad debe encontrar el punto $p_i' = (x_i, y_i)$ a analizar tras $p_i = (x_i, y_i)$. Esta función calcula las coordenadas del punto p_i' sumando a cada coordenada de p_i un valor real que sigue la distribución $N(0,1)$.

$$\begin{cases} x_i' = x_i + N(0,1) \\ y_i' = y_i + N(0,1) \end{cases} \tag{9.2.5}$$

Para la obtención de números aleatorios que siguen la distribución $N(0,1)$, se ha utilizado también el método de Box-Muller [16], expuesto en la fórmula 7.2.4. Además, como todas nuestras experimentaciones están realizadas sobre *diagramas de Voronoi* de puntos generados aleatoriamente en el cuadrado unidad, es decir, en el cuadrado de vértices $\{(0,0),(1,0),(1,1),(0,1)\}$, si sumamos a cada coordenada del punto $p_i = (x_i, y_i)$ un valor que siga la distribución $N(0,1)$, el punto obtenido será un punto exterior al a dicho cuadrado con una probabilidad alta. Así, una vez generados los valores normales estos han sido divididos por un *factor de vecindad* `div`, (que irá aumentando con cada temperatura), con lo que el punto obtenido tiene una probabilidad más alta de ser interior. Hemos considerado en nuestro algoritmo `div` $= 10.0$. En forma de pseudocódigo la función que genera el vecino $p_i' = (x_i', y_i')$ al punto $p_i = (x_i, y_i)$, tiene la siguiente forma:

Función Generar-Vecino

ENTRADA: Una región R del plano, un punto $p_i = (x_i, y_i)$ y un factor de vecindad `div`.
SALIDA: Un punto $p_i' = (x_i', y_i')$ vecino de p_i e interior a R.

```
[01]  u1 ←rand()*1.0/RAND_MAX;
[02]  u2 ←rand()*1.0/RAND_MAX;
[03]  do
[04]    {n1 ←Sqrt(-2.0*ln(u1))*sin(2 * π * u2);
[05]     n2 ←Sqrt(-2.0*ln(u1))*cos(2 * π * u2);
[06]     x_i' = x_i + n1/div;
[07]     y_i' = x_i + n2/div;
[08]    }while (p_i' = (x_i', y_i') exterior a R);
[09]    div ← div/0.9999;
[10]    return p_i';
```

La configuración inicial y las estrategias de templado son simulares a las expuestas en el Capítulo 7 para el problema MaxA-p-Pv1(P), teniendo en cuenta que como funciones de templado se ha elegido la combinación de temperatura inicial y función de disminución que mejores resultados proporcionó en este capítulo.

Configuración inicial:

Nuestra heurística considera como configuración inicial un punto p_0 interior a R, (para nosotros el cuadrado unidad), que se considerará como primera solución a analizar y en su caso a iterar. La elección de este primer punto p_0 a estudiar se realiza de forma aleatoria, generando mediante la función `rand` sus dos cooordenadas.

En forma de pseudocódigo la función que genera la configuración inicial tiene la siguiente forma:

Función Generar-Configuracion Inicial

ENTRADA: Una región R del plano.
SALIDA: Un punto $p_0 = (x_0, y_0)$ interior a R.

```
[01]  do
[02]    {x0 ←rand()*1.0/RAND_MAX;
[03]     y0 ←rand()*1.0/RAND_MAX;
[04]    }while (p0 = (x0, y0) exterior a R);
[09]    return p0;
```

Estrategias de templado y criterio de parada

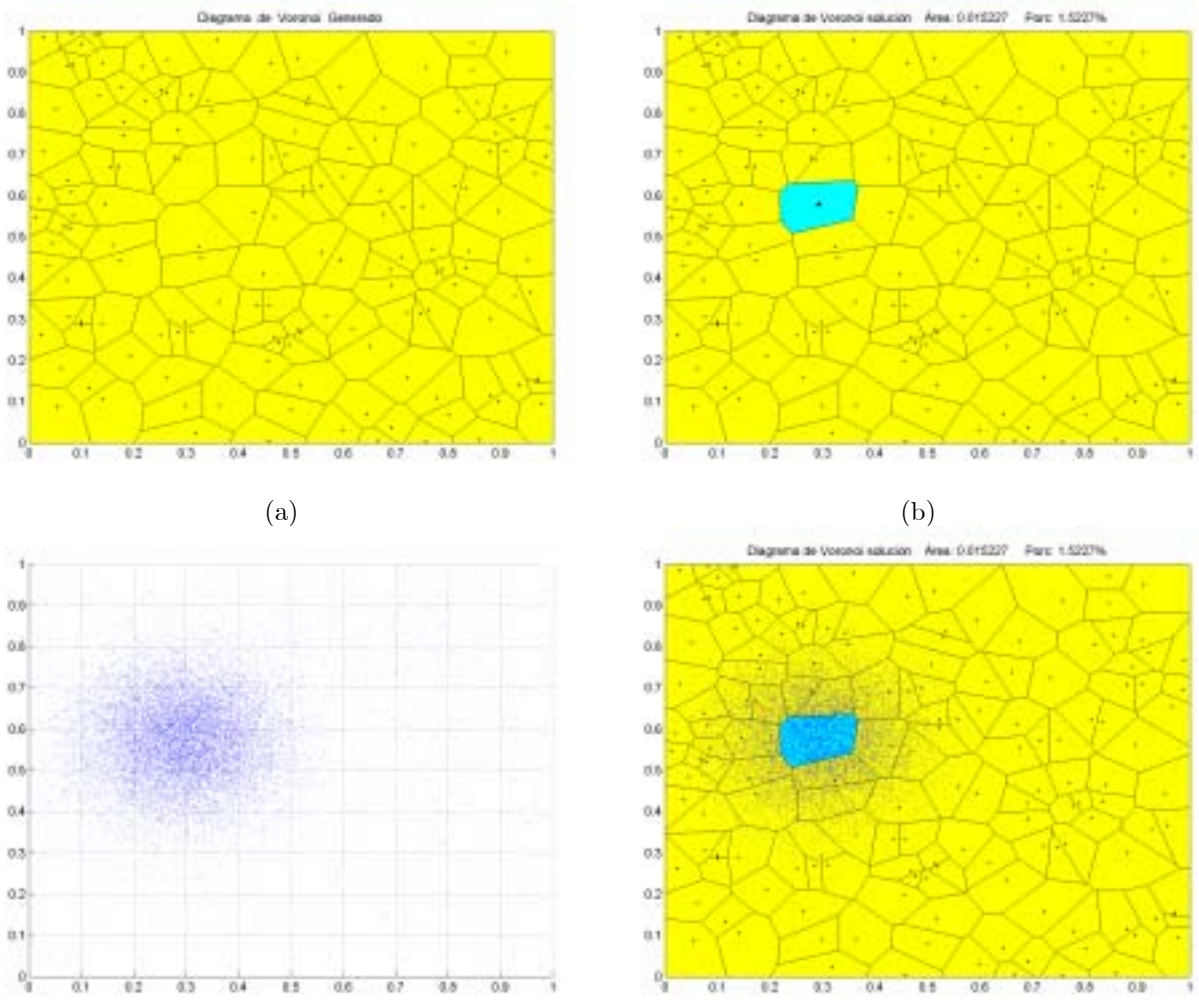
Las estrategias de templado y el criterio de parada determinan en gran medida la convergencia de la heurística para cada problema. Se han considerado los mismos criterios de templado que los elegidos para solucionar el problema MaxA-p-Pvk(P, k) en el capítulo anterior, pero adaptados a nuestro problema. Así, la temperatura inicial dependerá de la entrada de la heurística, $T_0 = n$, siendo n el cardinal del conjunto de puntos N, del que se tiene el *diagrama de Voronoi* como entrada de la heurística. Por otra parte, en cada iteración se ha elegido una disminución lineal de la temperatura en función del número de iteraciones, es decir, $T_i = \frac{T_0}{1+i}$, siendo $N(T) = \frac{1}{T}$ la función que determina este número de iteraciones para cada temperatura T.

Sin embargo, en el problema MaxA-p-Vor(N) observamos una rápida convergencia de la heurística hacia el valor óptimo, por lo que la condición de parada considerada ha sido $T_f \leq 0.025$, siendo $T_f \leq 0.005$, la elegida en los capítulos anteriores. De forma esquemática exponemos los criterios utilizados en la heurística:

- $T_0 = n$ (n cardinal del conjunto N).
- $T_i = \frac{T_0}{1+i}$ (i número de iteración).
- $N(T) = \frac{1}{T}$.
- $T_f \leq 0.025$.

Exponemos en la Figura 9.3 un ejemplo del resultado obtenido por SA, tomando como entrada de MaxA-p-Vor(N), el diagramd de Voronoi de 150 puntos de la Figura 9.1.

Presentamos en la figura (a) el diagrama de Voronoi inicial, en la figura (b) la solución aportada por SA, en la figura (c) la nube de puntos analizada y el la figura (d) todos los elementos. Como se puede observar la nube de puntos estudiada converge hacia la solución aportada por la heurística.



(a) (b)

(c) (d)

Figura 9.3: Un ejemplo de la aproximación del problema MaxA-p-Vor(N) con SA Card$(N) = 150$ $T_0 = 150$ $T(k) = \frac{T_0}{1+k}$ $T_f = 0.025$ $N(T) = \frac{1}{T}$

```
ALGORITMO SIMULATED ANNEALING SA T0=150
 * Región de Voronoi máxima calculada
 - Punto de área máxima........ p=(0.292940,0.579075)
 - Área máxima................. 0.015227 1.5227%
```

Pasamos a continuación a explicar los elementos de la heurística *random search-RS*, que

básicamente consistirá en realizar una búsqueda aleatoria sobre el conjunto infinito de configuraciones factibles del problema.

9.3 El problema MaxA-p-Vor(N) con random search-RS

Proponemos en esta sección una técnica heurística de búsqueda aleatoria o *random search-RS*, sobre el conjunto de todos los puntos interiores a la región R, (en las experimentaciones el cuadrado unidad), donde se encuentra la nube de puntos N, cuyo *diagrama de Voronoi* es la entrada de nuestro problema MaxA-p-Vor(N). Para ello se generará una nube $T = \{p_1, ..., p_m\}$ de m puntos interiores a la región R ,a la que nos restringimos, realizando una búsqueda de $p_i \in T$, tal que:

$$\acute{A}rea(Rv(p_i, T \cup \{p_i\}, R)) \geq \acute{A}rea(Rv(p_i, T \cup \{p_i\}, R)) \quad \forall j \neq i \quad i, j \in \{1, ..., m\} \qquad (9.3.6)$$

Necesitaremos por tanto, un mecanismo para generar la nube de puntos T. Este mecanismo, que exponemos en forma de pseudocódigo en la siguiente función, es igual a la generación de las nubes aleatorias diseñadas en capítulos anteriores para las técnicas RS.

Función Generar-Nube

ENTRADA: Una región poligonal plana R, y un entero m.
SALIDA: Un conjunto $T = \{p_1, ..., p_m\}$ de m puntos interiores a R.

```
[01]  i ←1;
[02]  T = {};
[03]  do
[04]    {do
[05]      {x ←rand()*1.0/RAND_MAX;
[06]       y ←rand()*1.0/RAND_MAX;
[07]      }while (p = (x, y) exterior a R);
[08]     p_i ← (x, y);
[09]     T ← T ∪ p_i;
[10]     i ← i + 1;
[11]    }while (i <= m);
[12]    return T;
```

Se ha considerado un tamaño de nube aleatoria $m = 50n$, siendo n el cardinal del conjunto de puntos N, que son entrada del problema. Con ello conseguimos que el tamaño de T esté relacionado con n, incrementando la búsqueda cuando el diagrama de Voronoi de entrada sea de mayor número de puntos. Evidentemente para cada punto $p_i \in T$, se debe calcular su *región de Voronoi* $Rv(p_i, T \cup \{p_i\}, R))$, utilizando para ello un paso del algoritmo incremental para el cálculo de *diagramas de Voronoi*, presentado en la Sección 9.1.1.

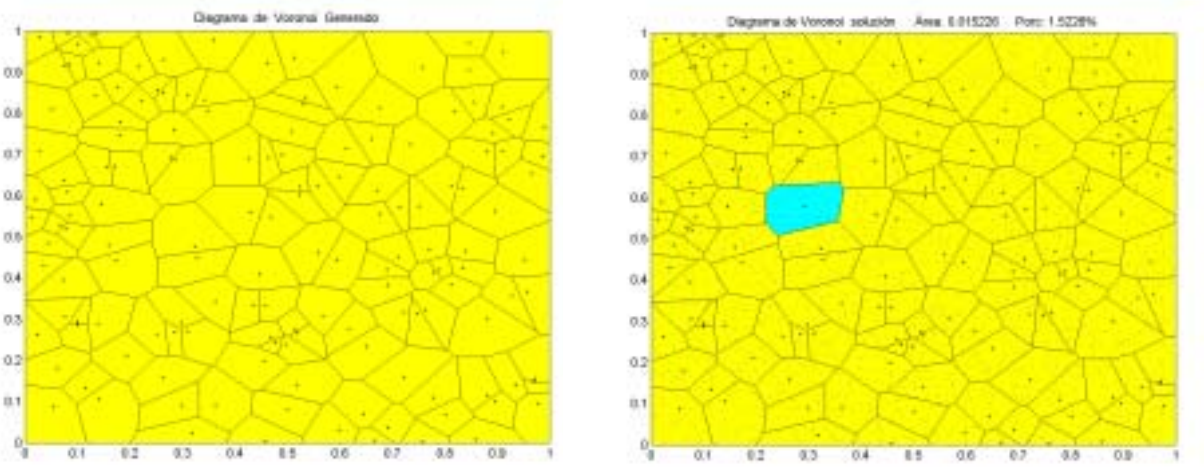
Así, el algoritmo RS propuesto lo podemos resumir de la siguiente manera:

Algoritmo RS-MaxA-p-Vor(N)

ENTRADA: Una región poligonal plana R y una nube $N = \{p_1, ..., p_n\}$ de puntos interiores..
SALIDA: Un punto p interior a R, tal que $Rv(p, N \cup \{p\}, R)$ tenga área máxima.

```
[01]  área ← 0;
[02]  T ← Generar_Nube(R, 50n);
[02]  Calcular_el_diagrama_de_Voronoi_Vor(N, R);
[03]  for (p_i ∈ T)
[08]    {Calcular la región de Voronoi Rv(p_i, N ∪ {p_i}, R);
[09]     if (área < Área(Rv(p_i, N ∪ {p_i}, R)))
[10]       área ← Área(Rv(p_i, N ∪ {p_i}, R));
[11]       p ← p_i;
[12]    }
[13]    return p;
```

Presentamos en la Figura 9.4 el resultado de aplicar RS al mismo conjunto de puntos N y por tanto al mismo diagrama de Voronoi inicial utilizado en el ejemplo de la heurística SA, en la Figura 9.3.



(a) (b)

Figura 9.4: Un ejemplo de la aproximación del problema MaxA-p-Vor(N) con RS $m = 50 \cdot 150 = 7500$

```
ALGORITMO RANDOM SEARCH RS m=7500:
 * Región de Voronoi máxima calculada
 - Punto de área máxima........ p=(0.2938332,0.583819)
 - Área máxima................. 0.015227 1.5227%
```

En este ejemplo podemos comprobar como SA obtiene un resultado mejor que RS, aunque muy parecido. Sin embargo, el esfuerzo en número de iteraciones de SA es mayor que RS, ya que por ejemplo en este caso, RS utiliza 7500 iteraciones frente a las 6629 de SA. En el siguiente apartado se exponen los resultados obtenidos por los tests comparativos realizados sobre ambas heurísticas, utilizando para ello un conjunto de *diagramas de Voronoi* generados aleatoriamente. Se presentan también las curvas de crecimiento sobre ejemplos concretos y las curvas de crecimiento medio sobre un conjunto de 50 *diagramas de Voronoi* aleatorios, pudiendo comparar de este modo el rendimiento de cada heurística.

9.4 Test comparativos

Siguiendo la misma idea de los test comparativos realizados en los capítulos anteriores, presentamos en la Tabla 9.1 y en la Tabla 9.2 el resultado de aplicar *simulated annealing-SA* y *random search-RS*, (con los parámetros que se han indicado anteriormente), sobre *diagramas de Voronoi* de 50, 100, 150 y 200 puntos generados aleatoriamente sobre el cuadrado unidad. Para cada número de puntos han sido generados a su vez 50 diagramas diferentes, aplicándoles ambas heurísticas y obteniendo el área de la *región de Voronoi* del punto solución, el tiempo de respuesta del método y el número de iteraciones o llamadas a la *función fitness* o *función de coste* utilizadas. Los resultados medios obtenidos son los siguientes:

- **Resultados** *simulated annealing-SA*

Puntos	% Área *región Voronoi*	Tiempo en seg.	Iteraciones
50	4.6257	5.28	2156
100	2.5271	31.84	4381
150	1.9191	96.4	6629
200	1.5203	218.58	8897

Tabla 9.1: Resultados simulated annealing-SA $T_0 = n \;\; T_i = \frac{T_0}{1+i}$ (VFA)

- **Resultados** *random search-RS.*

Puntos	% Área *región Voronoi*	Tiempo en seg.	Iteraciones
50	4.1845	5.96	2500
100	2.4519	35.64	5000
150	1.8676	111.36	7500
200	1.4709	246.92	10000

Tabla 9.2: Resultados random search-RS $m = 50n$

En la siguiente figura representamos gráficamente los resultados de la dos tablas anteriores. La coordenada x representa el número de puntos del *diagrama de Voronoi* de entrada, es decir,

el cardinal del conjunto N y la coordenada y la media del porcentaje, (respecto al cuadrado unidad R), de área de la *región de Voronoi* del punto solución aportado por cada heurística para MaxA-p-Vor(N).

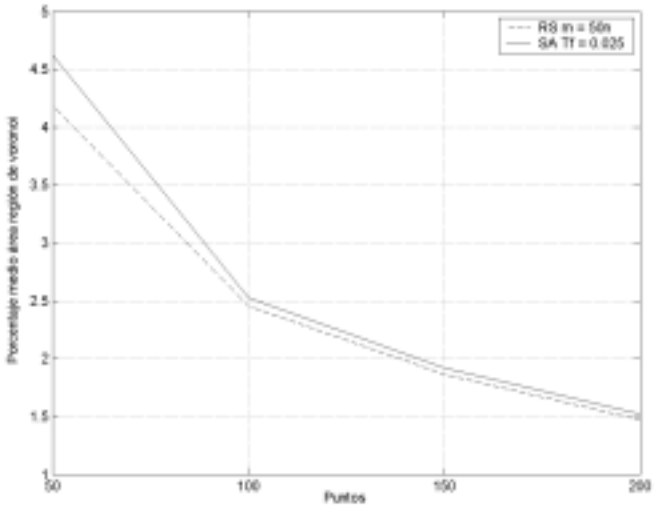


Figura 9.5: Resultados para SA $T_0 = n$ $T_i = \frac{T_0}{1+i}$ frente a RS

Como podemos comprobar SA obtiene mejores resultados en todos los parámetros: área de la *región de Voronoi*, tiempo de respuesta y número de iteraciones. Además, esta diferencia se hace más notable para un número pequeño de puntos del *diagrama de Voronoi* inicial.

Contraste de hipótesis

Observando la gráfica anterior, parece que el contraste podría ser negativo a medida que aumenta el número de puntos n. Por ello, se han realizado contrastes T, (utilizando el software matemático MatLab), con un nivel de significación del 95%, utilizando los resultados obtenidos sobre los 50 *diagramas de Voronoi* de 200 puntos, cuyos medias se encuentran en las últimas filas de las tablas anteriores. Las respuestas aportadas por los contrastes realizados se presentan en la siguiente tabla, indicando con el signo '+' si el contraste es significativo y con '–' si no lo es, y como podemos apreciar los resultados obtenidos son significativamente diferentes.

Heurística	*Simulated annealing-SA*	*Random Search-RS*
Simulated annealing-SA	•	+
Random Search-RS	+	•

Tabla 9.3: Tabla de contrastes realizados

Para el resto de puntos analizados el contraste sigue siendo positivo. Presentamos a continuación las *curvas de crecimiento*, es decir, las curvas que refleja las respuestas o aproximación de las heurísticas en cada iteración, de tal forma que podamos obtener una comparativa más clara respecto a la rapidez de convergencia de cada método.

Curvas de crecimiento

Las Figuras 9.6 y 9.7 muestran las *curvas de crecimiento* de SA y RS, aplicados al *diagrama de Voronoi* de la Figura 9.1, cuyas soluciones se presentan en las Figuras 9.3 y 9.4 respectivamente.

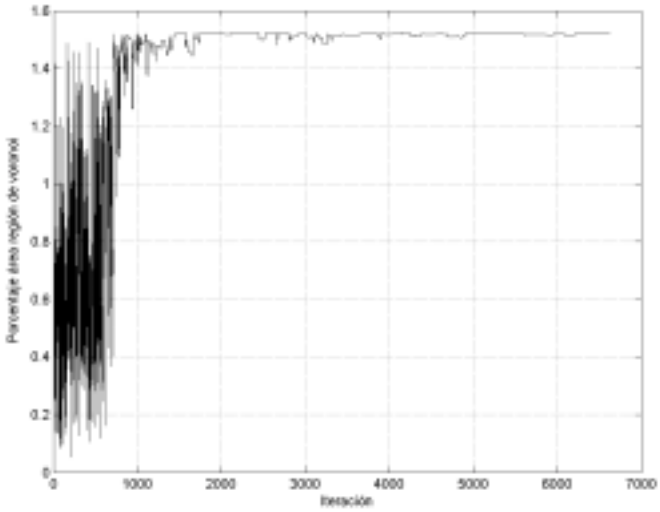


Figura 9.6: Curva de crecimiento para el problema MaxA-p-Vor(N) con SA

La siguiente figura muestra la *curva de crecimiento* para RS, tomando el mismo número de iteraciones que las utilizadas por SA..

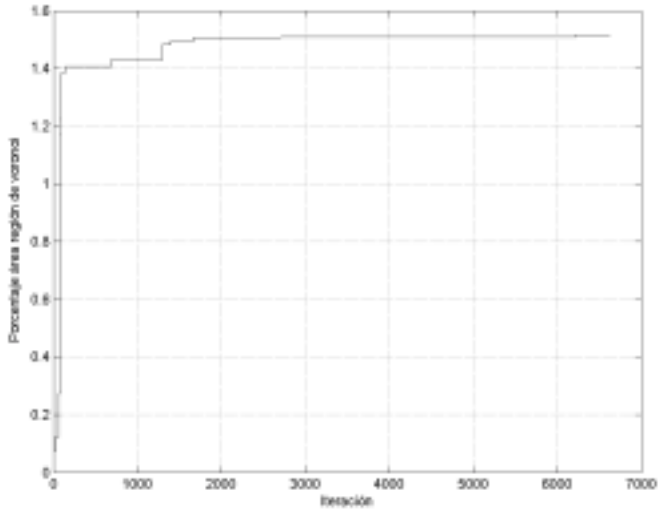


Figura 9.7: Curva de crecimiento para el problema MaxA-p-Vor(N) con RS

Podemos observar como RS converge más rápidamente, aunque obtiene en media peores resultados. Será importante saber, sin embargo, si esta convergencia rápida que se observa en RS para este ejemplo, será una característica de la heurística. Para comprobarlo se han relizado

las medias, (iteración a iteración), de todas las curvas de crecimiento obtenidas al aplicar cada una de las heurísticas sobre 50 *diagramas de Voronoi* generados aleatoriamente, de 100 puntos cada uno de ellos, (los utilizados para obtener las tablas de resultados). Las curvas obtenidas se representan en la siguiente figura:

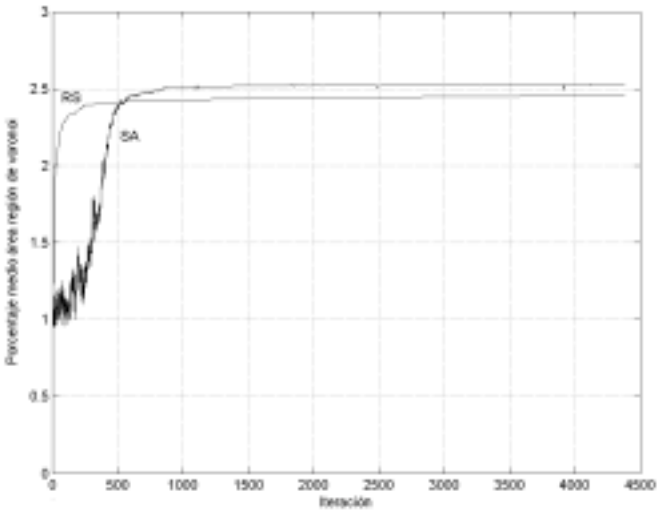


Figura 9.8: Curva de crecimiento medio para diagramas de Voronoi de 100 puntos

Como podemos observar la convergencia de RS es más rápida que la aportada por SA, pero sin embargo RS obtiene en media peores resultados. Por tanto, podemos obtener las siguientes conclusiones respecto a la comparativa de las dos heurísticas:

- En general *simulated annealing-SA* obtiene mejores soluciones para el problema MaxA-p-Vor(N) que *random search-RS*, tanto en área de la *región de Voronoi* del punto solución, como en tiempos de respuesta y en número de iteraciones.
- Con mayor número de iteraciones *random search-RS* obtiene peores medias en área de *región de Voronoi* del punto solución que *simulated annealing-SA*.
- La heurística *random search-RS* converge más rápidamente que *simulated annealing-SA* hacia la solución, por tanto será útil cuando deseemos obtener una respuesta al problema MaxA-p-Vor(N) en un número pequeño de iteraciones, aunque perdamos optimalidad de la respuesta.

9.5 Conclusiones y trabajos futuros

Se han presentado en este capítulo soluciones heurísticas al problema de la búsqueda de un punto p en una región poligonal R, (en el estudio experimental el cuadrado unidad), tal que el área de la *región de Voronoi* $Rv(p, N \cup \{p\}, R)$ sea máxima, siendo $N = \{p_1, ..., p_n\}$ una nube de puntos de la que se tiene su *diagrama de Voronoi* $Vor(N, R)$ como entrada del problema. La

primera de las heurísticas sigue una técnica *simulated annealing* y la segunda un mecanismo de búsqueda aleaotoria o *random search*. En media el primer método obtiene mejores resultados, aunque la convergencia de una búsqueda aleatoria es más rápida. Este hecho es debido a que SA realiza una búsqueda sobre conjuntos de puntos alejados del óptimo cuando la temperatura es alta, (en las primeras iteraciones), pero posteriormente focaliza mejor la zona de búsqueda, obteniendo mejores respuestas medias.

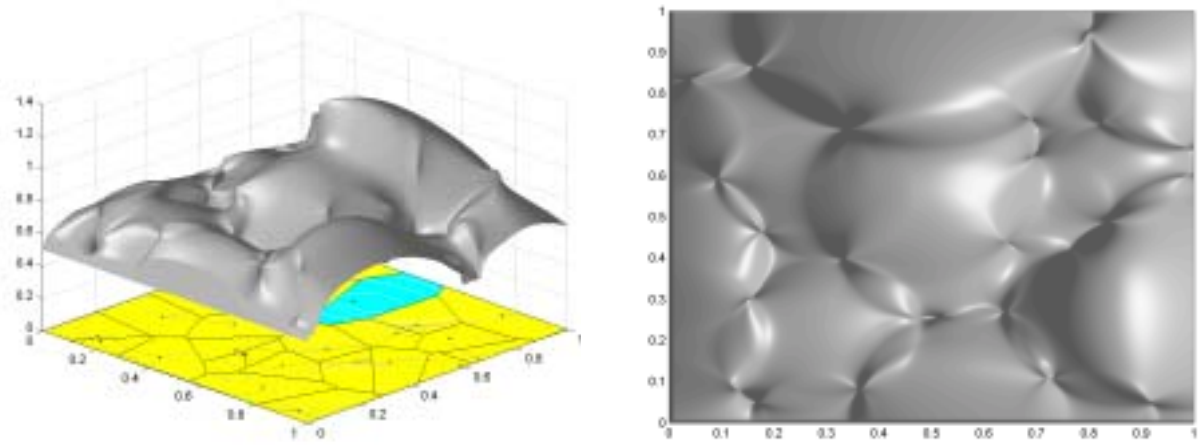
Siguendo el mismo esquema de capítulos anteriores presentamos en la siguiente tabla los problemas tratados y las soluciones aportadas:

Problema Estudiado	*Solución Aportada*	*Parámetros*
MaxA-p-Vor(N)	Aprox. *Simulated Annealing-SA*	$T_0 = n \;\; T_i = \frac{T_0}{1+i} \;\; (FSA) \;\; T_f \leq 0.025$
	Aprox. *Random Search-RS*	$m = 50n$

Tabla 9.4: Técnicas diseñadas para el problema MaxA-p-Vor(N)

Queda pendiente como trabajo futuro el estudio del comportamiento de otras técnicas heurísticas, como por ejemplo los *algoritmos genéticos* sobre el problema MaxA-p-Vor(N).

Por otra parte, siguiendo la misma idea de la *superficie de áreas* presentada en el Capítulo 7 para un polígono aleatorio P, podemos construir lo que llamaremos la *superficie de Voronoi*: a cada punto $p = (x, y)$ perteneciente a la región R, donde tenemos el *diagrama de Voronoi* de



(a) (b)

Figura 9.9: Un ejemplo de superficie de Voronoi (a) y su planta (b)

una nube de puntos N, $Vor(N, R)$, le asignaremos una coordenada z, cuyo valor será igual al área de la *región de Voronoi* $Rv(p, N \cup \{p\}, R)$, generando una superficie que ilustra el área de dichas regiones para puntos introducidos en $Vor(N, R)$. En la Figura 9.9 mostramos la vista y la planta de una *superficie de Voronoi*, correspondiente a un *diagrama de Voronoi* de 25

puntos aleatorios, incluyendo en la parte inferior de la superficie la solución aportada por SA. En la figura (a) podemos apreciar como la superficie se eleva alrededor del óptimo aportado por la heurística y en la figura (b), podemos intuir que las circunferencias que cincunscriben a los triángulos de la *triangulación de Delaunay* asociada al *diagrama de Voronoi*, juegan un papel importante en la solución del problema MaxA-p-Vor(N). El estudio detallado de estas superficies podría ser también motivo de futuras investigaciones.

Por otra parte, una ampliación interesante de este problema que podrá ser motivo de futuros trabajos es la búsqueda no de un sólo punto, sino de k puntos en un *diagrama de Voronoi* dado, tal que la suma de las áreas de las *regiones de Voronoi* de estos k puntos se máxima. Una posible solución de esta variante podría utilizar técnicas *greedy, (*ver Capítulo 1), basadas en las soluciones aportadas para MaxA-p-Vor(N) en cada punto.

Todas las implementaciones utilizadas en este capítulo, tanto para la generación aleatoria de *diagramas de Voronoi*, como las correspondientes a las heurísticas diseñadas, se encuentran en el Apéndice B de listados. Por motivos de espacio, no se ha incluido sin embargo, el código implementado para la obtención de las *superfices de Voronoi*.

Apéndice A

Sobre la Generación Aleatoria de Polígonos

La generación aleatoria en experimentos computacionales es una faceta muy estudiada por los investigadores. Realmente, el concepto aleatorio en una máquina se debe entende como pseudoaleatorio, ya que todo elemento generado por un ordenador contiene de alguna manera una componente no aleatoria. En particular, la generación de *polígonos aleatorios* es necesaria cuando se desean contrastar algoritmos en los que dicho polígonos juegan un papel importante en la entrada del problema, como son las entradas de las heurísticas diseñadas para solucionar nuestros problemas MaxA-p-Pv1(P), MaxA-p-Pvk(P, k) y MinN-p-Pvk(P). Existen en la bibliografía numerosos estudios sobre este tema ([6, 38, 81]), que muestran diferentes técnicas de generación aleatoria de polígonos contrastando sus ventajas e inconveniente. Dado que para realizar nuestras experimentaciones se necesita generar polígonos aleatoriamente se ha optado por elegir un generador presentado por Auer y Held en [6], denominado "Steady Growth", ("Crecimiento Constante"). Presentamos brevemente en este apéndice la lineas generales de este método de generación aleatoria de polígonos. Además, este generador que denotaremos con RPG, ha sido implementado expresamente en esta memoria y su código en lenguaje C se puede encontrar en el Apéndice B.

El problema de la *generación aleatoria de polígonos* es un problema importante cuando se desea contrastar el consumo real de algoritmos que tienen como entrada un conjunto de polígonos P.

Concretamente consideremos que tenemos un conjunto $C = \{v_1, ..., v_n\}$ de n puntos en el plano generados uniformente y queremos construir un polígono simple P cuyos vértices sean los puntos de C. En este contexto el polígono P se podrá generar con probabilidad $1/k$ si existen k polígonos simples diferentes que puedan tener a C como conjunto de vértices. En la implementación diseñada en esta memoria el conjunto C se genera utilizando la función `rand()`de C++, perteneciente a `stdlib.h`, que genera aleatoriamente, (pseudoaleatoriamente), puntos con distribución uniforme. Además en nuestro caso el conjunto de puntos C y por tanto el polígono

P, estará generado siempre sobre el cuadrado unidad, es decir sobre el cuadrado de vértices $\{(0,0),(1,0),(1,1),(0,1)\}$.

En forma de pseudocódigo la función que produce el conjunto C se puede expresar de la siguiente manera:

Función Generar-Conjunto Inicial

ENTRADA: Un número entero positivo n.
SALIDA: Un conjunto $C = \{v_1, ..., v_n\}$ de puntos uniformemente distribuidos en el cuadrado unidad..

```
[01]  i ← 1;
[02]  C ← {};
[03]   while (i <=n)
[04]   {x ←rand()*1.0/RAND_MAX;
[05]    y ←rand()*1.0/RAND_MAX;
[06]    v_i ← (x, y);
[07]    C ← C ∪ v_i;
[08]   }
[09]   return C;
```

El método "Steady Growth" o de "Crecimiento Constante", (por tanto nuestro generador RPG), presentado por Auer es un método incremental que parte de un terna aleatoria de puntos del conjunto C y por tanto de un triángulo, que no contiene a ningún otro punto del conjunto en su interior. En cada iteración i con $1 \leq i \leq n-3$, se construye un polígono P_i que añade un punto más de C entre sus vértices. Dividimos el método en dos pasos: inicialización y proceso.

1. **Inicializacion** Seleccionar aleatoriamente tres puntos $s_1, s_2, s_3 \in C$, (es decir representan tres puntos cualesquiera del conjunto no necesariamente los tres primeros), tal que ningún otro punto de C está en $CH(\{s_1, s_2, s_3\})$, ($CH(T)$ representa el *cierre convexo* de un conjunto de puntos T). Llamemos $C_1 = C \setminus \{s_1, s_2, s_3\}$ y $P_1 = CH(\{s_1, s_2, s_3\})$.

2. **Proceso** Para cada iteración i con $1 \leq i \leq n-3$, realizar las siguientes tareas:

   (a) Tomar aleatoriamente un punto $s_i \in C_i$ tal que ningún punto de $C_{i+1} = C_i \setminus \{s_i\}$, está contenido en $CH(P_{i-1} \cup \{s_i\})$.

   (b) Buscar una arista (v_k, v_{k+1}) de P_{i-1} que sea completamente visible desde s_i, reemplazándola por las aristas (v_k, s_i) y (s_i, v_{k+1}) para construir P_i.

Es importante destacar que el punto $s_i \in C_i$ necesario en el Paso a siempre existe, pues es suficiente tomar el punto más cercano a $CH(P_{i-1})$. No ocurre lo mismo en el Paso b, pues según se ilustra en la Figura A.1 puede darse el caso de que no existe ninguna arista de P que sea completamente visible por el punto p.

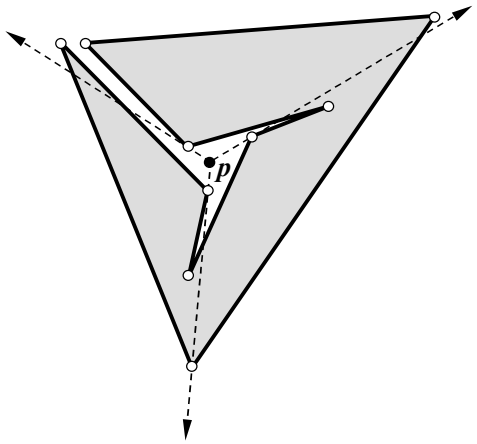


Figura A.1: El punto p no ve ninguna arista del polígono

Sin embargo, si p está fuera de $CH(P)$ existe al menos una arista que es completamente visible desde p. Esta propiedad se puede probar fácilmente por inducción como se muestra en [6]. EL procedimiento "Steady Growth" puede construir un polígono simple de n vértices en tiempo $O(n^2)$, ya que en cada paso se deben computar el conjunto de aristas visibles, (esto puede hacerse en tiempo lineal como probaron Joe and Simpson [63]).

Este Apéndice debe considerarse como un pequeño resumen del método de *generación aleatoria de polígonos* implementado en esta memoria para la obtención de resultados sobre los métodos heurístico diseñados. Para el estudio más detallado, tanto de "Steady Growth" como de otras técnicas puede verse [6].

Apéndice B

Comentarios de Implementación

Presentamos en este apéndice los códigos en lenguaje C de las implementaciones realizadas para el estudio de todas nuestras heurísticas. Exponemos en primer lugar los tipos, las estructuras de datos y el código de nuestro generador aleatorio de polígonos *RPG,* analizado en el Apéndice B. A continuación se presentan los códigos de los métodos con la misma secuencialidad con la que aparecen y se analizan en el transcurso de esta memoria. Es importante destacar que para intentar reducir la extensión de este apéndice sólamente se incluyen los códigos más importantes que determinan el seguimiento de la heurísticas, omitiendo las implementaciones de funciones auxiliares o secundarias.

B.1 Introducción

Respecto a las secciones en las que exponemos los códigos que solucionan nuestos problemas, se presentan los códigos de todas las heurísticas implementadas para cada problema. Además se presentan también los códigos realizados para la obtención de datos o resultados sobre conjuntos aleatorios de polígonos, que nos han permitido obtener las conclusiones de los estudios experimentales expuestos en los capítulos anteriores. Finalmente se presentan también los ficheros `.m` realizados con el software MatLab, que permitan visualizar de forma gráfica todos los datos y resultados de nuestros métodos, tales como polígonos, puntos, zonas visibles, etc.. Todos los códigos y programas ejecutables se encuentran en el CD-ROM adjunto a esta memoria.

En forma esquemática los códigos de este apéndice se presentan con la siguiente secuencialidad.

1. **B.2:** Estructuras, tipos de datos y prototipos de función.
2. **B.3:** Generardor aleatorio de polígonos *RPG*, según lo descrito en el Apéndice B.
3. **B.4:** Métodos aproximados para el problema MaxA-p-Pv1(P).
   - Implementaciones basadas en la heurística *simulated annealing-SA*.

- Implementaciones para la heurística *random search-RS*.
- Implementaciones para la heurística *gradiente-GRAD*.
- Implementaciones para el estudio de la descomposición $\mathcal{S}$ de un polígono P.
- Códigos para obtener datos sobre el problema PorA-p-Pv1(P).
- Código para la construcción de la *superficie de áreas* de un polígono.

4. **B.5:** Implementaciones relacionadas con MaxA-p-Pvk(P, k).

   - Unión de polígonos de visibilidad.
   - Implementaciones basadas en la heurística *simulated annealing-SA*.
   - Implementaciones con *algoritmos genéticos-AG*.
   - Implementaciones para la heurística *random search-RS*.

5. **B.6:** Códigos relacionados con el problema MinN-p-Pvk(P).

6. **B.7:** Implementaciones para MaxA-p-Vor(N).

   - Tipos de datos adicionales.
   - Construcción del *diagrama de Voronoi*.
   - Métodos aproximados para el problema MaxA-p-Vor(N).
     - Implementaciones basadas en *simulated annealing-SA*.
     - Implementaciones con *random search-RS*.

7. **B.8:** Funciones de cálculos generales y programa principal.

8. **B.9:** Subrutinas de visualización de datos con MatLab

B.2 Estructuras y tipos de datos

Presentamos a continuación en forma de código los tipos de datos necesarios en nuestras implementaciones. Estos tipos recogen todos los tipos de datos que permiten el manejo de nuestros problemas. Entre los más importantes podemos destacar: puntos, segmentos, polígonos simples y polígonos con agujeros.

Tipos de datos

Los tipos de datos se exponen de la misma manera que se encuentran en el código fuente. Se ha intentado que el nombre de todas las estructuras de datos sean lo suficientemente explícitos para que se identifique al elemento geométrico que representa.

```
/*-------------------------------------------------------------------------------------
Incluimos aquí la declaración de un punto como una estructura
-------------------------------------------------------------------------------------*/

typedef struct s
{double x;
 double y;
 char tipo;
}punto;

/*-------------------------------------------------------------------------------------
Ahora definimos dos listas enlazadas de puntos. Este tipo de datos nos  permitirá definir
un polígono como una lista de puntos. En nuestro caso  un polígono será una lista de pun-
tos ordenados en sentido antihorario.
-------------------------------------------------------------------------------------*/

struct lista_elementos
{punto p;
 struct lista_elementos *sig;
 struct lista_elementos *ant;
};

typedef struct lista_elementos lista;

/*-------------------------------------------------------------------------------------
Definimos ahora el tipo polígono con agujeros. Será necesario  para la implementación del
algoritmo de unión de polígonos  de Weiler-Atherton. Un polígono en general se  considera
una lista de nodos. Cada nodo  será un polígono con un tipo asociado. El tipo  de un nodo
podrá ser 'P', (principal), ó 'A', (agujeros). Realmente  se entiende un polígono como un
conjunto de polígonos principales tal que cada uno puede tener o no agujeros.
-------------------------------------------------------------------------------------*/

typedef struct
{char tipo;                              //El tipo podrá ser agujero 'A' o prinipal 'P'
 lista *puntos;
}nodo;

struct lista_elementos3
{nodo dat; //polígono principal
 struct lista_elementos3 *sig;    //agujeros
};

typedef struct lista_elementos3 poligono;

/*-------------------------------------------------------------------------------------
Generalizamos también el concepto de individuo. Un individuok es una lista de k puntos.
-------------------------------------------------------------------------------------*/


typedef lista individuok;
```

Incluimos a continuación los prototipos de función de las subrutinas y funciones más importantes. Algunas de las funciones que se implementan en el código no necesitan su prototipo debido al orden en que aparecen implementadas, por lo que se omiten en este listado.

```
/*-------------------------------------------------------------------------------------
Incluimos aquí los prototipos de función
-------------------------------------------------------------------------------------*/

double vectorial(punto p1,punto p2,punto p3);
bool cortan(punto p1,punto p2, punto p3, punto p4);
bool verificar(punto pu,lista *pol, lista *au,unsigned long int a,unsigned long int b);
void creapoligono(lista *polig,unsigned long int n);
bool Es_Ordenada_vertice(punto punt,lista *polig);
int cortes(punto punt, lista *polig);
void creanube(lista *nube,lista *polig,unsigned long int n);
void ver(lista *p);
void Conjunto_Inicial(lista *polig, unsigned long int n);
void Calcular_triangulo(lista *conjunto,lista *triang);
double angulo(punto p1,punto p2, punto p3, punto p4);
lista *Cierre_Convexo(lista *conjunto, lista *micierre);
int Dentro(lista *poligono, lista *quedan);
int cortes_bis(punto p,punto q,lista *poligono);
bool Iguales(punto p1, punto p2);
```

```
bool Pertenece(punto p,lista *polig);
void RPG(lista *poligono, lista *conjunto);
void RPG_MONOTONO(lista *poligono, lista *conjunto);
long int longitud(lista *poligono);
bool es_oreja(punto p1, punto p2, punto p3,lista *poligono);
double area(punto p1,punto p2, punto p3);
double area_poligono(lista *poligono);
double modulo(punto p, punto q);
bool cortanrectas(punto r1,punto r2,punto q1, punto q2, double *x,double *y);
lista *calpolvisi(punto p, lista *poligono, lista *polvisi);
double areapolvisi(punto p,lista *poligono);
void Menu1(int *op);
void Menu2(int *op);
void Menu3(int *op);
lista *MaxDet_PV1(lista *nube,lista *poligono,lista *polvisi,punto *p, double *a,double ar);
punto generapunto(lista *polig);
lista *MaxRsSV_PV1(double tpi,double dt,lista *poligono,lista *polvisi,punto *p,double *a,double ar);
void elegir1(lista *hormigas,listah *mishormigas,lista *poligono,double cir,punto *h,double *fh);
void elegir2(lista *hormigas,listah *mishormigas,lista *poligono,double cir,punto *h,double *fh);
void elegir3(lista *hormigas,listah *mishormigas,lista *poligono,double cir,punto *h,double *fh);
void actualizarferomona(listah *mishormigas,punto h,double fh,double red);
lista *MaxAnts_PV1(unsigned long antsini,unsigned long int ants,unsigned long int iter,double red,
                   double cir,lista*poligono,lista *polvisi,punto *p,double *a,double ar);
bool punto_int(punto p, lista *poligono);
bool Incluido(lista *pol1, lista *pol2);
lista *Anadir(lista *l,punto p);
lista *InSeg(punto p1,punto p2,lista *tri);
lista *unelistas(lista *l1,lista *l2);
bool VerticesDentro(lista *tri1,lista *tri2);
double area_poligonobis(lista *poligono);
double AreaInterPolPol(punto p1,lista *polvisi1,punto p2,lista *polvisi2);
void Resultados_MaxDet_PV1(void);
bool Poli_Correcto(lista *poligono);
lista *Limpiar(lista *polig);
lista *lista_cortes_total(lista *solu,lista *poligono1, lista *poligono2);
double area_poligonoagujeros(poligono *polig);
lista *desbrozar (lista *polig);
lista* contrastar(lista *l1,lista*l2,char c);
lista *eliminar(punto p,lista *l);
bool valido(punto p,lista *previa,lista *polig);
poligono *LimpiarSolucionRS(poligono *polvisi,lista *polig,lista *ps,double a,double *area);
```

Según se muestra en el Apéndice B sobre la generación aleatoria de polígonos, ninguno de los métodos es totalmente aleatorio, ya que la aleatoriedad real se transforma en pseudoaleatoriedad cuando hablamos de codificación. Presentamos en el siguiente punto el código implementado para nuestro generador aleatorio de polígonos *RPG*. Como este generador precisa de otros algoritmos tales como el cálculo del *cierre convexo*, (CH), se detallan también las implementaciones de los algoritmos en cada caso.

B.3 Generador aleatorio de polígonos RPG

El algoritmo elegido para el cálculo del *cierre convexo*, (CH), ha sido *Scan de Graham*, (ver [82]), debido a su eficiencia y rapidez de cálculo, (su complejidad es $O(n \log n)$, siendo n el número de puntos). La implementación construida para este algoritmo es la siguiente:

Cierre convexo, (CH)

En el propio código se explican con comentarios cada uno de los pasos seguidos.

```
lista *Cierre_Convexo(lista *conjunto, lista *micierre)
{lista *aux,*ymin,*aux1;
 double angulos[Tamcierre];
 punto cierre[Tamcierre];
 double ang,temp;
 punto p1,p2,p3;
 int i,n,m;
                    //Copiamos la lista en el cierre
 i=0;
 aux=conjunto;
 while(aux!=NULL)
 {cierre[i]=aux->p;
  aux=aux->sig;
  ++i;
 }
                  //Elegimos un punto como el baricentro formado por los tres primeros puntos
 aux=conjunto;
 p1.x=(conjunto->p.x+conjunto->sig->p.x+conjunto->sig->sig->p.x)/3;
 p1.y=(conjunto->p.y+conjunto->sig->p.y+conjunto->sig->sig->p.y)/3;
 p2.x=1;
 p2.y=p1.y;
                    //Construimos el vector de ángulos
 aux=conjunto;
 i=0;
 while(aux!=NULL)
 {ang=angulo(p1,p2,p1,aux->p);
  if(aux->p.y<p1.y)
   angulos[i]=(2*pi)-ang;
  else
   angulos[i]=ang;
  if(aux->sig!=NULL) ++i;
  aux=aux->sig;
 }
                    //Ordenamos el cierre y el vector de ángulos de menor a mayor
 n=0;
 while(n<i)
 {m=n+1;
  while(m<=i)
  {if(angulos[m]<angulos[n])
   {temp=angulos[n];
    angulos[n]=angulos[m];
    angulos[m]=temp;
    p3=cierre[n];
    cierre[n]=cierre[m];
    cierre[m]=p3;
   }++m;
  }++n;
 }
                    //Lo guardamos en una lista circular
 micierre->p.x=cierre[0].x;
 micierre->p.y=cierre[0].y;
 n=1;
 aux=micierre;
 while(n<=i)
 {aux->sig=(lista*)malloc(sizeof(lista));
  aux=aux->sig;
  aux->p=cierre[n];
  ++n;
 }
 aux->sig=micierre;
                    //Buscamos la ordenada mínima
 n=0;
 aux=micierre;
 ymin=micierre;
 while (n<=i)
 {if(aux->p.y<ymin->p.y)
   ymin=aux;
  aux=aux->sig;
  ++n;
 }
 aux=ymin;
                    //Vamos recorriendo con productos vectoriales
 Otravez:
 while(vectorial(aux->sig->p,aux->p,aux->sig->sig->p)>=0.0)
 {aux1=aux->sig;
  aux->sig=aux->sig->sig;
  free(aux1);}
  aux=aux->sig;
```

```
  aux->ant=ymin;
  while(aux!=ymin)
  {if(vectorial(aux->sig->p,aux->p,aux->sig->sig->p)<0.0)
   {aux1=aux;
    aux=aux->sig;
    aux->ant=aux1;
   }
   else
   {aux1=aux->sig;
    aux->sig=aux->sig->sig;
    free(aux1);
    aux=aux->ant;
    if (aux==ymin)
    goto Otravez;
   }
  }
 aux=ymin->sig;
 while(aux!=ymin)
 {aux1=aux;
  aux=aux->sig;
 }
 aux1->sig=NULL;
 micierre=ymin;
 return(micierre);
}                  //fin Cierre_Convexo
```

La codificación del generador aleatorio de polígonos junto con todas las funciones auxiliares necesarias se presenta a continuación. Este generador ha sido utilizado para la realización de todas las experimentaciones sobre polígonos en nuestras heurísticas. Como se puede observar en la codificación se ha intentado buscar la mayor aleatoriedad posible. En este sentido la semilla de generación aleatoria producida por la función `srand()` del lenguaje C++, se ha hecho depender de una función que depende del tiempo de compuatadora.

Generador aleatorio de polígonos RPG

Como se mencionó en el Apéndice B se ha implementado para nuestros estudios un generador aleatorio de polígonos, (RPG), basado en los estudios que Auer y Held presentaron en 1996, [6]. Se incluye, además de la codificación de nuestro generador, las implementaciones de todas las funciones auxiliares a éste, indicando al inicio de las más importantes un comentario sobre su función.

```
/*-------------------------------------------------------------------------------------
Diseñamos una primera función que genera una nube de puntos en  nuestro cuadrado  unidad,
es decir, el cuadrado de  vértice (0,0),(1,0),(1,1) y (0,1). La nube se mantiene mediante
una lista enlazada. El conjunto inicial generado se guarda además  en un fichero de datos
para su visualización posterior  con los correspondientes programas de MatLab,  expuestos
en la Sección B.9 de la página 285.
-------------------------------------------------------------------------------------*/

void Conjunto_Inicial(lista *polig, unsigned long int n)
{unsigned long int i;
 lista *aux;
 punto punt;
 FILE *conjunto;                  //Creo un fichero de texto para guardar el polígono

 conjunto=fopen(''c:\\Pruebas_Heur_P1\\Matlab\\conjunto.txt'',''w+'');

                                  //se genera un punto
 punt.x=(1.0*rand())/(RAND_MAX);
 punt.y=(1.0*rand())/(RAND_MAX);
                                  //se guarda en la lista y en el fichero de datos
 polig->p.x=punt.x;
 polig->p.y=punt.y;
 polig->p.tipo='v';
 fprintf(conjunto,''%f %f\n'',punt.x,punt.y);
```

```
 aux=polig;
 punt.x=(1.0*rand())/(RAND_MAX); //se genera un punto
 punt.y=(1.0*rand())/(RAND_MAX); //se guarda en la lista
 aux->sig=(lista*)malloc(sizeof(lista));
 aux=aux->sig;
 aux->p.x=punt.x;
 aux->p.y=punt.y;
 aux->p.tipo='v';
 fprintf(conjunto,''%f %f\n'',punt.x,punt.y);
 punt.x=(1.0*rand())/(RAND_MAX); //se genera un punto
 punt.y=(1.0*rand())/(RAND_MAX); //se guarda en la lista
 aux->sig=(lista*)malloc(sizeof(lista));
 aux=aux->sig;
 aux->p.x=punt.x;
 aux->p.y=punt.y;
 aux->sig=NULL;
 fprintf(conjunto,''%f %f\n'',punt.x,punt.y);

                                  //ya tenemos generados los tres primeros puntos.
                                  //todo polígono tiene como mínimo 3 vértices
 i=4;
 while(i<=n)
 {punt.x=(1.0*rand())/(RAND_MAX);
  punt.y=(1.0*rand())/(RAND_MAX);
  if(i==n)
  {aux->sig=(lista*)malloc(sizeof(lista));
   aux=aux->sig;
   aux->p.x=punt.x;
   aux->p.y=punt.y;
   aux->sig=NULL;
   fprintf(conjunto,''%f %f\n'',punt.x,punt.y);

                                  //se guarda el punto en el fichero
  }
  else
  {aux->sig=(lista*)malloc(sizeof(lista));
   aux=aux->sig;
   aux->p.x=punt.x;
   aux->p.y=punt.y;
   fprintf(conjunto,''%f %f\n'',punt.x,punt.y);

                                  //se guarda el punto en el fichero
  }++i;
 }
fclose(conjunto);
}/*fin Conjunto_Inicial*/

/*-------------------------------------------------------------------------------------
Buscamos el triángulo inicial vacio de puntos,que nos permita comenzar la generación. Los
vértices de este triángulo no  pueden estar alineados y deben estar orientados en sentido
positivo para conseguir que el polígono generado P tenga este mismo sentido.
-------------------------------------------------------------------------------------*/

void Calcular_triangulo(lista *conjunto,lista *triang)
{lista *aux1, *aux2, *aux3,*punto;
 int cor;
 bool fin=false;
                                  //Vamos estudiando todos los trios de punteros posibles
 aux1=conjunto;
 while((aux1!=NULL)&&(fin==false))
 {aux2=aux1->sig;
  while((aux2!=NULL)&&(fin==false))
  {aux3=aux2->sig;
   while((aux3!=NULL)&&(fin==false))
   {if(vectorial(aux1->p,aux2->p,aux3->p)!=0)
    {if(vectorial(aux1->p,aux2->p,aux3->p)>0)
     {triang->p=aux1->p;
      triang->sig=(lista*) malloc(sizeof(lista));
      triang->sig->p=aux2->p;
      triang->sig->sig=(lista*) malloc(sizeof(lista));
      triang->sig->sig->p=aux3->p;
      triang->sig->sig->sig=NULL;
     }
     else
     {triang->p=aux1->p;
      triang->sig=(lista*) malloc(sizeof(lista));
      triang->sig->p=aux3->p;
      triang->sig->sig=(lista*) malloc(sizeof(lista));
```

```
      triang->sig->sig->p=aux2->p;
      triang->sig->sig->sig=NULL;
     }
     punto=conjunto;
     cor=0;
     while((punto!=NULL))
     {if(((punto->p.x!=aux1->p.x)||(punto->p.y!=aux1->p.y))&&((punto->p.x!=aux2->p.x)||
        (punto->p.y!=aux2->p.y))&&((punto->p.x!=aux3->p.x)||(punto->p.y!=aux3->p.y)))
       cor=cor+cortes(punto->p,triang);
      punto=punto->sig;
     }
     if(cor==0)
      fin=true;
    }//del if
    aux3=aux3->sig;
   }
   aux2=aux2->sig;
  }
  aux1=aux1->sig;
 }
}/*fin Conjunto_Inicial*/

/*-------------------------------------------------------------------------------------
La siguiente función calculará el ángulo que forman dos vectores. Los  vectores  vendrán
dados por los puntos que determinan sus extremos.
-------------------------------------------------------------------------------------*/

double angulo (punto p1,punto p2, punto p3, punto p4)
{double u[2];
 double v[2];
 double m1,m2;
 u[0]=p2.x-p1.x;
 u[1]=p2.y-p1.y;
 v[0]=p4.x-p3.x;
 v[1]=p4.y-p3.y;
                                    //Módulos

 m1=sqrt((u[0]*u[0])+(u[1]*u[1]));
 m2=sqrt((v[0]*v[0])+(v[1]*v[1]));
 return(acos( ((u[0]*v[0])+(u[1]*v[1]))/(m1*m2)));
}

/*-------------------------------------------------------------------------------------
Ahora iremos añadiendo incrementalemente puntos siguiendo el  mecanismo RPG. Utilizaremos
el algoritmo de cierres convexo, (CH), anterior. La siguiente  función nos da como salida
la cantidad de puntos pertenecientes a una lista quedan hay dentro de un polígono.
-------------------------------------------------------------------------------------*/

int Dentro (lista *poligono, lista *quedan)
{int i=0;
 lista *aux;
 aux=quedan;
 while(aux!=NULL)
 {if(fmod(cortes(aux->p,poligono),2.0)!=0.0) ++i;
   aux=aux->sig;
 }
return(i);
}

/*-------------------------------------------------------------------------------------
Necesitamos conocer el número de cortes que tiene un segmento con un polígono. Esto  será
realizado por la función cortes_bis.
-------------------------------------------------------------------------------------*/

int cortes_bis(punto p,punto q,lista *poligono)
{int i=0; lista *aux;
 aux=poligono;
 if(aux!=NULL)
 {while(aux->sig!=NULL)
  {if(cortan(p,q,aux->p,aux->sig->p)==true)++i;
    aux=aux->sig;
  }
  if(cortan(p,q,aux->p,poligono->p)==true)++i;
 }
 return(i);
}
```

```
/*-------------------------------------------------------------------------------------
La siguiente función auxiliar muestra si dos puntos son iguales.
-------------------------------------------------------------------------------------*/

bool Iguales (punto p1, punto p2)
{if((long double(fabs(p1.x-p2.x)<=0.00001))&&long double((fabs(p1.y-p2.y)<=0.00001)))
  return(true);
 else
  return(false);
}

bool listasIguales(lista *l1,lista *l2)
{lista *aux1,*aux2;
 int fin;
 fin=0;
 aux1=l1;
 aux2=l2;
 if(longitud(l1)!=longitud(l2))
  return(false);
 while((aux1!=NULL)&&(fin==0))
  if(Iguales(aux1->p,aux2->p)==true)
  {aux1=aux1->sig;
   aux2=aux2->sig;
  }
  else
   fin=1;
  if(fin==0)
   return(true);
  else
   return(false);
}

/*-------------------------------------------------------------------------------------
Vamos a diseñar una función que nos  diga si un punto ve los dos extremos de un  segmento
sin cortar al resto de lados del polígono. Esta función será fundamental en el  mecanismo
incremental que implementamos.
-------------------------------------------------------------------------------------*/

bool ve(punto p1, punto p2, punto p3, lista *poligono)
{lista *triangulo,*aux;
 int d;
                                  //Construimos el triangulo inicial

triangulo=(lista*) malloc(sizeof(lista));
triangulo->p=p2;
triangulo->sig=(lista*) malloc(sizeof(lista));
triangulo->sig->p=p1;
triangulo->sig->sig=(lista*) malloc(sizeof(lista));
triangulo->sig->sig->p=p3;
triangulo->sig->sig->sig=NULL;
d=0;
                     //Vamos a contar los puntos del polígono que no sean ni p2 ni p3 y que estén
                     //dentro del triángulo. La función Limpiar libera la memoria dinámica.

aux=poligono;
while(aux!=NULL)
{if((Iguales(aux->p,p2)==false)&&(Iguales(aux->p,p3)==false))
 if(fmod(cortes(aux->p,triangulo),2.0)!=0.0)++d;
 aux=aux->sig;
}
if((d==0)&&(cortes_bis(p1,p2,poligono)==2)&&(cortes_bis(p1,p3,poligono)==2))
 return(true);
else
 return(false);
triangulo=Limpiar(triangulo);
}

/*-------------------------------------------------------------------------------------
La siguiente función responde a la pregunta de si un punto está en una lista o no.
-------------------------------------------------------------------------------------*/

bool Pertenece(punto p,lista *polig)
{lista *aux;
 int i=0;
 aux=polig;
 while (aux!=NULL)
 {if(Iguales(aux->p,p)==true) ++i;
   aux=aux->sig;
 }
```

```
 if(i!=0)
  return(true);
 else
  return(false);
}

/*-------------------------------------------------------------------------------------
Ya estamos en condiciones de diseñar nuestro generador aleatorio de polígonos  basado en
la teoría expuesta en el Apéndice B.
-------------------------------------------------------------------------------------*/

void RPG(lista *poligono, lista *conjunto)
{lista *aux,*cierreaux,*aux1,*aux2;
 bool seguir;
 FILE *polígon;
                                    //Generamos el triángulo inicial
 Calcular_triangulo(conjunto,poligono);
                                    //Si la longitud del conjunto es 3 finalizamos.
 if (longitud(conjunto)==3)
  goto fin; //si no seguimos
                                    //Quitamos los tres elementos del conjunto
aux=conjunto;
while(Pertenece(aux->p,poligono)==true)
{aux1=aux; //Miramos al principio
 aux=aux->sig;
 conjunto=aux;
 aux1->sig=NULL;
 aux1=Limpiar(aux1);
}
aux1=aux;
aux=aux->sig;
while(aux!=NULL)
{if(Pertenece(aux->p,poligono)==true)
 {aux2=aux; //Seguimos hasta el final
  aux1->sig=aux->sig;
  aux=aux1;
  aux2->sig=NULL;
  aux2=Limpiar(aux2);
  aux=aux->sig;
 }
 else
 {aux1=aux;
  aux=aux->sig;
 }
}
inicio=time(NULL);
while(conjunto!=NULL)
{aux=conjunto;
                                    //Tomo un punto
 conjunto=conjunto->sig;
                                    //Lo quito del conjunto
 aux->sig=NULL;
                                    //Lo añado al poligono
 aux2=poligono;
 while(aux2->sig!=NULL) aux2=aux2->sig;
  aux2->sig=aux;
                                    //Calculo el cierre del nuevo poligono
 cierreaux=(lista*) malloc(sizeof(lista));
 cierreaux->sig=NULL;
cierreaux=Cierre_Convexo(poligono,cierreaux);

/*-------------------------------------------------
Quito el punto del polígono mirando los elementos del
conjunto dentro del cierre convexo. Por otra parte si
dentro del conjunto no hay ningún punto perteneciente
al conjunto restante se añade el punto al polígono.
/*-------------------------------------------------

aux2=poligono;
while(aux2->sig!=aux) aux2=aux2->sig;
 aux2->sig=NULL;
if(Dentro(cierreaux,conjunto)==0)
{aux2=poligono;
 seguir=true;
 while((aux2->sig!=NULL)&&(seguir))
 {if(ve(aux->p,aux2->p,aux2->sig->p,poligono)==true)
  seguir=false;
```

```
  else
   aux2=aux2->sig;
 }
 if(aux2->sig!=NULL)
 {aux->sig=aux2->sig;
  aux2->sig=aux;
 }
 else
  aux2->sig=aux;
}
else
{aux2=conjunto;
 while(aux2->sig!=NULL)
 aux2=aux2->sig;
 aux2->sig=aux;
}                                //Limpiamos memoria
cierreaux=Limpiar(cierreaux);
} /*Fin while*/

fin:                             //Guardamos datos para si visualización con MatLab
aux=poligono;
poligon=fopen(''c:\\Pruebas_Heur_P1\\Matlab\\poligono.txt'',''w+'');
while(aux!=NULL)
{fprintf(poligon,''%lf %lf\n'',aux->p.x,aux->p.y);
 aux=aux->sig;
}
fclose(poligon);

} /*Fin RPG*/
```

Generador aleatorio de una clase de polígonos monótonos

Como se puede observar en la Sección 7.6, algunas de nuestras experimentaciones se basan en una clase particular de polígonos monótonos. Presentamos a continuación la implementación de un generador para este tipo de polígonos.

```
/*-------------------------------------------------------------------------------------
El mecanismo de constucción de  nuestros  polígonos  mónotonos es el siguiente: ordenamos
los puntos por su abcisa, tomando después  el primero y el último de estos  puntos   para
construir el segmento que los  une; ahora   generamos las cadenas monótonas de los puntos
situados por encima de este segmento  y por debajo;para finalizar se unen los extremos de
dichas cadenas monótonas.
-------------------------------------------------------------------------------------*/

void RPG_MONOTONO(lista *poligono, lista *conjunto)
{lista *aux;
 int i,n,m,n1,m1;
 double pend;
 punto tot[Tamcierre],inf[Tamcierre],sup[Tamcierre],temp;
 FILE *poligon;
                                 //guardamos todo el conjunto en listatotal
 i=-1;
 aux=conjunto;
 while(aux!=NULL)
 {i=i+1;
  tot[i]=aux->p;
  aux=aux->sig;
 }
                                 //Ordenamos esta lista por la abcisa
 n=0;
 while(n<i)
 {m=n+1;
  while(m<=i)
  {if(tot[m].x<tot[n].x)
   {temp=tot[n];
    tot[n]=tot[m];
    tot[m]=temp;
   }++m;
  }++n;
 }
                                 //Construimos la lista inferior y superior
 n=-1;
 m=-1;
```

```
pend=(tot[i].y-tot[0].y)/(tot[i].x-tot[0].x);
aux=conjunto;
while(aux!=NULL)
{if (((aux->p.y-tot[0].y)-pend*(aux->p.x-tot[0].x))<=0)
 {++n;
  inf[n]=aux->p;
 }
 else
 {++m;
  sup[m]=aux->p;
 }
 aux=aux->sig;
}
                                    //Ordenamos ahora la lista inferior y la superior
n1=0;
while(n1<n)
{m1=n1+1;
 while(m1<=n)
 {if(inf[m1].x<inf[n1].x)
  {temp=inf[n1];
   inf[n1]=inf[m1];
   inf[m1]=temp;
  }++m1;
 }++n1;
}
n1=0;
while(n1<m)
{m1=n1+1;
 while(m1<=m)
 {if(sup[m1].x<sup[n1].x)
  {temp=sup[n1];
   sup[n1]=sup[m1];
   sup[m1]=temp;
  }++m1;
 }++n1;
}
                                    //Se guarda en el polígono
i=0;
polígono->p=inf[i];
i=i+1;
aux=poligono;
while(i<=n)
{aux->sig=(lista*)malloc(sizeof(lista));
 aux=aux->sig;
 aux->p=inf[i];
 i=i+1;
}
                                    //la superior al revés
i=m;
while(i>=0)
{aux->sig=(lista*)malloc(sizeof(lista));
 aux=aux->sig;
 aux->p=sup[i];
 --i;
}
aux->sig=NULL;
                                    //Lo guardamos ahora en el fichero del polígono
aux=poligono;
poligon=fopen(''c:\\PVGC\\Pruebas_Heur_P1\\Matlab\\poligono.txt'',''w+'');
while(aux!=NULL)
{fprintf(poligon,''%lf %lf\n'',aux->p.x,aux->p.y);
 aux=aux->sig;
}
fclose(poligon);
}/*fin RPG_MONOTONO*/
```

En ocasiones se ha necesitado generar polígonos con características específicas para validar la corrección de ciertos algoritmos. Esto se ha conseguido implementado los correspondientes `ficheros.m` gráficos de MatLab,(que nos permiten dibujar en la pantalla del ordenador un polígono determinado) y cuyas implementaciones se encuentran en la Sección B.9. Una vez dibujados el polígono con estas funciones, es necesario incorporarlo a nuestro código C++ para aplicarle las técnicas heurísticas. Esto se consigue con las siguientes funciones.

```
/*-------------------------------------------------------------------------------------
La siguiente función leerá el polígono pintado mediante  el fichero.m  generapolígono  de
Matlab.
-------------------------------------------------------------------------------------*/

void Lectura_RPG(lista *poligono)
{FILE *poligon;
 lista *aux;
 punto punt;
 poligon=fopen(''c:\\Pruebas_Heur_P1\\Matlab\\poligono.txt'',''r'');
 fscanf(poligon,''%lf %lf\n'',&punt.x,&punt.y);
 poligono->p.x=punt.x; //se guarda en la lista
 poligono->p.y=punt.y;
 aux=poligono;
 while(feof(poligon)==0)
 {fscanf(poligon,''%lf %lf\n'',&punt.x,&punt.y);
  aux->sig=(lista*)malloc(sizeof(lista));
  aux=aux->sig; //Vamos generando la lista
  aux->p.x=punt.x;
  aux->p.y=punt.y;
 }
                                  //Cerramos la lista
 fclose(poligon);
 aux->sig=NULL;
}

/*-------------------------------------------------------------------------------------
La siguiente función leerá la nube pintada mediante el fichero.m generanube
de MatLab.
-------------------------------------------------------------------------------------*/

void Lectura_NUBE(lista *nube)
{FILE *poligon;
 lista *aux;
 punto punt;
 poligon=fopen(''c:\\Pruebas_Heur_P1\\Matlab\\nube.txt'',''r'');
 fscanf(poligon,''%lf %lf\n'',&punt.x,&punt.y);
 nube->p.x=punt.x; //se guarda en la lista
 nube->p.y=punt.y;
 aux=nube;
 while(feof(poligon)==0)
 {fscanf(poligon,''%lf %lf\n'',&punt.x,&punt.y);
  aux->sig=(lista*)malloc(sizeof(lista));
  aux=aux->sig;
                                  //Vamos generando la lista
  aux->p.x=punt.x;
  aux->p.y=punt.y;
 }
                                  //Cerramos la lista
 fclose(poligon);
 aux->sig=NULL;

}
```

B.4 Métodos aproximados para el problema MaxA-p-Pv1(P)

Presentamos los códigos implementados para solucionar el problema MaxA-p-Pv1(P). Los programas aparecen con la misma secuencialidad con la que se expusieron en el Capítulo 7. Teniendo en cuenta que los códigos para la heurística GA en este caso $k = 1$, se encuentran contenidos en los códigos para el problema MaxA-p-Pvk(P, k), esta secuencialidad es la siguiente:

1. Implementaciones basadas en la heurística *simulated annealing-SA*.
2. Implementaciones para la heurística *random search-RS*.
3. Implementaciones para la heurística *gradiente-GRAD*.
4. Implementaciones para el estudio de la descomposición $\mathcal{S}$ de un polígono P.

5. Códigos para obtener datos sobre el problema PorA-p-Pv1(P).

Incluimos a continuación la implementación de funciones axiliares a nuestras heurísticas. Algunas de estas funciones son importantes para el funcionamiento de los métodos, como pueden ser la construcción del *polígonos de visibilidad* de un punto interior a un polígono, (`calpolvisi`), y el cálculo del área de éste, (`área_polígono`), mediante la triangulación por "orejas"[68]. Estas funciones implementan la *función objetivo* o *función fitness* de nuestras heurísticas y que se debe maximizar.

```
/*-------------------------------------------------------------------------------------
Debemos  realizar en esta parte algunas  funciones que permitan calcular el área de cual-
quier polígono. Para ello es preciso triangular un polígono y después calcular el área de
cada uno de  los triángulos. En el caso  de un polígono de visibilidad,(función que habrá
que implementar también), será más  fácil  triangular  este polígono  pues es un polígono
estrellado, (con tomar un punto interior y unirlo con los vértices  tendremos triangulado
el polígono. Veamos en primer lugar un  polígono  que  triangule un polígono dado por sus
vértices y  que están  guardados en una matriz. Implementamos una función  que calcula el
número de nodos que quedan en una lista.
-------------------------------------------------------------------------------------*/

long int longitud(lista *poligono)
{lista *aux;
 long int i;
 i=0;
 aux=poligono;
 while(aux!=NULL)
 {aux=aux->sig;
  ++i;
 }
return(i);
}

/*-------------------------------------------------------------------------------------
Implementamos ahora una función que mide la longitud de una lista circular
-------------------------------------------------------------------------------------*/

long int longitudc(lista *poligono)
{lista *aux;
 long int i;
 i=0;
 aux=poligono;
 while(aux->sig!=poligono)
 {aux=aux->sig;
  ++i;
 }
 return(i);
}

/*-------------------------------------------------------------------------------------
Un triángulo es oreja si el poligono es un triangulo o el segmento que une p1 con p3 está
dentro del poligono y no hay cortes con los lados. Esto  lo  hacemos  mirando que el seg-
mento p1p3 solo corta al polígono  en  los  vértices, (el nºde cortes  será 4) y el punto
medio de pip3 está dento del polígono.
-------------------------------------------------------------------------------------*/

bool es_oreja(punto p1, punto p2, punto p3,lista *poligono)
{punto medio;
 medio.x=(p1.x+p3.x)/2;
 medio.y=(p1.y+p3.y)/2;
                                  //Miramos si están alineados o si es un triángulo
if((fabs(vectorial(p1,p2,p3))<=0.0000001)||(longitud(poligono)==3)||
  ((cortes(medio,poligono)%2!=0)&&(cortes_bis(p1,p3,poligono)==4) ))
 return(true);
else
 return(false);
}

/*-------------------------------------------------------------------------------------
Damos  a continuación  un algoritmo que calcula el area de un triangulo dado por sus tres
vértices.
-------------------------------------------------------------------------------------*/

double area(punto p1,punto p2, punto p3)
```

```
{double sp,l1,l2,l3;
 l1=sqrt(((p1.x-p2.x)*(p1.x-p2.x))+((p1.y-p2.y)*(p1.y-p2.y)));
 l2=sqrt(((p1.x-p3.x)*(p1.x-p3.x))+((p1.y-p3.y)*(p1.y-p3.y)));
 l3=sqrt(((p3.x-p2.x)*(p3.x-p2.x))+((p3.y-p2.y)*(p3.y-p2.y)));
 sp=((l1+l2+l3)/2.0);
 if((sp*(sp-l1)*(sp-l2)*(sp-l3))<=0)
  return(0.0);
 else
  return(sqrt(sp*(sp-l1)*(sp-l2)*(sp-l3)));
}

/*-------------------------------------------------------------------------------------
Creamos ahora  una  función que guarda tanto en una lista como en un fichero los vértices
del polígono  tal  que cada tres consecutivos forman un triángulo de la triangulación del
polígono. Calculamos así el área del polígono.
-------------------------------------------------------------------------------------*/

double area_poligono(lista *poligono)
{double ar=0;
 lista *aux,*aux1,*aux2,*aux3,*copia;
 time_t ini,actu;
 FILE *triangulacion;
                                  //Hacemos una copia del poligono para no perderlo
 aux=poligono;
 if(aux==NULL) return(0);
 if(aux!=NULL)
 {copia=(lista*) malloc(sizeof(lista));
  copia->p=aux->p;
  aux=aux->sig;
  aux1=copia;
 }
 while(aux!=NULL)
 {aux1->sig=(lista*) malloc(sizeof(lista));
  aux1=aux1->sig;
  aux1->p.x=aux->p.x;
  aux1->p.y=aux->p.y;
  aux=aux->sig;
 }
 aux1->sig=NULL;
                                 //Vamos buscando orejas y guardándolas en el fichero

 triangulacion=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\tri.txt'',''w+'');
 while(longitud(copia)>3)
 {aux1=copia;
  aux2=copia->sig;
  aux3=copia->sig->sig;
  ini=time(NULL);
  while(es_oreja(aux1->p,aux2->p,aux3->p,copia)==false)
  {actu=time(NULL);
   if(difftime(actu,ini)>3){ar=-1.0;goto fin;}
   if(aux3->sig==NULL)
   {aux1=aux1->sig;              //La oreja está en el último triángulo
    aux2=aux2->sig;
    aux3=copia;
   }
   else
    if(aux2->sig==NULL)
    {aux1=aux1->sig;
     aux2=copia;
     aux3=aux3->sig;
    }
    else
     if(aux1->sig==NULL)
     {aux1=copia;
      aux2=aux2->sig;
      aux3=aux3->sig;
     }
     else
     {aux1=aux1->sig;
      aux2=aux2->sig;
      aux3=aux3->sig;
     }
  }
                                  //Ya encontré una oreja
                                  //La guardo en el fichero y la quito de la copia

 fprintf(triangulacion,''%f %f\n'',aux1->p.x,aux1->p.y);
 fprintf(triangulacion,''%f %f\n'',aux2->p.x,aux2->p.y);
```

```
 fprintf(triangulacion,''%f %f\n'',aux3->p.x,aux3->p.y);
 ar=ar+ area(aux1->p,aux2->p,aux3->p);

                                  //Sumamos las área de triángulos
                                  //Liberamos aux2 con cuidado de que aux2 haya caido justo en copia
 if(aux2==copia)
 {copia=copia->sig;
  aux2->sig=NULL;
  aux2=Limpiar(aux2);
 }
 else
 {aux=copia;
  while(aux!=NULL)
  {if(aux->sig==aux2)
   {aux->sig=aux->sig->sig;
    aux2->sig=NULL; //Liberamos aux2
    aux2=Limpiar(aux2);
   }
 aux=aux->sig;
  }
 }
 }

                                  //Sumamos el área del último triángulo

fprintf(triangulacion,''%f %f\n'',copia->p.x,copia->p.y);
fprintf(triangulacion,''%f %f\n'',copia->sig->p.x,copia->sig->p.y);
fprintf(triangulacion,''%f %f\n'',copia->sig->sig->p.x,copia->sig->sig->p.y);
ar=ar+area(copia->p,copia->sig->p,copia->sig->sig->p);

                                  //Lo guardamos en la última fila del fichero
                                  //El último triángulo no lo guardo en el fichero

fprintf(triangulacion,''%lf %f\n'',ar,0);
fclose(triangulacion);
fin:
copia=Limpiar(copia);
return(ar);
}

double modulo(punto p, punto q)
{double v1,v2;
 v1=q.x-p.x;
 v2=q.y-p.y;
 return(sqrt((v1*v1)+(v2*v2)));
}

/*-------------------------------------------------------------------------------------
Implementamos ahora un algoritmo que calculará el polígono de  visibilidad  de  un  punto
interior de un polígono. Para  calcular  el área de este polígono podemos usar la función
anterior o tener en cuenta que  al ser  un polígono estrellado el punto de la luz se debe
poder  unir con todos los vértices triangulando el polígono y podemos calcular el area de
esos polígonos. Implementamos  una función que nos informa si dos rectas dadas por un par
de puntos se cortan en el plano. En el caso que sea así  nos devuelve las coordenadas del
punto.
-------------------------------------------------------------------------------------*/

bool cortanrectas (punto r1,punto r2,punto q1, punto q2, double *x,double *y)
{double v1,v2,w1,w2; punto p;
 v1=r2.x-r1.x;
 v2=r2.y-r1.y;
 w1=q2.x-q1.x;
 w2=q2.y-q1.y;
 if((v1*w2)==(v2*w1)) //Son paralelas
  return(false);
 else
 {*x=(q1.x*(q2.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x)-q2.x*(q1.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x))/
  (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));
  *y=(q1.x*q2.y*(r1.y-r2.y)-q1.y*(q2.x*(r1.y-r2.y)+r1.x*r2.y-r1.y*r2.x)+q2.y*(r1.x*r2.y-r1.y*r2.x))/
  (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));

                                  //Miramos ahora si (x,y) está en el segmento q1q2 y en
                                  //sentido positivo
  p.x=*x;
  p.y=*y;
  if((Iguales(p,q1)||Iguales(p,q2))&&((((p.x-r1.x)!=0)&&(((r2.x-r1.x)/(p.x-r1.x))>0))||
    (((p.y-r1.y)!=0)&&(((r2.y-r1.y)/(p.y-r1.y))>0))))
   return(true);
```

```
  else
   if((vectorial(r1,p,q1)/vectorial(r1,p,q2)<0)&&((((p.x-r1.x)!=0)&&(((r2.x-r1.x)/(p.x-r1.x))>0))||
     (((p.y-r1.y)!=0)&&(((r2.y-r1.y)/(p.y-r1.y))>0))))
    return(true);
   else
    return(false);
 }
}

/*-------------------------------------------------------------------------------------
La función cortan rectas anterior solamente calcula los puntos de intersección en un sen-
tido. Vamos a simplificarla un poco para que calcule en ambos sentidos
-------------------------------------------------------------------------------------*/

bool cortanrectasbis (punto r1,punto r2,punto q1, punto q2, double *x,double *y)
{double v1,v2,w1,w2; punto p;
 v1=r2.x-r1.x;
 v2=r2.y-r1.y;
 w1=q2.x-q1.x;
 w2=q2.y-q1.y;
 if((v1*w2)==(v2*w1))           //Son paralelas
  return(false);
 else
 {*x=(q1.x*(q2.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x)-q2.x*(q1.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x))/
     (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));
  *y=(q1.x*q2.y*(r1.y-r2.y)-q1.y*(q2.x*(r1.y-r2.y)+r1.x*r2.y-r1.y*r2.x)+q2.y*(r1.x*r2.y-r1.y*r2.x))/
     (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));
  p.x=*x;
  p.y=*y;
  if(Igualesbis(p,q1)||Igualesbis(p,q2))
   return(true);
  else
   if ((vectorial(r1,p,q1)/vectorial(r1,p,q2)<0.0))
    return(true);
   else
    return(false);
 }
}

/*-------------------------------------------------------------------------------------
Implementamos ahora un algoritmo que calculará el polígono de  visibilidad  de  un  punto
El algoritmo implementado es un algoritmo lineal, (ver [72]).
-------------------------------------------------------------------------------------*/

lista *calpolvisi(punto p, lista *poligono, lista *polvisi)
{lista *copia,*aux,*aux1,*aux2,*aux3,*aux4,*aux11,*aux22;
 punto paux,paux1;
 double cx,cy,d;
 time_t inicio,actual,ini2,para2,ini3,para3;

                                //Lo guardamos todo en una lista circular

 aux=poligono;
 if(aux!=NULL)
  {copia=(lista*) malloc(sizeof(lista));
   copia->p=aux->p;
   aux=aux->sig;
   aux1=copia;
  }
  while(aux!=NULL)
  {aux1->sig=(lista*) malloc(sizeof(lista));
   aux1=aux1->sig;
   aux1->p.x=aux->p.x;
   aux1->p.y=aux->p.y;
   aux=aux->sig;
  }
 aux1->sig=NULL;
                                //De momento no es circular
                                //Buscamos el primer punto de corte más cercano a p
                                //Buscamos el primer elemento
 aux=copia;
 paux.x=1.0;
 paux.y=p.y;
 d=2;
 while(aux->sig!=NULL)
 {if(cortanrectas(p,paux,aux->p,aux->sig->p,&cx,&cy)==true)
  {paux1.x=cx;
   paux1.y=cy;
   if(modulo(p,paux1)<d)
```

```
  {aux2=aux;
   aux1=aux->sig;
   d=modulo(p,paux1);
   polvisi->p=paux1;
  }
 }
 aux=aux->sig;
}
                                //Analizamos el último lado para cerrar el polígono

if(cortanrectas(p,paux,aux->p,copia->p,&cx,&cy)==true)
{paux1.x=cx;
 paux1.y=cy;
 if(modulo(p,paux1)<d)
{aux2=aux;
  aux1=copia;
 d=modulo(p,paux1);
   polvisi->p=paux1;
 }
}
                                //Fin primer elemento. Ahora los demás.
                                //Ahora hacemos la lista circular
aux->sig=copia;
aux3=polvisi; //aux3 apunta al ultimo punto introducido en polvisi
aux4=aux3;
                                //Añadimos el vértice al que apunta aux4
                                //que es el primero
while(aux4!=aux2)               //Hasta que demos toda la vuelta
                                //Tomamos el más cercano para ver que el nºde cortes es
                                //2 con el punto más cercano a P entre aux4->p y aux1->p

{if(modulo(p,aux4->p)<modulo(p,aux1->p))
  aux=aux4;
 else
  aux=aux1;
 if(fabs(vectorial(p,aux4->p,aux1->p))<=0.00000000001)
  if(cortes_bis(p,aux->p,poligono)==2)
  {aux3->sig=(lista*) malloc(sizeof(lista));
   aux3=aux3->sig;
   aux3->p=aux1->p;             //En este if estudiamos todos los casos de alineación
   aux4=aux1;
   aux1=aux1->sig;
  }
  else
  {aux4=aux1;
   aux1=aux1->sig;
  }
 else
  if((vectorial(p,aux4->p,aux1->p))>0.0)
                                //Positivo. Se añade si los cortes son 2

   if(cortes_bis(p,aux1->p,poligono)==2)
   {aux3->sig=(lista*) malloc(sizeof(lista));
    aux3=aux3->sig;
    aux3->p=aux1->p;
    aux4=aux1;
    aux1=aux1->sig;
   }
   else                         //Sino simplemente se avanza
   {aux4=aux1;
    aux1=aux1->sig;
   }
  else                          //Producto vectorial negativo
                                //El vértice se esconde

   if ((cortes_bis(p,aux1->p,poligono)!=2)&&(cortes_bis(p,aux4->p,poligono)==2))
   {aux=aux1;
    d=2.0;
    aux3->sig=(lista*) malloc(sizeof(lista));
    aux3=aux3->sig;
    while(aux->sig!=aux4)
    {if(cortanrectas(p,aux4->p,aux->p,aux->sig->p,&cx,&cy)==true)
     {paux1.x=cx;
      paux1.y=cy;
      if(modulo(p,paux1)<d)
      {aux22=aux;
```

```
        aux11=aux->sig;
        d=modulo(p,paux1);
        aux3->p=paux1;
       }
      }
      aux=aux->sig;
     }
     aux4=aux22;
     aux1=aux11;
     }
                                //El vértice que vemos es el nuevo
    else
     if ((cortes_bis(p,aux1->p,poligono)==2)&&(cortes_bis(p,aux4->p,poligono)!=2))
     {aux=aux1->sig;
      d=2.0;
      aux3->sig=(lista*) malloc(sizeof(lista));
      aux3=aux3->sig;
      while(aux!=aux4)
      {if(cortanrectas(p,aux1->p,aux->p,aux->sig->p,&cx,&cy)==true)
       {paux1.x=cx;
        paux1.y=cy;
        if(modulo(p,paux1)<d)
        {d=modulo(p,paux1);
         aux3->p=paux1;
        }
       }
       aux=aux->sig;
      }
    aux3->sig=(lista*) malloc(sizeof(lista));
    aux3=aux3->sig;
    aux3->p=aux1->p;
    aux4=aux1;
    aux1=aux1->sig;
    }
    else
                                //No se ve ninguno. Seguimos
    {aux4=aux1;
     aux1=aux1->sig;
    }
 }/*Del while*/
 aux3->sig=NULL;
 aux=polvisi;
                                //Marcamos los vértices con tipo 'v'
 while (aux!=NULL)              //Posteriormente se utilizará en la unión de polígonos
 {aux->p.tipo='v';
  aux=aux->sig;
 }                              //Limpiamos la copia
 aux=copia;
 while(aux->sig!=copia) aux=aux->sig;
  aux->sig=NULL;
 copia=Limpiar(copia);
 return(polvisi);
 aux=Limpiar(aux);
 aux1=Limpiar(aux1);aux2=Limpiar(aux2);
 aux3=Limpiar(aux3);aux4=Limpiar(aux4);aux11=Limpiar(aux11);aux22=Limpiar(aux22);
 }

/*-------------------------------------------------------------------------------------
Diseñamos una función que se utilizará para guardar todos los  polígonos de visibilidad y
poder visualizarlos con Matlab.
-------------------------------------------------------------------------------------*/

void guardar(punto p,lista *poligono,double ar)
{FILE *visi;
 lista *aux;
 double area;
 area=areapolvisi(p,poligono);
 visi=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisi.txt'',''w+'');
 aux=poligono;
 while(aux!=NULL)
 {fprintf(visi,''%f %f\n'',aux->p.x,aux->p.y);
  aux=aux->sig;
 }                              //Guardamos el punto y area en el fichero para pintarlo

 fprintf(visi,''%lf %lf\n'',p.x,p.y);
 fprintf(visi,''%lf %lf\n'',area,((area)*100)/ar);
 fclose(visi);
```

```
 }
/*-------------------------------------------------------------------------------------
Implementamos ahora un algoritmo que calculará el polígono de  visibilidad  de  un  punto
Ya  hemos implementado las funciones que calculan el poligono de visibilidad y que trian-
gulan.Podemos definir ahora una función que calcula el área de un polígono de visibilidad
dado con el punto. Como es un poligono estrellado es suficiente calcular el area de todos
los triángulos que se forman al unir el punto con todos los lados.
-------------------------------------------------------------------------------------*/

double areapolvisi(punto p,lista *poligono)
{lista *aux1,*aux2;
 double ar=0;
 aux1=poligono;
 aux2=aux1->sig;
 while(aux1->sig!=NULL)
 {ar=ar+area(p,aux1->p,aux2->p);
  aux1=aux1->sig;
  aux2=aux2->sig;
 }
                                    //Sumamos el triángulo que cierra
 ar=ar+area(p,aux1->p,poligono->p);
 return(ar);
}
```

Implementaciones basadas en la heurística simulated annealing-SA

Presentamos los códigos correspondientes a la heurística SA para solucionar el problema MaxA-p-Pv1(P). Incluimos comentarios sobre los tipos de decremento de la temperatura utilizados.

```
/*-------------------------------------------------------------------------------------
TIPO 1 : T(k)=(T0)/(ln(1+k))
TIPO 2 : T(k)=(T0)/(1+k))
TIPO 3 : T(k)=(T0)/(exp(k))
TIPO 4 : T(k)=(T0)/(exp(exp(k)))
TIPO 5 : T(k)=(0.9)(T(k))
-------------------------------------------------------------------------------------*/

lista *MaxSA_PV1(int tipor,double tpi,double dt,int *itera,lista *poligono,lista *polvisi,
                punto *p,double *a,double ar)
{unsigned long int i,k;
 long hora;
 int horasistema;
 punto p1,p2; lista *polix,*poliy,*aux;
 double apolix,apoliy,delta,tk,U,U1,U2,div;
 FILE *visits,*rastro;
 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''w+'');

                                     //Generamos una primera solución

 hora=time(NULL); //Generamos una semilla de sistema
 horasistema=(unsigned int) hora/2;
 srand(horasistema);
                                    //Creamos el punto/
 p1=generapunto(poligono);
 fprintf(rastro,''%f %f\n'',p1.x,p1.y);

                                    //Lo guardamos en rastro

 polix=(lista*) malloc(sizeof(lista));
 polix=calpolvisi(p1,poligono,polix);
 apolix=areapolvisi(p1,polix);
 k=1;
 tpi=longitud(poligono);
 tk=tpi;                            //tpi; TEMPERATURA INICIAL
 do
 {i=1;
  do
  {do
   {U1=(1.0*rand())/(RAND_MAX);
    U2=(1.0*rand())/(RAND_MAX);
    p2.x=p1.x+((sqrt(-(2.0*log(U1)))*sin(2.0*pi*U2))/10.0);
    p2.y=p1.y+((sqrt(-(2.0*log(U1)))*cos(2.0*pi*U2))/10.0);
```

```
   }while(punto_int(p2,poligono)==false);

    poliy=(lista*) malloc(sizeof(lista));
    poliy=calpolvisi(p2,poligono,poliy);
    apoliy=areapolvisi(p2,poliy);
    fprintf(rastro,''%f %f\n'',p2.x,p2.y);
    *itera=*itera+1;
                                  //Generamos una nueva solución
                                  //Calculamos delta
    delta=apoliy-apolix;
                                  //Estudiamos el cambio
    if(delta>=0)
    {polix=Limpiar(polix);
     polix=poliy;
     apolix=apoliy;
     p1=p2;
     fprintf(rastro,''%f %f\n'',p1.x,p1.y);
    }
    else
    {U=(1.0*rand())/(RAND_MAX);
     if((U<exp((delta)/(tk)))&&(poliy->p.x!=-10.0))
     {printf(''*'');
      polix=Limpiar(polix);
      polix=poliy;
      apolix=apoliy;
      p1=p2;
      fprintf(rastro,''%f %f\n'',p1.x,p1.y);
     }
    }
   ++i;
  }while ((i)<=(tk));
  switch(tipor)
  {case 0:tk=dt*tk;break;
   case 1:tk=(tpi)/log(1+k);break;
   case 2:tk=(tpi)/(1+k); break;
   case 3:tk=(tpi)/exp(k); break;
   case 4:tk=(tpi)/(exp(exp(k)));break;
   case 5:tk=(0.9)*tk;break;
  }
  k=k+1;
  div=div/0.99;
  printf(''\n\t%lf..\t'',tk);
 }while(tk>0.0005);
*p=p1;                            //Actualizamos la salida
*a=apolix;                        //Guardamos en un fichero par ver en Matlab

visits=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisirs.txt'',''w+'');
aux=polix;
while(aux!=NULL)
{fprintf(visits,''%f %f\n'',aux->p.x,aux->p.y);
 aux=aux->sig;
}                                 //Guardamos el punto y área en el fichero para pintarlo

fprintf(visits,''%lf %lf\n'',(*p).x,(*p).y);
fprintf(visits,''%lf %lf\n'',apolix,((*a)*100)/ar);
fclose(visits);
fclose(rastro);
return(polix);

}
```

Implementaciones para la heurística random search-RS.

La heurística RS está basada en la búsqueda sobre una nube de puntos interiores al polígono P, generada aleatoriamente mediante la función `creanube`, cuyo código se puede encontrar en el conjunto de funciones generales y programa principal.

```
/*-----------------------------------------------------------------------------
Implementaremos  primero un algoritmo de búsqueda aleatoria, que consiste simplemente  en
recorrer la nube interna de polígono calculada y  tomar el punto cuyo polígono de visibi-
lidad tenga mayor área. Evidentemente este sistema no será el más óptimo.
-----------------------------------------------------------------------------*/
```

```
lista *MaxRS_PV1(lista *nube,lista *poligono,lista *polvisi,punto *p, double *a,double ar)
{lista *aux,*polvisiaux1;
 double areaux=0,area=0;
 FILE *visidet;
 int i=1;
 aux=nube;
 while(aux!=NULL)
 {polvisiaux1=(lista*) malloc(sizeof(lista));
  polvisiaux1=calpolvisi(aux->p,poligono,polvisiaux1);
  areaux=areapolvisi(aux->p,polvisiaux1);
  printf(''\n\t\t %d AREA=\t %lf'',i,areaux);
  ++i;
  if(areaux>area)
  {polvisi=Limpiar(polvisi);
   polvisi=polvisiaux1;
   area=areaux;
   *p=aux->p;
  }
  else
  if(areaux!=0) polvisiaux1=Limpiar(polvisiaux1);
   aux=aux->sig;
 }
                                    //Ahora lo guardamos en el fichero
 *a=area;
 visidet=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisidet.txt'',''w+'');
 aux=polvisi;
 while(aux!=NULL)
 {fprintf(visidet,''%f %f\n'',aux->p.x,aux->p.y);
  aux=aux->sig;
 }
                                    //Guardamos el punto y area en el fichero para pintarlo

 fprintf(visidet,''%lf %lf\n'',(*p).x,(*p).y);
 fprintf(visidet,''%lf %lf\n'',area,((*a)*100)/ar);
 fclose(visidet);
 aux=Limpiar(aux);
 return(polvisi);

}
```

Implementaciones para la heurística gradiente-GRAD.

Además del código correspondiente a la heurística incluimos funciones auxiliares necesarias en el método aproximado. Por ejemplo la función `candidatos`, obtiene el punto de máxima iluminación alrededor de un punto dado, siguiendo las ideas expuestas en la Sección 7.5.

```
/*-------------------------------------------------------------------------------------
Implementaremos  primero un algoritmo de búsqueda aleatoria, que consiste simplemente  en
Ahora ya podemos implementar el método del gradiente. Para ello guardaremos los  vértices
del poligono en una lista auxiliar,los vamos transformando según el gradiente  y al final
nos quedamos con el mejor.
-------------------------------------------------------------------------------------*/

lista *MaxGRAD_PV1(int *itera,lista *poligono,lista *polvisi,punto *p, double *a,double ar)
{lista *aux,*aux1,*copia,*polvisiaux1;
 double areaux=0,area=0;
 int cambios=1,j;
 punto paux;
 FILE *visigrad,*rastro;
 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''w+'');
 *itera=0;
 aux=poligono;
 if(aux!=NULL)
 {copia=(lista*) malloc(sizeof(lista));
  copia->p=aux->p;
  aux=aux->sig;
  aux1=copia;
 }
 while(aux!=NULL)
 {aux1->sig=(lista*) malloc(sizeof(lista));
  aux1=aux1->sig;
  aux1->p.x=aux->p.x;
  aux1->p.y=aux->p.y;
  aux=aux->sig;
```

```
 }
 aux1->sig=NULL;
                                  //Ahora le aplicamos a la lista el método del gradiente
                                  //hasta que en una de las vueltas no tengamos cambios
 while(cambios!=0)
 {cambios=0;
  aux=copia;
  j=1;
  while(aux!=NULL)
  {printf(''%d:'',j++);
   if(aux->p.tipo!='F') //Si con un punto hemos terminado no lo analizamos más
   {paux=aux->p;
    Candidato(poligono,&(aux->p),0.001);
   *itera=*itera+16;
    if(Iguales(paux,aux->p)==false)
     cambios=cambios+1;
    else
     aux->p.tipo='F';
   }
   aux=aux->sig;
  }
  printf(''\nC-%d \n '',cambios);     //Guardamos el rastro
  aux1=copia;
  while(aux1!=NULL)
  {fprintf(rastro,''%f %f\n'',aux1->p.x,aux1->p.y);
   aux1=aux1->sig;
  }
 }
                                  //Buscamos el mejor candidato
 aux=copia;
 while(aux!=NULL)
 {if((fmod(cortes(aux->p,poligono),2.0)!=0.0)&&(Es_Ordenada_vertice(aux->p,poligono)==false))
  {polvisiaux1=NULL;
   polvisiaux1=(lista*) malloc(sizeof(lista));
   polvisiaux1=calpolvisi(aux->p,poligono,polvisiaux1);
   areaux=areapolvisi(aux->p,polvisiaux1);
   if(areaux>area)
   {polvisi=Limpiar(polvisi);
    polvisi=polvisiaux1;
    area=areaux;
    *p=aux->p;
   }
   else
    if(polvisiaux1->p.x!=10.0)
     polvisiaux1=Limpiar(polvisiaux1);
  }
  else
   aux=aux->sig;
 }
                                 //Ahora lo guardamos en el fichero
 *a=area;
 visigrad=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisigrad.txt'',''w+'');
 aux=polvisi;
 while(aux!=NULL)
 {fprintf(visigrad,''%f %f\n'',aux->p.x,aux->p.y);
  aux=aux->sig;
 }
                                 //Guardamos el punto y area en el fichero para pintarlo

 fprintf(visigrad,''%lf %lf\n'',(*p).x,(*p).y);
 fprintf(visigrad,''%lf %lf\n'',area,((*a)*100)/ar);
 fclose(visigrad);fclose(rastro);
 aux=Limpiar(aux);
 return(polvisi);
}

/*-------------------------------------------------------------------------------------
Implementamos ahora una función que calcula el mejor  candidato  alrededor  de  un  nuevo
punto.
-------------------------------------------------------------------------------------*/

void Candidato (lista *poligono,punto *p,double inc)
{double area=0.0,areaux=0.0,areapol;
 lista *polvisiaux;
 punto cand,paux;
 int k;
                                  //Vamos mirando cada uno de los candidatos
```

```
 areapol=area_poligono(poligono);
 cand.x=(*p).x;cand.y=(*p).y; cand.tipo=(*p).tipo;
 paux=cand;
 if((fmod(cortes(cand,poligono),2.0)!=0.0)&&(Es_Ordenada_vertice(cand,poligono)==false))
 {polvisiaux=(lista*) malloc(sizeof(lista));
  polvisiaux=calpolvisi(cand,poligono,polvisiaux);
  areaux=areapolvisi(cand,polvisiaux);
  if((areaux>area)&&(areaux<areapol)) {area=areaux; paux=cand;}
 }
 for(k=0;k<64;++k)
 {cand.x=(*p).x+(inc*cos(k*(2*pi/64)));cand.y=(*p).y+(inc*sin(k*(2*pi/64)));
  if((fmod(cortes(cand,poligono),2.0)!=0.0)&&(Es_Ordenada_vertice(cand,poligono)==false))
  {polvisiaux=(lista*) malloc(sizeof(lista));
   polvisiaux=calpolvisi(cand,poligono,polvisiaux);
   areaux=areapolvisi(cand,polvisiaux);
   if((areaux>area)&&(areaux<areapol)) {area=areaux; paux=cand;}
  }
 }
 *p=paux;
}
```

Implementaciones para el estudio de la descomposición $\mathcal{S}$ de un polígono P.

Presentamos los códigos necesarios para construir la descomposición $\mathcal{S}$ de un polígono P, en *regiones de visibilidad*. Esta descomposición fue estudiada en la Sección 7.8 y permitía realizar un estudio para obtener un algoritmo exacto que soluciona el problema MaxA-p-Pv1(P).

```
/*-----------------------------------------------------------------------------------------
Implementamos finalmente  para  el problema  Max_p_PV1  un  pequeño  código  que  permita
justificar la hipótesis  que tenemos sobre  la  descomposición  S. Para ello calcularemos
todos los vértices  cóncavos y calcularemos  los puntos de  intersección de las  regiones
generadas por la descomposición al prolongar  los lados  adyacentes a vértices  cóncavos.
Cada vez  que encontremos un vértice  cóncavo lo  guardaremos en una lista y  colocaremos
después de él sus dos vértices adyacentes. Finalmente  calcularemos  la lista  de cortes.
Diseñamos previamente una función que  añade  elementos  a una  lista, aunque estén repe-
tidos.
-----------------------------------------------------------------------------------------*/

lista *AnadirR(lista *l,punto p)
{lista *aux;
 if(l==NULL)
 {l=(lista*)malloc(sizeof(lista));
  l->p=p;
  l->sig=NULL;
 }
 else
 {aux=l;
  while(aux->sig!=NULL)
   aux=aux->sig;
  aux->sig=(lista*)malloc(sizeof(lista));
  aux=aux->sig;
  aux->p=p;
  aux->sig=NULL;
 }
 return(l);
}

/*-----------------------------------------------------------------------------------------
Diseñamos una función especial de cortanrectas para que sea en sentido positivo en  ambos
segmentos.
-----------------------------------------------------------------------------------------*/

bool cortanrectasP (punto r1,punto r2,punto q1, punto q2, double *x,double *y)
{double v1,v2,w1,w2; punto p;
 v1=r2.x-r1.x;
 v2=r2.y-r1.y;
 w1=q2.x-q1.x;
 w2=q2.y-q1.y;
 if((v1*w2)==(v2*w1))
                                  //Son paralelas
 return(false);
 else
```

```
 {*x=(q1.x*(q2.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x)-q2.x*(q1.y*(r1.x-r2.x)-r1.x*r2.y+r1.y*r2.x))/
     (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));
  *y=(q1.x*q2.y*(r1.y-r2.y)-q1.y*(q2.x*(r1.y-r2.y)+r1.x*r2.y-r1.y*r2.x)+q2.y*(r1.x*r2.y-r1.y*r2.x))/
     (q1.x*(r1.y-r2.y)+q1.y*(r2.x-r1.x)+q2.x*(r2.y-r1.y)+q2.y*(r1.x-r2.x));
                                    //Miramos ahora si (x,y) está en el segmento q1q2 y en
                                    //sentido positivo
  p.x=*x;
  p.y=*y;
  return(true);
 }
}

/*-------------------------------------------------------------------------------------
Ya implementamos la búsqueda del punto de máxima iluminación utilizando la descomposición
S.
-------------------------------------------------------------------------------------*/

lista *MaxS_PV1(int *itera,lista *poligono,lista *polvisi,punto *p, double *a,double ar)
{lista *pcortes;
 lista *previa,*aux,*aux1,*aux2,*aux3,*polvisiaux1,*aux4;
 punto punt,puntaux,punt1,punt2,paux;
 double x,y,areaux=0,area=0,m1,m2;
 FILE *visidet,*fnube;
 int i,cambios;
 previa=NULL;
 pcortes=NULL;
 aux=poligono;
                                    //hacemos la lista circular
 while(aux->sig!=NULL)
  aux=aux->sig;
 aux->sig=poligono;
                                    //Analizamos los tres primeros vértices
 aux1=poligono;
 aux2=aux1->sig;
 aux3=aux2->sig;
 if(vectorial(aux3->p,aux1->p,aux2->p)<0)
 {previa=AnadirR(previa,aux2->p);
  previa=AnadirR(previa,aux1->p);
  previa=AnadirR(previa,aux3->p);
 }
 aux1=aux1->sig;
 aux2=aux2->sig;
 aux3=aux3->sig;
                                    //Le damos toda la vuelta alrededor

 while(aux1!=poligono)
 {if(vectorial(aux3->p,aux1->p,aux2->p)<0) //aux2->p es cóncavo
  {previa=AnadirR(previa,aux2->p);
   previa=AnadirR(previa,aux1->p);
   previa=AnadirR(previa,aux3->p);
  }
  aux1=aux1->sig;
  aux2=aux2->sig;
  aux3=aux3->sig;
 }
                                    //Quitamos la circularidad de poligono
 aux=poligono;
 while(aux->sig!=poligono)
  aux=aux->sig;
 aux->sig=NULL;
                                    //Vamos calculando los puntos de intersección
                                    //y vamos guardando en cortes
 aux=previa;
 while(aux!=NULL)
 {punt.x=aux->p.x; //Añadimos el vértice cóncavo
  punt.y=aux->p.y;
  punt.tipo='V';
  aux->p.tipo='V';
  pcortes=AnadirR(pcortes,aux->p);
  aux1=aux->sig;
  aux2=aux1->sig;
  aux3=aux2->sig;
  fflush(stdin);
  while(aux3!=NULL)
  {if(cortanrectasP(aux1->p,aux->p,aux3->sig->p,aux3->p,&x,&y)==true)
   {punt.x=x;
    punt.y=y;
```

```
    punt.tipo='I';
    if(Iguales(punt,aux1->p)==false)
    {if((x>0.0)&&(x<1.0)&&(y>0.0)&&(y<1.0))
      if(((fmod(cortes(punt,poligono),2.0)==0.0)||(Es_Ordenada_vertice(punt,poligono)==true))==false)
       if((cortes_bis(aux->p,punt,poligono)<=2)&&(cortes_bis(aux3->p,punt,poligono)<=2))
       {pcortes=Anadir(pcortes,punt);}
    }
   }
   if(cortanrectasP(aux1->p,aux->p,aux3->sig->sig->p,aux3->p,&x,&y)==true)
   {punt.x=x;
    punt.y=y;
    punt.tipo='I';
    if(Iguales(aux1->p,punt)==false)
    {if((x>0.0)&&(x<1.0)&&(y>0.0)&&(y<1.0))
      if(((fmod(cortes(punt,poligono),2.0)==0.0)||(Es_Ordenada_vertice(punt,poligono)==true))==false)
       if((cortes_bis(aux->p,punt,poligono)<=2)&&(cortes_bis(aux3->p,punt,poligono)<=2))
       {pcortes=Anadir(pcortes,punt);}
    }
   }
   if(cortanrectasP(aux2->p,aux->p,aux3->sig->p,aux3->p,&x,&y)==true)
   {punt.x=x;
    punt.y=y;
    punt.tipo='I';
    if(Iguales(aux2->p,punt)==false)
    {if((x>0.0)&&(x<1.0)&&(y>0.0)&&(y<1.0))
      if(((fmod(cortes(punt,poligono),2.0)==0.0)||(Es_Ordenada_vertice(punt,poligono)==true))==false)
       if((cortes_bis(aux->p,punt,poligono)<=2)&&(cortes_bis(aux3->p,punt,poligono)<=2))
        {pcortes=Anadir(pcortes,punt);}
     }
   }
   if(cortanrectasP(aux2->p,aux->p,aux3->sig->sig->p,aux3->p,&x,&y)==true)
   {punt.x=x;
    punt.y=y;
    punt.tipo='I';
    if(Iguales(aux2->p,punt)==false)
    {if((x>0.0)&&(x<1.0)&&(y>0.0)&&(y<1.0))
      if(((fmod(cortes(punt,poligono),2.0)==0.0)||(Es_Ordenada_vertice(punt,poligono)==true))==false)
       if((cortes_bis(aux->p,punt,poligono)<=2)&&(cortes_bis(aux3->p,punt,poligono)<=2))
       {pcortes=Anadir(pcortes,punt);}
    }
   }
   aux3=aux3->sig->sig->sig;
  }

//Antes de seguir miramos si la prolongación de los lados cóncavos corta al polígono

 aux4=poligono;
 m1=2; m2=2;
 while(aux4->sig!=NULL)
 {if((Iguales(aux1->p,aux4->p)==false)&&(Iguales(aux1->p,aux4->sig->p)==false)
    &&(Iguales(aux->p,aux4->p)==false)&&(Iguales(aux->p,aux4->sig->p)==false))
   if(cortanrectas(aux1->p,aux->p,aux4->p,aux4->sig->p,&x,&y)==true)
   {puntaux.x=x;
    puntaux.y=y;
    if(modulo(aux->p,puntaux)<m1)
    {m1=modulo(aux->p,puntaux);
     punt1=puntaux;
    }
   }
  if((Iguales(aux2->p,aux4->p)==false)&&(Iguales(aux2->p,aux4->sig->p)==false)
    &&(Iguales(aux->p,aux4->p)==false)&&(Iguales(aux->p,aux4->sig->p)==false))
   if(cortanrectas(aux2->p,aux->p,aux4->p,aux4->sig->p,&x,&y)==true)
   {puntaux.x=x;
    puntaux.y=y;
    if(modulo(aux->p,puntaux)<m2)
    {m2=modulo(aux->p,puntaux);
     punt2=puntaux;
    }
   }
   aux4=aux4->sig;
 }
                                    //Ahora el último lado

  if((Iguales(aux1->p,aux4->p)==false)&&(Iguales(aux1->p,poligono->p)==false)
    &&(Iguales(aux->p,aux4->p)==false)&&(Iguales(aux->p,poligono->p)==false))
```

```
  if(cortanrectas(aux1->p,aux->p,aux4->p,poligono->p,&x,&y)==true)
  {puntaux.x=x;
   puntaux.y=y;
   if(modulo(aux->p,puntaux)<m1)
   {m1=modulo(aux->p,puntaux);
    punt1=puntaux;
   }
  }

 if((Iguales(aux2->p,aux4->p)==false)&&(Iguales(aux2->p,poligono->p)==false)
   &&(Iguales(aux->p,aux4->p)==false)&&(Iguales(aux->p,poligono->p)==false))
  if(cortanrectas(aux2->p,aux->p,aux4->p,poligono->p,&x,&y)==true)
  {puntaux.x=x;
   puntaux.y=y;
   if(modulo(aux->p,puntaux)<m2)
   {m2=modulo(aux->p,puntaux);
    punt2=puntaux;
   }
  }
                                    //Ahora ya avanzamos
 punt1.tipo='C';
 punt2.tipo='C';
 if(m1!=2)pcortes=AnadirR(pcortes,punt1);
  if(m2!=2)pcortes=AnadirR(pcortes,punt2);
   aux=aux->sig->sig->sig;
}

//Ahora calculamos el mejor de los puntos de corte, es decir, de la lista pcortes

 if(pcortes==NULL) //Si pcortes==NULL el polígono es convexo
 {aux=poligono;
  while(aux!=NULL)
  {pcortes=Anadir(pcortes,aux->p);
   aux=aux->sig;printf('.');
  }
 }
 aux=pcortes;
 fnube=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\nube.txt'',''w+'');
 while(aux!=NULL)
 {if (aux->p.tipo=='V')
   fprintf(fnube,''%f %f %d\n'',aux->p.x,aux->p.y,1);
  else
   fprintf(fnube,''%f %f %d\n'',aux->p.x,aux->p.y,2);
  aux=aux->sig;
 }
 fclose(fnube);
 aux=pcortes;
 i=1;
 while(aux!=NULL)
 {paux=aux->p;
  Candidato(poligono,&(aux->p),0.0001);
  polvisiaux1=(lista*) malloc(sizeof(lista));
  polvisiaux1=calpolvisi(aux->p,poligono,polvisiaux1);
  areaux=areapolvisi(aux->p,polvisiaux1);
  printf(''\n\t\t %d Punton (%f,%f)-(%f,%f) %c AREA=\t %.5lf'',i,paux.x,paux.y,aux->p.x,aux->p.y,
         aux->p.tipo,areaux);
  ++i;
  if( areaux>area)
  {polvisi=Limpiar(polvisi);
   polvisi=polvisiaux1;
   area=areaux;
   *p=aux->p;
  }
  aux=aux->sig;
 }
                                    //Ahora lo guardamos en el fichero
 *a=area;
 visidet=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisibose.txt'',''w+'');
 aux=polvisi;
 while(aux!=NULL)
 {fprintf(visidet,''%f %f\n'',aux->p.x,aux->p.y);
  aux=aux->sig;
 }
                                    //Guardamos el punto y area en el fichero para pintarlo

 fprintf(visidet,''%lf %lf\n'',(*p).x,(*p).y);
 fprintf(visidet,''%lf %lf\n'',area,((*a)*100)/ar);
```

```
 fclose(visidet);
 aux=Limpiar(aux);
 return(polvisi);
}
```

Incluimos a continuación los códigos que han permitido construir la curva de crecimiento del porcentaje de área iluminada por una *luz-punto* interior a un polígono P, en función del número n de vértices de éste. Según se mostró en la Sección 7.7 esta curva es de la forma $\alpha e^{-\beta n}$, respondiendo así al problema PorA-p-Pv1(P).

Códigos para obtener datos sobre el problema PorA-p-Pv1(P)

Incluimos en primer lugar el código utilizado para obtener resultados mediante la heurística *simulated annelaig-SA* y a continuación las implementaciones para obtener la tabla también de porcentajes con la heurística *random search-RS*. Los códigos para generar y leer bases de datos de polígonos aleatorios, utilizando nuestro generador aleatorio RPG, se encuentran en la sección de códigos generales de este apéndice.

1. **Mediante simulated annealing-SA**

```
void Resultados_MaxRST0_PV1Datos(void)
{FILE *Resultados;
 punto puntoreal;
 int vertices,polig,i,itera;
 lista *poligonoreal,*polvisireal;
 double apreal,apvisi;
 poligono *poligonos,*aux;
 char c;
 char cadena[100],nombre[100];
                                    //Pedimos datos de los polígonos
 printf(''\n * Cuantos vertices?'');
scanf(''%d'',&vertices);
printf(''\n * Nombre del fichero de datos?'');
 fflush(stdin);
 gets(nombre);
 printf(''\n * Cuantos llevamos?'');
 scanf(''%d'',&polig);

 poligonos=NULL;
 poligonos=Lectura_Poligonos(poligonos,nombre);
 aux=poligonos;
 printf(''\n Poligonos leidos satisfactoriamente'');
 getche();
 i=1;
 while(i<=polig)
 {++i;
  aux=aux->sig;
 }
 printf(''\n Situados en el poligono %d'',i);
 getche();

 while(aux!=NULL)
 {Resultados=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab
                    \\Result_MaxRST0_PV1.txt'',''a'');
  if(aux==poligonos)
   fprintf(Resultados,''\n'');
  poligonoreal=aux->dat.puntos;
  apreal=area_poligono(poligonoreal);
  printf('' 3 '');
                                    //busco el mejor punto
  printf('' 4 '');
  polvisireal=(lista*) malloc(sizeof(lista));
  polvisireal->sig=NULL;
  polvisireal=MaxRs_PV1(0,100,0.9,&itera,poligonoreal,polvisireal,&puntoreal,&apvisi,apreal);
```

```
 polvisireal=Limpiar(polvisireal);printf(''+'');
  poligonoreal=Limpiar(poligonoreal);printf(''+'');
                                    //guardamos el porcentaje en el fichero
  fprintf(Resultados,''%lf '',((apvisi)*100)/apreal);
  polig=polig+1;
  printf(''\n Ejecutado....vertices:\t%d poligono:\t%d - %lf'',vertices,polig,((apvisi)*100)/apreal);
  fclose(Resultados);
  printf(''\n * Otro ?'');
  fflush(stdin);
  c='S';
 aux=aux->sig;
 }
                                    //hemos terminado todos los polígonos de un número de vértices
}
```

2. **Mediante random search-RS**

```
void Resultados_MaxDet_PV1Datos(void)
{FILE *Resultados;
 punto puntoreal;
 int vertices,polig,i;
 lista *poligonoreal,*polvisireal,*nubereal;
 double apreal,apvisi;
 poligono *poligonos,*aux;
 char c;
 char cadena[100],nombre[100];
 printf(''\n * Cuantos vertices?'');
 scanf(''%d'',&vertices);
 printf(''\n * Nombre del fichero de datos?'');
 fflush(stdin);
 gets(nombre);
 printf(''\n * Cuantos llevamos?'');
 scanf(''%d'',&polig);
 poligonos=NULL;
 poligonos=Lectura_Poligonos(poligonos,nombre);
 aux=poligonos;
 printf(''\n Poligonos leidos satisfactoriamente'');
 getche();
 i=1;
 while(i<=polig)
 {++i;
  aux=aux->sig;
 }
 printf(''\n Situados en el poligono %d'',i);
 getche();

 while(aux!=NULL)
{Resultados=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab
                    \\Result_MaxDet_PV1.txt'',''a'');
 if(aux==poligonos)
 fprintf(Resultados,''\n'');
 poligonoreal=aux->dat.puntos;
 apreal=area_poligono(poligonoreal);
 nubereal=(lista*) malloc(sizeof(lista));
 creanube(nubereal,poligonoreal,1000);
                                    //busco el mejor punto
 polvisireal=(lista*) malloc(sizeof(lista));
 polvisireal->sig=NULL;
 polvisireal=MaxDet_PV1(nubereal,poligonoreal,polvisireal,&puntoreal,&apvisi,apreal);
 nubereal=Limpiar(nubereal);
 polvisireal=Limpiar(polvisireal);printf(''+'');
 poligonoreal=Limpiar(poligonoreal);printf(''+'');
                                    //guardamos el porcentaje en el fichero
 fprintf(Resultados,''%lf '',((apvisi)*100)/apreal);
 polig=polig+1;
 printf(''\n Ejecutado....vertices:\t%d poligono:\t%d - %lf'',vertices,polig,((apvisi)*100)/apreal);
 fclose(Resultados);
 printf(''\n * Otro ?'');
 fflush(stdin);
```

```
       c='S';
       aux=aux->sig;
       }                              //hemos terminado todos los polígonos de un número de vértices
      }
```

Código para la construcción de la superficie de áreas de un polígono

La *superficie de áreas* de un polígono P es la superficie que se obtiene al asociar a cada punto de un polígono su porcentaje de área iluminada. Así si (x, y, z), es un punto de dicha superficie las dos primeras coordenadas representan un punto interior a P, mientras z representan el porcentaje de área iluminada por dicho punto. La construcción de la superficie de áreas nos permitió establecer la Conjetura 7.8.3. Con el siguiente código se obtienen los puntos de la superficie, que se podrán visualizar con el correspondiente programa de Matlab, incluido en la Sección B.9 de este Apéndice.

```
/*-------------------------------------------------------------------------------------
Diseñamos una pequeña función que nos permitirá construir la superficie de áreas.Es decir
tenemos un polígono, generamos una nube en su interior, calculamos el área iluminada  por
todos los puntos de la nube, construimos el delaunay, (esto en matlab),de las coordenadas
x,y y lo levantamos hacia arriba, para ver que forma puede tener la superficie de  áreas.
En la función solamente calculamos para cada punto de la nube  su  área   iluminada y  lo
guardamos todo en el fichero superficie.txt
-------------------------------------------------------------------------------------*/

void Superficie(lista *nube,lista *poligono)
{lista *aux,*polvisiaux1;
 double areaux=0,area=0;
 FILE *super;
 int i=1;
 aux=nube;
 super=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\superficie.txt'',''w+'');

 while(aux!=NULL)
 {polvisiaux1=(lista*) malloc(sizeof(lista));
  polvisiaux1=calpolvisi(aux->p,poligono,polvisiaux1);
  areaux=areapolvisi(aux->p,polvisiaux1);
  fprintf(super,''%lf %lf %lf\n'',aux->p.x,aux->p.y,areaux);
  printf(''%lf %lf %lf\n'',aux->p.x,aux->p.y,areaux);
  polvisiaux1=Limpiar(polvisiaux1);
  aux=aux->sig;
 }
                                    //Ahora lo guardamos en el fichero

  fclose(super);
 }

/*-------------------------------------------------------------------------------------
Vamos a diseñar un programa igual al anterior, pero generando una malla, que nos  permita
dibujarlo en matlab
-------------------------------------------------------------------------------------*/

void SuperficieBis(lista *poligono)
{lista *polvisiaux1;
 double areaux=0;
 FILE *super;
 float x,y;
 punto p;
 super=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\superficie.txt'',''w+'');
 y=0.0;
 while(y<=1.0)
 {x=0.0;
  while(x<=1.0)
  {areaux=0.0;
   p.x=x;
   p.y=y;
   if(punto_int(p,poligono)==true)
   {polvisiaux1=(lista*) malloc(sizeof(lista));
    polvisiaux1=calpolvisi(p,poligono,polvisiaux1);
```

```
    areaux=areapolvisi(p,polvisiaux1);
    polvisiaux1=Limpiar(polvisiaux1);
   }
   fprintf(super,''%lf '',areaux);
   printf(''%.3f '',x);
   x=x+0.005;
  }
  fprintf(super,''\n'');
  printf(''\n'');
  y=y+0.005;
 }
 fclose(super);
}
```

B.5 Implementaciones relacionadas con MaxA-p-Pvk(P, k)

Presentamos los códigos relacionados con el problema MaxA-p-Pvk(P, k), es decir, el problema de la búsqueda de k puntos de máxima iluminación. Las implementaciones de las heurísticas siguen la misma a la presentada en el Capítulo 8. Antes de presentar los códigos de la heurística exponemos la implementaciones para construir la *unión* de k *polígonos de visibilidad* mediante el algoritmo de *Weiler-Atherton* analizado en la Sección 8.2.

Unión de polígonos de visibilidad

Incluimos en primer lugar un conjunto de funciones auxiliares, como por ejemplo las que permiten construir las listas circulares mencionadas en el algoritmo de *Weiler-Atherton*, que permitirán finalmente construir una función para calcular el polígono, (no conexo y con agujeros), que puede producir la unión de k *polígonos de visibilidad*.

```
/*-----------------------------------------------------------------------------------
Las siguientes funciones nos permiten el manejo de segmentos, cortes y pertenencia.
-----------------------------------------------------------------------------------*/

bool pertenece_segm(punto p, punto p1,punto p2)
{if(Iguales(p,p1)||Iguales(p,p2))
  return(true);
 else
  if(long double(fabs(modulo(p,p1)+modulo(p,p2)-modulo(p1,p2)))<=0.000000000000001)
   return(true);
  else
   return(false);
}

/*-----------------------------------------------------------------------------------
La siguiente función produce los cortes de dos segmentos.Si son iguales se  considera que
hay dos cortes.
-----------------------------------------------------------------------------------*/
lista *cortesegmentos(punto p1,punto p2, punto p3, punto p4,lista *l)
{double v1,v2,w1,w2;
 punto p;
 v1=p2.x-p1.x;
 v2=p2.y-p1.y;
 w1=p4.x-p3.x;
 w2=p4.y-p3.y;
 l=NULL;
 if(pertenece_segm(p1,p3,p4)==true)
  l=Anadir(l,p1);
 if(pertenece_segm(p2,p3,p4)==true)
  l=Anadir(l,p2);
 if(pertenece_segm(p3,p1,p2)==true)
  l=Anadir(l,p3);
 if(pertenece_segm(p4,p1,p2)==true)
  l=Anadir(l,p4);
```

```
 if((l==NULL)&&((vectorial(p1,p3,p4)/vectorial(p2,p3,p4))<=0.0)&&
   ((vectorial(p3,p1,p2)/vectorial(p4,p1,p2))<=0.0))
 {p.x=(p3.x*(p4.y*(p1.x-p2.x)-p1.x*p2.y+p1.y*p2.x)-p4.x*(p3.y*(p1.x-p2.x)-p1.x*p2.y+p1.y*p2.x))/
      (p3.x*(p1.y-p2.y)+p3.y*(p2.x-p1.x)+p4.x*(p2.y-p1.y)+p4.y*(p1.x-p2.x));
  p.y=(p3.x*p4.y*(p1.y-p2.y)-p3.y*(p4.x*(p1.y-p2.y)+p1.x*p2.y-p1.y*p2.x)+p4.y*(p1.x*p2.y-p1.y*p2.x))/
      (p3.x*(p1.y-p2.y)+p3.y*(p2.x-p1.x)+p4.x*(p2.y-p1.y)+p4.y*(p1.x-p2.x));
  if((pertenece_segm(p,p1,p2)==true)&&(pertenece_segm(p,p3,p4)==true))
   l=Anadir(l,p);
 }
 return(l);
}

/*-------------------------------------------------------------------------------------
La siguiente función nos dice si un punto es interior a un polígono.
-------------------------------------------------------------------------------------*/

bool punto_int(punto p,lista *poligono)
{if((fmod(cortes(p,poligono),2.0)!=0.0)&&(Pertenece(p,poligono)==false))
  return(true);
 else                          //Los vértices se consideran interiores
 return(false);
}

/*-------------------------------------------------------------------------------------
Diseñamos ahora una función que nos dice si un polígono está incluido  en el  interior de
otro. Podríamos tener esta situación.
-------------------------------------------------------------------------------------*/

bool Incluido(lista *pol1, lista *pol2)
{int i,j;
 lista *aux;
 aux=pol1;
 j=0;
 while(aux!=NULL)
 {if(punto_int(aux->p,pol2)==true)
   ++j;
  aux=aux->sig;
 }
 i=longitud(pol1);
 if(i==j)
  return(true);
 else
  return(false);
}

bool Hay_interseccion(lista *pol1,lista *pol2)
{lista *aux;
 int i=0;
 aux=pol1;
 while (aux->sig!=NULL)
 {i=i+cortes_bis(aux->p,aux->sig->p,pol2);
  aux=aux->sig;
 }
 if(i!=0)
  return(true);
 else
  return(false);
}

/*-------------------------------------------------------------------------------------
La siguiente función añade un elemento a una lista.
-------------------------------------------------------------------------------------*/

lista *Anadir(lista *l,punto p)
{lista *aux;
 if(l==NULL)
 {l=(lista*)malloc(sizeof(lista));
  l->p=p;
  l->sig=NULL;
 }
 else
 {aux=l;
  if(Pertenece(p,l)==false) //Para que no se repitan vértices
  {while(aux->sig!=NULL)
     aux=aux->sig;
   aux->sig=(lista*)malloc(sizeof(lista));
   aux=aux->sig;
   aux->p=p;
```

```
   aux->sig=NULL;
  }
 }
 return(l);
}

/*-------------------------------------------------------------------------------------
La siguiente función une dos lista.
-------------------------------------------------------------------------------------*/

lista *unelistas(lista *l1,lista *l2)
{lista *aux;
 aux=l2;
 while (aux!=NULL)
 {l1=Anadir(l1,aux->p);
  aux=aux->sig;
 }
 return(l1);
}

/*-------------------------------------------------------------------------------------
Reproducimos aquí la función que calcula el área de un polígono, pero sin la  parte final
para guardar en el fichero. LLamaremos a esta función area_poligonobis.
-------------------------------------------------------------------------------------*/

double area_poligonobis(lista *poligono)
{double ar=0;
 lista *aux,*aux1,*aux2,*aux3,*copia;
                                  //Hacemos una copia del poligono para no perderlo
 if (longitud(poligono)>=3)
 {aux=poligono;
  if(aux!=NULL)
  {copia=(lista*) malloc(sizeof(lista));
   copia->p=aux->p;
   aux=aux->sig;
   aux1=copia;
  }
  while(aux!=NULL)
  {aux1->sig=(lista*) malloc(sizeof(lista));
   aux1=aux1->sig;
   aux1->p.x=aux->p.x;
   aux1->p.y=aux->p.y;
   aux=aux->sig;
  }
  aux1->sig=NULL;
                                  //Vamos buscando orejas y
  while(longitud(copia)>3)
  {aux1=copia;
   aux2=copia->sig;
   aux3=copia->sig->sig;
   while(es_oreja(aux1->p,aux2->p,aux3->p,copia)==false)
   {if(aux3->sig==NULL)
    {aux1=aux1->sig;
                                  //La oreja está en el último triángulo
     aux2=aux2->sig;
     aux3=copia;
    }
    else
     if(aux2->sig==NULL)
     {aux1=aux1->sig;
      aux2=copia;
      aux3=aux3->sig;
     }
     else
      if(aux1->sig==NULL)
      {aux1=copia;
       aux2=aux2->sig;
       aux3=aux3->sig;
      }
      else
      {aux1=aux1->sig;
       aux2=aux2->sig;
       aux3=aux3->sig;}
      }
    ar=ar+ area(aux1->p,aux2->p,aux3->p);
                                  //Sumamos las area de triángulos
    aux=copia;
    while(aux!=NULL)
```

```
    {if(aux->sig==aux2)
     {aux->sig=aux->sig->sig;
      free(aux2);
     }
     aux=aux->sig;
    }
   }
                                    //Sumamos el área del último triangulo
  ar=ar+area(copia->p,copia->sig->p,copia->sig->sig->p);
                                    //Lo guardamos en la última fila del fichero
  copia=Limpiar(copia);
 }/*del primer if*/
 return(ar);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función tal que dado un segmento con un par de puntos y un polígono  nos da
como solución una lista con los puntos de corte con el  polígono  ordenados  de  menor  a
mayor por la distancia al punto.
-------------------------------------------------------------------------------------*/

lista *lista_cortes(lista *solu,punto p1,punto p2,lista *poligono)
{lista *aux,*solaux,*aux1;
 punto vector[Tamcierre];
 punto p;
 double x,y;
 int i=0,m,n;
 aux=poligono;
 solu=NULL;
 if(aux!=NULL)
 {while(aux->sig!=NULL)
                                    //Vamos mirando lo cortes con los lados del polígono
  {solaux=cortesegmentos(p1,p2,aux->p,aux->sig->p,solaux);
   if(solaux!=NULL)
   {aux1=solaux;
    while(aux1!=NULL)
    {vector[i].x=aux1->p.x;
     vector[i].y=aux1->p.y;
     vector[i].tipo='c';
     i=i+1;
     aux1=aux1->sig;
    }
   }
   aux=aux->sig;
  }
                                    //Miramos el último lado
  solaux=cortesegmentos(p1,p2,aux->p,poligono->p,solaux);
  if(solaux!=NULL)
  {aux1=solaux;
   while(aux1!=NULL)
   {vector[i].x=aux1->p.x;
    vector[i].y=aux1->p.y;
    vector[i].tipo='c';
    i=i+1;
    aux1=aux1->sig;
   }
  }
                                    //Ordenamos el vector por el método de la burbuja
  for(m=0;m<i-1;++m)
   for(n=m+1;n<i;++n)
    if(modulo(p1,vector[m])>modulo(p1,vector[n]))
    {p=vector[m];
     vector[m]=vector[n];
     vector[n]=p;
    }
                                    //Lo guardamos en una lista
  n=0;
  if(i>0)
  {solu=(lista*)malloc(sizeof(lista));
   solu->p.x=vector[n].x;
   solu->p.y=vector[n].y;
   solu->p.tipo=vector[n].tipo;
   aux=solu;
   aux->sig=NULL;
  }
  ++n;
  while(n<i)
  {aux->sig=(lista*)malloc(sizeof(lista));
```

```
   aux=aux->sig;
   aux->p.x=vector[n].x;
   aux->p.y=vector[n].y;
   aux->p.tipo=vector[n].tipo;
   aux->sig=NULL;
   ++n;
  }
 }
 return(solu);
}

/*-------------------------------------------------------------------------------------
Diseñamos ahora una función que crea una lista circular según indica el algoritmo con los
vértices seguidos de sus puntos de intersección hasta el vértice siguiente.
-------------------------------------------------------------------------------------*/

lista *lista_circular(lista *solu,lista *poligono1, lista *poligono2)
{lista *aux,*cortes;
 solu=NULL;
 cortes=NULL;
 if(poligono1==NULL)
  solu=NULL;
 else
 {aux=poligono1;
  while(aux->sig!=NULL)
  {cortes=lista_cortes(cortes,aux->p,aux->sig->p,poligono2);
   if((Pertenece(aux->p,cortes)==true))
    aux->p.tipo='c';
   solu=Anadir(solu,aux->p);
   solu=unelistas(solu,cortes);
   cortes=Limpiar(cortes);
   aux=aux->sig;
  }
                                  //Ahora el último lado
  cortes=lista_cortes(cortes,aux->p,poligono1->p,poligono2);
  if((Pertenece(aux->p,cortes)==true))
   aux->p.tipo='c';
  solu=Anadir(solu,aux->p);
  solu=unelistas(solu,cortes);
  cortes=Limpiar(cortes);
                                  //Ahora la hacemos circular
  aux=solu;
  while(aux->sig!=NULL) aux=aux->sig;
   aux->sig=solu;
 }
 return(solu);
}/*fin lista_circular*/

/*-------------------------------------------------------------------------------------
Diseñamos ahora una función que crea una lista circular según indica el algoritmo con. En
esta parte vamos a implementar todo los relativo a la unión de k polígonos. Aquí termina-
remos teniendo una función que nos dará el área de la unión de  k  polígonos. Empezaremos
diseñando un algoritmo para la unión de dos polígonos generales que pueden tener agujeros
,(recordemos que la unión de polígonos de visibilidad puede ser un polígono con agujeros.
Dèspués iteraremos este proceso para calcular la intersección de k polígonos de visibili-
dad. Esta función será la base para calcular un fitness para calcular los k puntos de má-
xima iluminación. Con esta función  minimizar el nºde luces que  iluminarán un  polígono
en un porcentaje elevado será fácil.Creamos una función que nos da la  lista de cortes.
-------------------------------------------------------------------------------------*/

lista *lista_cortes_total(lista *solu,lista *poligono1, lista *poligono2)
{lista *aux,*cortes;
 solu=NULL;
 cortes=NULL;
 if(poligono1==NULL)
  solu=NULL;
 else
 {aux=poligono1;
  while(aux->sig!=NULL)
  {cortes=lista_cortes(cortes,aux->p,aux->sig->p,poligono2);
   solu=unelistas(solu,cortes);
   cortes=Limpiar(cortes);
   aux=aux->sig;
  }
                                  //Ahora el último lado
  cortes=lista_cortes(cortes,aux->p,poligono1->p,poligono2);
  solu=unelistas(solu,cortes);
  cortes=Limpiar(cortes);
```

```
 }
 return(solu);
}/*fin lista_cortes_total*/

/*-----------------------------------------------------------------------------------
La siguiente función elimina un punto de una lista.
-----------------------------------------------------------------------------------*/

lista *eliminar(punto p,lista *l)
{lista *aux1,*aux2;
 if(Pertenece(p,l)==true)
 {if(Iguales(l->p,p)==true)
  {aux1=l;
   l=l->sig;
   free(aux1);
  }
  else
  {aux2=l;
   aux1=l->sig;
   while(Iguales(aux1->p,p)==false)
   {aux2=aux1;
    aux1=aux1->sig;
   }
   aux2->sig=aux1->sig;
   free(aux1);
  }
 }
 return(l);
}

/*-----------------------------------------------------------------------------------
Diseñamos  una función que añade un nodo a un polígono general, es decir, un  conjunto de
componentes conexas con agujeros.
-----------------------------------------------------------------------------------*/

poligono *Anadir_Nodo(poligono *p,nodo n)
{poligono *aux;
 if(longitud(n.puntos)>=3)
 {if(p==NULL)
  {p=(poligono*)malloc(sizeof(poligono));
   p->dat=n;
   p->sig=NULL;
  }
  else
  {aux=p;
   while(aux->sig!=NULL)
    aux=aux->sig;
   aux->sig=(poligono*)malloc(sizeof(poligono));
   aux=aux->sig;
   aux->dat=n;
   aux->sig=NULL;
  }
 }
 return(p);
}

/*-----------------------------------------------------------------------------------
Unimos dos polígonos generales
-----------------------------------------------------------------------------------*/

poligono *Juntar_Poligonos(poligono *p1,poligono *p2)
{poligono *aux;
 aux=p2;
 while(aux!=NULL)
 {p1=Anadir_Nodo(p1,aux->dat);
  aux=aux->sig;
 }
 return(p1);
}

/*-----------------------------------------------------------------------------------
A continuación implementamos las funciones que permiten unir componentes del polígono con
un nuevo polígono de visibilidad que es una componente principal.Se distinguen dos casos:
unión con un agujeros y unión con una componente principal.
-----------------------------------------------------------------------------------*/

poligono *Union_Nodo_Agujero(poligono *solu, nodo n,lista *l)
{lista *circular1,*circular2,*aux1,*aux2,*aux;
```

```
 solu=NULL;
 nodo final;
 punto p;
 int i,esta;
 if(Hay_interseccion(n.puntos,l)==false)
  solu=Anadir_Nodo(solu,n);
 else
 {final.tipo='A';
  final.puntos=NULL;
                                    //Un agujero solo puede generar un único agujero
  circular1=NULL;
  circular2=NULL;
  circular1=lista_circular(circular1,l,n.puntos);
  circular2=lista_circular(circular2,n.puntos,l);
  aux1=circular1;
  while(aux1->p.tipo!='c') aux1=aux1->sig;
   aux=aux1;
  esta=1;
  i=1;
  while((Iguales(aux1->p,aux->p)==false)||(i==1))
  {++i;
   p=aux1->p;
   p.tipo='v';
   final.puntos=Anadir(final.puntos,p);
   if(aux1->p.tipo=='c')
   {if(esta==1)
     aux2=circular2;
    else
     aux2=circular1;
    while(Iguales(aux2->p,aux1->p)==false) aux2=aux2->sig;
    if((vectorial(aux1->p,aux1->sig->p,aux2->sig->p)<0)&&(aux1->sig!=aux))
    {aux1=aux2->sig;
     if(esta==1)
      esta=2;
     else
      esta=1;
    }
    else
     aux1=aux1->sig;
   }
   else
    aux1=aux1->sig;
  }
 }
 solu=Anadir_Nodo(solu,final);
 return(solu);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función igual que la de nodo principal para agujeros.
-------------------------------------------------------------------------------------*/

poligono *Union_Nodo_Agujero2(poligono *solu, nodo n,lista *l)
{lista *circular1,*circular2,*cortes,*cortes1,*aux,*aux1,*aux2,*aux3,*aux4;
 solu=NULL;
 nodo final;
 int i,esta;
 punto p;
 bool fin;
 time_t ini,para,ini2,para2;
 cortes=NULL;
 cortes=lista_cortes_total(cortes,n.puntos,l);
 cortes1=lista_cortes_total(cortes1,l,n.puntos);
 cortes=unelistas(cortes,cortes1);
 final.puntos=NULL;
 if((n.puntos!=NULL)&&(l!=NULL))
 {if(Incluido(l,n.puntos)==true)
   solu=Anadir_Nodo(solu,n);
  else
   if((cortes==NULL)&&(Incluido(n.puntos,l)==false))
    solu=Anadir_Nodo(solu,n);
   else
    if(cortes!=NULL)
    {circular1=NULL;
     circular2=NULL;
     circular1=lista_circular(circular1,l,n.puntos);
     circular2=lista_circular(circular2,n.puntos,l);
     circular1=contrastar(circular1,circular2,'c');
     circular2=contrastar(circular2,circular1,'c');
```

```
     cortes=contrastar(cortes,circular1,'l');
     cortes=contrastar(cortes,circular2,'l');
     ini2=time(NULL);
     while(cortes!=NULL)
     {para2=time(NULL);
      if(difftime(para2,ini2)>4.0) {printf(`` 2 '');solu=NULL;goto fin1;}
       final.puntos=NULL;
      aux1=circular1;
      while(Iguales(aux1->p,cortes->p)==false)
       aux1=aux1->sig;
      aux=aux1;
      esta=1;
      i=1;
      ini=time(NULL);
      while((Iguales(aux1->p,aux->p)==false)||(i==1))
      {para=time(NULL);
       ++i;
       p=aux1->p;
       p.tipo='v';
       final.puntos=Anadir(final.puntos,p);
       if(aux1->p.tipo=='c')
       {cortes=eliminar(aux1->p,cortes);
        if(esta==1)
         aux2=circular2;
        else
         aux2=circular1;
        while(Iguales(aux2->p,aux1->p)==false)
         aux2=aux2->sig;
        if((vectorial(aux1->p,aux1->sig->p,aux2->sig->p)<0.0)&&(aux1->sig!=aux))
        {aux1=aux2->sig;
         if(esta==1)
          esta=2;
         else
          esta=1;
        }
        else
         aux1=aux1->sig;
       }
       else
        aux1=aux1->sig;
      }
      if(final.puntos!=NULL)
      {final.puntos=desbrozar(final.puntos);
       final.tipo='A';
       solu=Anadir_Nodo(solu,final);
      }
     }//1ºWhile
     cortes=Limpiar(cortes);
     cortes1=Limpiar(cortes1);
                                  //Limpiamos cortes y listas circulares
     aux=circular1;
     while(aux->sig!=circular1)
      aux=aux->sig;
     aux->sig=NULL;
     circular1=Limpiar(circular1);
     aux=circular2;
     while(aux->sig!=circular2) aux=aux->sig;
     aux->sig=NULL;
     circular2=Limpiar(circular2);
    }//else
   }
  return(solu);
}

/*-------------------------------------------------------------------------------------
Una  función para determinar si un polígono es principal o un agujero. Lo haremos mirando
el sentido de giro.Se genera un punto interior al polígono tal que los segmentos que unen
este punto con los dos primeros puntos del polígono estén dentro del polígono. Después se
mira el sentido de giro.
-------------------------------------------------------------------------------------*/

char Tipo_Pol(lista *polig)
{lista *aux,*ymin;
 char c;
                                  //Buscamos el nodo de ordenada mínima
 ymin=polig;
 aux=polig;
 while(aux!=NULL)
 {if(aux->p.y<ymin->p.y)
```

```
   ymin=aux;
  aux=aux->sig;
 }
                                  //Hacemos la lista circular
 aux=polig;
 while(aux->sig!=NULL) aux=aux->sig;
  aux->sig=polig;
                                  //Buscamos ymin
 aux=polig;
 while(aux->sig!=ymin)
  aux=aux->sig;
                                  //Podemos calcular el prodcuto vectorial de los
                                  //tres vértices. El más bajo y los adyacentes
 if(vectorial(ymin->p,aux->p,ymin->sig->p)<=0.00001)
  c='P';
 else
  c='A';
                                  //Quitamos la circularidad
 aux=polig;
 while(aux->sig!=polig) aux=aux->sig;
  aux->sig=NULL;
 return(c);
}

/*-------------------------------------------------------------------------------------
Función de unión con nodo principal.
-------------------------------------------------------------------------------------*/

poligono *Union_Nodo_Principal(poligono *solu, nodo n,lista *l,char *c)
{lista *circular1,*circular2,*cortes,*cortes1,*aux,*aux1,*aux2,*aux3,*aux4;
 solu=NULL;
 nodo final;
 int i,esta;
 punto p;
 bool fin;
 cortes=NULL;
 cortes=lista_cortes_total(cortes,n.puntos,l);
 cortes1=lista_cortes_total(cortes1,l,n.puntos);
 cortes=unelistas(cortes,cortes1);
 if(cortes==NULL)
  *c='N';
 else
  *c='S';
 final.puntos=NULL;
 if((n.puntos!=NULL)&&(l!=NULL))
 {if((Incluido(n.puntos,l)==true)&&(*c=='N'))
  {final.tipo='P';
   final.puntos=l;
   solu=Anadir_Nodo(solu,final);
                                  //Miramos si los polígonos están incluidos
  }
  else
   if((Incluido(l,n.puntos)==true)&&(*c=='N'))
   {*c='I';
    solu=Anadir_Nodo(solu,n);
   }
   else
    if(cortes==NULL)
    {final.tipo='P';
     final.puntos=l;
     solu=Anadir_Nodo(solu,final);
     solu=Anadir_Nodo(solu,n);
    }
    else
    {circular1=NULL;
     circular2=NULL;
     circular1=lista_circular(circular1,l,n.puntos);
     circular2=lista_circular(circular2,n.puntos,l);
     ini2=time(NULL);
     while(cortes!=NULL)
     {final.puntos=NULL;
      aux1=circular1;
      ini3=time(NULL);
      while(Iguales(aux1->p,cortes->p)==false)
       aux1=aux1->sig;
      aux=aux1;
      esta=1;
      i=1;
      while((Iguales(aux1->p,aux->p)==false)||(i==1))
```

```
      {para=time(NULL);
       ++i;
       p=aux1->p;
       p.tipo='v';
       final.puntos=Anadir(final.puntos,p);
       if(aux1->p.tipo=='c')
       {cortes=eliminar(aux1->p,cortes);
        if(esta==1)
         aux2=circular2;
        else
         aux2=circular1;
        while(Iguales(aux2->p,aux1->p)==false)
         aux2=aux2->sig;
       }
       if((vectorial(aux1->p,aux1->sig->p,aux2->sig->p)<0.0))
       {aux1=aux2->sig;
        if(esta==1)
         esta=2;
        else
         esta=1;
       }
       else
        aux1=aux1->sig;
      }
      else
       aux1=aux1->sig;
     }
     if(final.puntos!=NULL)
     {final.puntos=desbrozar(final.puntos);
      final.tipo=Tipo_Pol(final.puntos);
      solu=Anadir_Nodo(solu,final);
     }
    }//1ºWhile
    cortes=Limpiar(cortes);
    cortes1=Limpiar(cortes1);
                                  //Limpiamos cortes y listas circulares
    aux=circular1;
    while(aux->sig!=circular1)
     aux=aux->sig;
    aux->sig=NULL;
    circular1=Limpiar(circular1);
    aux=circular2;
    while(aux->sig!=circular2) aux=aux->sig;
    aux->sig=NULL;
    circular2=Limpiar(circular2);
   }//else
  }
 return(solu);
}

/*-------------------------------------------------------------------------------------
La siguiente función reactualiza el nodo principal.
-------------------------------------------------------------------------------------*/

lista *Asignar_Nodo_Principal(poligono *p,lista *l)
{poligono *aux;
 aux=p;
 if(aux!=NULL)
 {while((aux->dat.tipo!='P')&&(aux->sig!=NULL))
   aux=aux->sig;
  return(aux->dat.puntos);
 }
}

/*-------------------------------------------------------------------------------------
La siguiente función permite eliminar un nodo principal.
-------------------------------------------------------------------------------------*/

poligono *Eliminar_Nodo_Principal(poligono *p,lista *l)
{poligono *aux1,*aux2;
 if((p!=NULL)&&(l!=NULL))
 {if((p->dat.tipo=='P')&&(listasIguales(p->dat.puntos,l)==true))
  {aux1=p;
   p=p->sig;
   free(aux1);
  }
  else
  {aux2=p;
```

```
   aux1=p->sig;
   while((aux1->dat.tipo!='P')||(listasIguales(aux1->dat.puntos,l)==false))
   {aux2=aux1;
    aux1=aux1->sig;
   }
   if((aux1->dat.tipo=='P')&&(aux1->dat.puntos==l))
   {aux2->sig=aux1->sig;
    free(aux1);
   }
  }
 }
 return(p);
}

poligono *Limpiar2(poligono *polig)
{poligono *aux;
 if(polig!=NULL)
 {aux=polig;
  polig=polig->sig;
  free(aux);
  polig=Limpiar2(polig);
  return(polig);
 }//Limpiar2(aux);
}

/*-------------------------------------------------------------------------------------
Diseñamos  una función que produce el polígono general  resultado de  unir un polígono de
visibilidad con  un polígono general,(conjunto de componentes no conexas y con agujeros).
-------------------------------------------------------------------------------------*/

poligono *Union_Poligono_lista(poligono *solu,poligono *p,lista *l)
{poligono *aux,*soluparcial,*soluparcial1,*aux1;
 lista *copia,*aux3,*aux2;
 nodo final;
 solu=NULL;
 soluparcial=NULL;
 char car;
                                  //Hacemos una copia para unir a los agujeros, pues con los
                                  //agujeros debemos utilizar siempre la lista original
 aux3=l;
 if(aux3!=NULL)
 {copia=(lista*) malloc(sizeof(lista));
  copia->p=aux3->p;
  aux3=aux3->sig;
  aux2=copia;
 }
 while(aux3!=NULL)
 {aux2->sig=(lista*) malloc(sizeof(lista));
  aux2=aux2->sig;
  aux2->p.x=aux3->p.x;
  aux2->p.y=aux3->p.y;
  aux3=aux3->sig;
 }
 aux2->sig=NULL;
                                  //fin copia
 aux=p;
 while(aux!=NULL)
{if(aux->dat.tipo=='A')
 {printf(" A ");
  soluparcial1=NULL;
  soluparcial1=Union_Nodo_Agujero2(soluparcial,aux->dat,copia);
  if(soluparcial1!=NULL)
  {solu=Juntar_Poligonos(solu,soluparcial1);
   soluparcial1=Limpiar2(soluparcial1);
  }
  if(aux->sig==NULL)
  {final.tipo='P';
   final.puntos=l;
   soluparcial=NULL;
   soluparcial=Anadir_Nodo(soluparcial,final);
   solu=Juntar_Poligonos(solu,soluparcial);
   soluparcial=Limpiar2(soluparcial);
  }
 }
 else
 {soluparcial=Union_Nodo_Principal(soluparcial,aux->dat,l,&car);
  if((aux->sig!=NULL)&&(car=='S'))
```

```
  {l=Asignar_Nodo_Principal(soluparcial,l);
   soluparcial=Eliminar_Nodo_Principal(soluparcial,l);
  }
  else
   if((aux->sig!=NULL)&&(car=='N'))
    soluparcial=Eliminar_Nodo_Principal(soluparcial,l);
   else
    if((aux->sig!=NULL)&&(car=='I'))
    {l=Asignar_Nodo_Principal(soluparcial,l);
     soluparcial=Eliminar_Nodo_Principal(soluparcial,l);
    }
   if(soluparcial!=NULL)
   {solu=Juntar_Poligonos(solu,soluparcial);
    soluparcial=Limpiar2(soluparcial);
   }
  }
  aux=aux->sig;
 }
 return(solu);
}

/*-----------------------------------------------------------------------------------------
Parece que ya tenemos la union. Diseñamos una funcion  Construye_Union, tal  que  dado un
polígono normal y una lista de puntos interiores a él, nos devuelve el polígono unión  de
los polígonos de visibilidad de todos los puntos
-----------------------------------------------------------------------------------------*/

poligono *Construye_Union(poligono *solu,lista *poligo, lista* nube,double a,double *apa)
{lista *polvisi,*auxx,*solauxy;int i;
 poligono *soluaux;
 nodo final;
 FILE *polunionagujeros;
 solu=NULL;
 auxx=nube;
 i=0;
 while(auxx!=NULL)
 {i=i+1;
  adelante:
                                  //Calculo el primer polígono de visibilidad
  polvisi=NULL;
  polvisi=(lista*) malloc(sizeof(lista));
  polvisi=calpolvisi(auxx->p,poligo,polvisi);
  if((polvisi==NULL)||(polvisi->p.x==-10)||(longitud(polvisi)==1))
  {printf(''Ya estamos'');
   auxx->p=generapunto(poligo);
   goto adelante;}
   if(polvisi!=NULL)
   {polvisi=desbrozar(polvisi);
    if(auxx==nube)
    {final.tipo='P';
     final.puntos=polvisi;
     solu=Anadir_Nodo(solu,final);
    }
    else
    {soluaux=Union_Poligono_lista(soluaux,solu,polvisi);
     solu=Limpiar2(solu);
     solu=NULL;
     solu=Juntar_Poligonos(solu,soluaux);
     soluaux=Limpiar2(soluaux);
    }
   auxx=auxx->sig;
   }
  }
                                  //Limpiamos la solucion
  soluaux=solu;
  while(soluaux!=NULL)
  {soluaux->dat.puntos=desbrozar(soluaux->dat.puntos);
   soluaux=soluaux->sig;
  }
  *apa=area_poligonoagujeros(solu);
  printf(''\n\n\t\t* Area Poligono Inicial %lf *Area Iluminada %lf\n'',a,*apa);
  polunionagujeros=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1
                          \\Matlab\\polunionagujeros.txt'',''w+'');
  soluaux=solu;
  while(soluaux!=NULL)
  {solauxy=soluaux->dat.puntos;
   while(solauxy!=NULL)
```

```
   {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
    solauxy=solauxy->sig;
   }
   if(soluaux->dat.tipo=='P')
    fprintf(polunionagujeros,''%f %f\n'',2.0,2.0);
   else
    fprintf(polunionagujeros,''%f %f\n'',2.0,1.0);
   soluaux=soluaux->sig;
  }
  fprintf(polunionagujeros,''%f %f\n'',*apa,(*apa*100.0)/a);
  fclose(polunionagujeros);
  return(solu);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función que limpia la solución para eliminar puntos alineados
-------------------------------------------------------------------------------------*/

lista *desbrozar(lista *polig)
{lista *aux,*aux1,*aux2;
 if(longitud(polig)>=3)
 {aux1=polig;
  aux2=polig->sig;
  while(aux2->sig!=NULL)
  {if(pertenece_segm(aux2->p,aux1->p,aux2->sig->p)==true)
   {aux1->sig=aux2->sig;
    aux=aux2;
    aux2=aux2->sig;
    free(aux);
   }
   else
   {aux1=aux2;
    aux2=aux2->sig;
   }
  }
 }
 return(polig);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función que calcular el área de un polígono general
-------------------------------------------------------------------------------------*/

double area_poligonoagujeros(poligono *polig)
{poligono *aux;
 lista *aux1,*aux2;
 double area,areaaux;
 char c;
 area=0;
 aux=polig;
 while(aux!=NULL)
 {c=aux->dat.tipo;
  switch(c)
  {case 'P':areaaux=area_poligono(aux->dat.puntos);
            if (areaaux!=-1.0)
             area+=areaaux;
            else
             {area=0;goto fin;}
            break;
   case 'A':areaaux=area_poligono(aux->dat.puntos);
            if (areaaux!=-1.0)
             area=area-areaaux;
            else
             {area=0;goto fin;}
            break;
  }
  aux=aux->sig;
 }
 fin:
 return(area);
}

/*-------------------------------------------------------------------------------------
Ya podemos implementar la función objetivo fitnessk
-------------------------------------------------------------------------------------*/

double fitnessk(lista *indi,lista *poligo)
{double area;
double areapol;
```

```
 int primeravez=1;
 poligono *polaux;
 areapol=area_poligono(poligo);
 polaux=NULL;
 polaux=Construye_UnionBis(polaux,poligo,indi,areapol,&area);
 polaux=Limpiar2(polaux);
                                  //Limpiamos el poligono solución
 return((area*100.0)/areapol);
}
```

Tenemos ya implementados todos los códigos que permiten encontrar el valor de la *función fitness* o *función objetivo* de nuestro problema. Pasamos a continuación a exponer las implementaciones diseñadas para cada una de las heurísticas que solucionan el problema MaxA-p-Pvk(P, k).

Implementaciones basadas en la heurística simulated annealing-SA

Podemos entender el siguiente programa como una generalización de expuesto anteriormente para *simulated annealing-SA* con el problema MaxA-p-Pv1(P), con esta misma heurística. Como se puede observar la variación fundamental radica el la función objetivo utilizada, pero la estructura de la heurística es la misma.

```
/*-------------------------------------------------------------------------------------
La implementación utilizada e igual a la diseñada para la búsqueda  de1 punto  de  máxima
iluminación.
-------------------------------------------------------------------------------------*/

poligono *MaxRS_PVK(int tipor,double tpi,double dt,int *itera,lista *polig,poligono *polvisi,lista *ps,
                    int k,double  a,double *apa)
{unsigned long int i,ki,t;
 long hora;
 int horasistema;
 punto p1,p2; lista *psaux,*psauy,*aux,*aux1,*solauxy;
 poligono *soluaux;
 double apsaux,apsauy,delta,tk,U,U1,U2,div;
 FILE *polunionagujeros,*rastro;
rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''w+'');
                                  //Generamos una primera solución

 hora=time(NULL);
                                  //Generamos una semilla de sistema
 horasistema=(unsigned int) hora/2;
 srand(horasistema);
 psaux=NULL;
 for(t=0;t<k;++t)
 {p1.x=(1.0*rand())/(RAND_MAX);
  p1.y=(1.0*rand())/(RAND_MAX);
  while((fmod(cortes(p1,polig),2.0)==0.0)||(Es_Ordenada_vertice(p1,polig)==true))
  {p1.x=(1.0*rand())/(RAND_MAX);
   p1.y=(1.0*rand())/(RAND_MAX);
  }
  psaux=Anadir(psaux,p1);
}
 apsaux=fitnessk(psaux,polig);
 ki=1;
 tk=tpi;
 div=10.0;
 do
 {i=1;
  do
  {aux=psaux;
                                  //Genero una nueva solución moviendo un
                                 //poco los puntos anteriores
   psauy=NULL;
   while(aux!=NULL)
```

```
  {do
   {U1=(1.0*rand())/(RAND_MAX);
    U2=(1.0*rand())/(RAND_MAX);
    p2.x=aux->p.x+((sqrt(-(2.0*log(U1)))*sin(2.0*pi*U2))/div);
    p2.y=aux->p.y+((sqrt(-(2.0*log(U1)))*cos(2.0*pi*U2))/div);
   }while(punto_int(p2,polig)==false);
  psauy=Anadir(psauy,p2);
   aux=aux->sig;
  }
  *itera=*itera+1; //Miramos el área iluminada por la nueva solución
  apsauy=fitnessk(psauy,polig);
                                  //Calculamos delta
  delta=apsauy-apsaux;
                                  //Estudiamos el cambio
  if(delta>0)
  {psaux=Limpiar(psaux);
   psaux=psauy;
   apsaux=apsauy;
   aux1=psaux;
   while(aux1!=NULL)
   {fprintf(rastro,''%f %f\n'',aux1->p.x,aux1->p.y);
    aux1=aux1->sig;
   }
  }
                                  //Si no depende de la temperatura
  else
  {U=(1.0*rand())/(RAND_MAX);
   if(U<exp((delta)/(tk)))
   {psaux=Limpiar(psaux);
    psaux=psauy;
    apsaux=apsauy;
    aux1=psaux;
    while(aux1!=NULL)
    {fprintf(rastro,''%f %f\n'',aux1->p.x,aux1->p.y);
     aux1=aux1->sig;
    }
   }
  }
  ++i;
 }while ((i)<=(tk));
 switch(tipor)
{case 0:tk=dt*tk;break;
 case 1:tk=(tpi)/log(1+ki);break;
 case 2:tk=(tpi)/(1+ki); break;
 case 3:tk=(tpi)/exp(ki); break;
 case 4:tk=(tpi)/(exp(exp(ki)));break;
 case 5:tk=(0.9)*tk;break;
}
div=div/(0.99);
ki=ki+1;
printf(''\n\t%lf..\t'',tk);
}while(tk>0.0005);
ps=psaux;
                                  //Actualizamos la salida
                                  //Guardamos en un fichero par ver en Matlab
polvisi=NULL;
polvisi=Construye_Union(polvisi,po lig,ps,a,&*apa);
polvisi=LimpiarSolucionRS(polvisi,polig,ps,a,&*apa);

                                  //Ahora lo guardamos en el fichero.

polunionagujeros=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1
                       \\Matlab\\polvisirsk.txt'',''w+'');
soluaux=polvisi;
while(soluaux!=NULL)
{solauxy=soluaux->dat.puntos;
 while(solauxy!=NULL)
 {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
  solauxy=solauxy->sig;
 }
 if(soluaux->dat.tipo=='P')
  fprintf(polunionagujeros,''%f %f\n'',2.0,2.0);
 else
  fprintf(polunionagujeros,''%f %f\n'',2.0,1.0);
soluaux=soluaux->sig;
}
                                  //Pintamos los puntos
```

```
 solauxy=ps;
 while(solauxy!=NULL)
 {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
  solauxy=solauxy->sig;
 }
 fprintf(polunionagujeros,''%f %f\n'',2.0,3.0);
 fprintf(polunionagujeros,''%f %f\n'',*apa,(*apa*100.0)/a);
 fclose(polunionagujeros);
 fclose(rastro);
return(polvisi);

}
```

Implementaciones con algoritmos genéticos-AG

Incluimos los códigos para el algoritmo genético descrito en la Sección 8.4. Se exponen también las funciones para la ejecución de los operadores de cruce y mutación utilizados.

```
/*-----------------------------------------------------------------------------------
Parece que ya tenemos la union. Diseñamos una funcion Construye_Union, tal  que  dado  un
Implementamos ahora un algoritmo genético para calcular los k puntos de  máxima  ilumina-
ción. De forma resumida los elemenos que interviene son los siguientes:
  Individuo:Una lista con k puntos.
  Población:Un vector con kn individuos; n vértices de P.
  Selección:Proporcional por ruleta.
  Cruce:En un punto.
  Mutación:Sumado a cada coordenada N(0,1)/100 con probabilidad pm.
  Generación inicial: Aleatoria. Se genera dentro de la función.
  Evaluar población : fitnessk del mejor individuo.
  Paso de generación: Sustituir el hijo por el peor individuo de la población.
  Condición de parada: Cuando en 100 generaciones no cambie el fitness.
La función devolverá el polígono unión.
-----------------------------------------------------------------------------------*/

poligono *MaxGNT_PVK(int *itera,lista *polig,poligono *polvisi,lista *ps,int k,double a,double *apa)
{FILE *polunionagujeros,*rastro;
 poligono *soluaux,*polvisi2,*aux;
 punto p1;
 lista *poblacion[Tampobla],*aux1;
                                  //Guarda la población en cada generación

 lista *padre1,*padre2,*hijo,*solauxy;
 double pc=0.8;
                                  //Probabilidad de cruce
 double pm=0.05;
                                  //Probabilidad de mutación
 double P;
 double objaux,obj=0;
                                  //Valor del mejor individuo de cada población
 int i,m,pd1,pd2;
 int t,j,hora,horasistema;
                                  //Contador y generación aleatoria
                                  //Generamos la población inicial de kn individuos

 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\datoscurva.txt'',''w+'');
 hora=time(NULL);
 horasistema=(unsigned int) hora/longitud(polig);
 srand(horasistema);
 j=0;
                                  //Contador número de individuos
 aux1=polig;
 while(j!=k*longitud(polig))
 {poblacion[j]=NULL;
                                  //Creamos los puntos y los guardamos en población
  poblacion[j]=NULL;
  for(t=0;t<k;++t)
  {p1.x=(1.0*rand())/(RAND_MAX);
   p1.y=(1.0*rand())/(RAND_MAX);
   while((fmod(cortes(p1,polig),2.0)==0.0)||(Es_Ordenada_vertice(p1,polig)==true))
   {p1.x=(1.0*rand())/(RAND_MAX);
    p1.y=(1.0*rand())/(RAND_MAX);
   }
```

```
   poblacion[j]=Anadir(poblacion[j],p1);
  }
  ++j;
  aux1=aux1->sig;
  }
 printf(''\n\n\t\t 1.- Poblacion Inicial OK'');
                                  //Generamos el fitness del mejor individuo de la
                                  //población inicial

 obj=evaluarpoblacionk(poblacion,polig);
 fprintf(rastro,''%f\n'',obj);
 i=0;
 printf(''\n\n\t\t 2.- Evaluacion Poblacion Inicial OK obj=%lf'',obj);

                                  //Entramos en el bucle de iteración
 while(i<=20)
 {*itera=*itera+k*longitud(polig);
                                  //Seleccionamos dos individuos

  printf(''\n\n\t\t 3.- Iteracion: valor de i=%d'',i);
  Ruletak(poblacion,&pd1,&pd2,polig);

                                  //Generamos un n [0,1] para comparar con pc
  P=(1.0*rand())/(RAND_MAX);
  if(P<=pc)
   Crucek(poblacion,pd1,pd2,polig);

                                  //Cruzamos los hijos
                                  //Mutamos

  Mutak(poblacion,pd1,pd2,polig,pm);

                                  //Pasamos a la nueva generacion en Mutak
                                  //Evaluamos población

  objaux=evaluarpoblacionk(poblacion,polig);
  fprintf(rastro,''%f\n'',objaux);
  printf(''\nFitness ultima poblacion %lf - %d --%lf'',objaux,i,obj);
  if(objaux>obj)
   {i=0;
    obj=objaux;
   }
   else
    ++i;
 }//while
 fclose(rastro);
 printf(''\n\n\t\t 4.- Iteracion: Fin OK'');

                                  //Tomamos el mejor individuo de la población final

obj=0;j=0;
 for(t=0;t<k*longitud(polig);++t)
 {objaux=fitnessk(poblacion[t],polig);
  if(objaux>obj)
  {obj=objaux;
   j=t;
  }
 }
 ps=poblacion[j];
                                  //Construimos unión
 polvisi=NULL;
 polvisi=Construye_Union(polvisi,polig,ps,a,&*apa);

                                  //Ahora lo guardamos en el fichero

polunionagujeros=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1
                        \\Matlab\\polvisigntk.txt'',''w+'');
 soluaux=polvisi;
 while(soluaux!=NULL)
 {solauxy=soluaux->dat.puntos;
  while(solauxy!=NULL)
  {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
   solauxy=solauxy->sig;
  }
  if(soluaux->dat.tipo=='P')
   fprintf(polunionagujeros,''%f %f\n'',2.0,2.0);
  else
   fprintf(polunionagujeros,''%f %f\n'',2.0,1.0);
```

```
  soluaux=soluaux->sig;
 }
                                    //Pintamos los puntos
 solauxy=ps;
 while(solauxy!=NULL)
 {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
  solauxy=solauxy->sig;
 }
 fprintf(polunionagujeros,''%f %f\n'',2.0,3.0);
 fprintf(polunionagujeros,''%f %f\n'',*apa,(*apa*100.0)/a);
 fclose(polunionagujeros);
 fclose(rastro);
 return(polvisi);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función que nos da la EVALUACIÓN DE LA  POBLACIÓN es decir, el fitnessk del
mejor individuo de la población.Esta es el fitness de la  población o la evaluación de la
población.
-------------------------------------------------------------------------------------*/

double evaluarpoblacionk (lista *poblacion[],lista *poligo)
{int i=0;
 double obj=0,objaux;
 obj=fitnessk(poblacion[i],poligo);
 for(i=1;i<longitud(poblacion[0]*longitud(poligo);++i)
 {objaux=fitnessk(poblacion[i],poligo);
 if(objaux>obj)
  obj=objaux;
 }
 return(obj);

}
```

Las implementaciones de los operadores de selección, cruce y mutación son las siguientes:

1. **Operador de Selección**

```
/*-------------------------------------------------------------------------------------
Parece que ya tenemos la union. Diseñamos una funcion Construye_Union, tal  que  dado  un
La función Ruletak nos devolverá como dos  parámetros  de  salida  las  celdillas  de  la
población i y j donde se encuentran los  padres seleccionados.La función CrucekBis susti-
tuirá el individuo i y j por los hijos  obtenidos con cruce en 1 punto.Finalmente la fun-
ción Mutak mutará los nuevos hijos que se encontrarán ya en las posiciones i y j.
-------------------------------------------------------------------------------------*/

void Ruletak(lista *poblacion[],int *i, int *j,lista *poligo)
{double vectorfitness[Tampobla],suma,sumaux,sumapar;
 int t,a;
 suma=0.0;
 for(t=0;t<longitud(poblacion[1]*longitud(poligo);++t)
 {vectorfitness[t]=fitnessk(poblacion[t],poligo);
  suma=suma+vectorfitness[t];
 }
 do
 {a=RAND_MAX;
                                    //Hacemos girar la ruleta para le 1ºpadre
  while(a==RAND_MAX) a=rand();
   sumapar=suma*((1.0*a)/RAND_MAX);
  t=0;
  sumaux=vectorfitness[t];
  while(sumaux<sumapar)
  {t=t+1;
   sumaux=sumaux+vectorfitness[t];
  }
  *i=t;
                                    //Ahora ruleta para el segunda padre
  a=RAND_MAX;
  while(a==RAND_MAX) a=rand();
   sumapar=suma*((1.0*a)/RAND_MAX);
  t=0;
  sumaux=vectorfitness[t];
  while(sumaux<sumapar)
  {t=t+1;
   sumaux=sumaux+vectorfitness[t];
```

```
  }
  *j=t;
  printf(‘‘ Selección %d %d’’,*i,*j);
 }while(*i==*j);

}
```

2. **Operador de Cruce**

```
void Crucek(lista *poblacion[],int i, int j,lista *poligo)
{lista *hijo1,*hijo2,*aux1,*aux2;
 punto p;
 int U,k;
 double t;
 hijo1=NULL;
 hijo2=NULL;
                                    //Genero un nºentero aleatorio
 t=longitud(poblacion[i]);
 U=1.0+((double(t)*rand())/(RAND_MAX));
 aux1=poblacion[i];
 aux2=poblacion[j];
                                    //Mezclo en un punto
 for(k=1;k<=U;++k)
 {hijo1=Anadir(hijo1,aux1->p);
  hijo2=Anadir(hijo2,aux2->p);
  aux1=aux1->sig;
  aux2=aux2->sig;
 }
 for(k=U+1;k<=t;++k)
 {hijo1=Anadir(hijo1,aux2->p);
  hijo2=Anadir(hijo2,aux1->p);
  aux1=aux1->sig;
  aux2=aux2->sig;
 }
                                    //Mutamos y pasamos a la generación siguiente

 hijo1=Mutak(hijo1,poligo,0.2);printf(‘‘Hay mutacion’’);
 hijo2=Mutak(hijo2,poligo,0.2);printf(‘‘Hay mutacion’’);
 Pasok(hijo1,poblacion,poligo);
 Pasok(hijo2,poblacion,poligo);
 poblacion[i]=Limpiar(poblacion[i]);
 poblacion[j]=Limpiar(poblacion[j]);
 poblacion[i]=hijo1;
 poblacion[j]=hijo2;*/

}
```

3. **Operador de Mutación:** Incluimos la función de mutación y de paso a la generación siguiente.

```
/*-------------------------------------------------------------------------------------
Diseñamos ahora el operador de  MUTACIÓN.Le sumamos una  N(0,1)/10  a  cada componente de
cada punto.
-------------------------------------------------------------------------------------*/

lista *Mutak(lista *indi,lista *polig,double pm)
{double U1,U2,N1,N2;
 int i,U,t,hora,horasistema;
 punto p;
 lista *aux;
 i=longitud(indi);
                                    //Elegimos un gen. Es mutación por sustitución

 U=1.0+((double(i)*rand())/(RAND_MAX));
                                    //Mutamos cada individuo de la población

 aux=indi;
 while(aux!=NULL)
 {U=((1.0*rand())/(RAND_MAX));
  do
  {U1=(1.0*rand())/(RAND_MAX);
  U2=(1.0*rand())/(RAND_MAX);
```

```
                                    //Genero dos normales
   N1=((sqrt((-2.0*log(1-U1)))*sin(U2)));
   N2=((sqrt((-2.0*log(1-U1)))*cos(U2)));
   }while((N1>1.0)||(N2>1.0));
                                    //Lo sumo a las coordenadas a cada punto
  N1=N1/100.0;
  N2=N2/100.0;
  p.x=aux->p.x+N1;
  p.y=aux->p.y+N2;
  if((fmod(cortes(p,polig),2.0)!=0.0)&&(Es_Ordenada_vertice(p,polig)==false))
  {printf(''\n Antes (%f,%f)'',aux->p.x,aux->p.y);
   aux->p.x=aux->p.x+N1;
   aux->p.y=aux->p.y+N2;
  }
 }
  aux=aux->sig;
 }
 return(indi);
}

 /*-------------------------------------------------------------------------------------
La siguiente función transforma la población a la generación siguiente
-------------------------------------------------------------------------------------*/

void Pasok(lista *indi,lista *poblacion[],lista *poligo)
{double obj=100,objaux;
 int i,indice=0;
 lista *aux;
 FILE *rastro;
 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''a'');
 for(i=0;i<longitud(poblacion[1])*longitud(poligo);++i)
                                    //Buscamos el peor individuo de la población

 {objaux=fitnessk(poblacion[i],poligo);
  aux=poblacion[i];
  while(aux!=NULL)
  {fprintf(rastro,''%f %f\n'',aux->p.x,aux->p.y);
   aux=aux->sig;
  }
 if(objaux<obj)
  {obj=objaux;
   indice=i;
  }
 }
                                    //Ahora lo sustituimos

 poblacion[indice]=Limpiar(poblacion[indice]);
 poblacion[indice]=indi;
 fclose(rastro);

}
```

Implementaciones para la heurística random search-RS

Análogamente incluimos los códigos para la búsqueda aleatoria o *random search-RS*.

```
/*-------------------------------------------------------------------------------------
Implementamos como primera aproximación  una función que  busca de manera  aleatoria  una
lista con k puntos de máxima iluminación. La idea básica consiste en ir  generando muchas
listas y quedarnos con la mejor.
-------------------------------------------------------------------------------------*/

poligono *MaxRS_PVK(lista *nube,lista *polig,poligono *polvisi,lista *ps,int k,double a,double *apa)
{lista *aux,*aux1,*aux2,*psaux,*solauxy;
 double area=0;
 poligono *polaux,*soluaux;
 int i,j;
 FILE *polunionagujeros;
 aux=nube;
 *apa=0;
 j=0;
 while(aux!=NULL)
 {++j;
  printf('' %d k-elemento\n'',j);
```

```
                                    //Genero un conjunto con k elementos de la nube
 i=1;
 psaux=NULL;
 while(i<=k)
 {psaux=Anadir(psaux,aux->p);
  aux1=aux;
  aux=aux->sig;
  free(aux1);
  ++i;
 }
                                    //Construyo su unión
 polaux=NULL;
 polaux=Construye_Union(polaux,polig,psaux,a,&area);
 if(area>*apa)
  {polvisi=Limpiar2(polvisi);
   polvisi=polaux;
   aux2=ps->sig;
   aux2=Limpiar(aux2);
   ps->p=psaux->p;
   ps->sig=NULL;
   aux1=psaux->sig;
   while(aux1!=NULL)
   {ps=Anadir(ps,aux1->p);
    aux1=aux1->sig;
   }
   psaux=Limpiar(psaux);
   *apa=area;
  }
  else
  {psaux=Limpiar(psaux);
   polaux=Limpiar2(polaux);
  }
 }
                                    //Guardamos la solución
printf(''\n\n\t\t\t* Area Poligono Inicial %lf *Area Iluminada final %lf\n'',a,*apa);
polunionagujeros=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\polvisidetk.txt'',''w+'');
soluaux=polvisi;
while(soluaux!=NULL)
{solauxy=soluaux->dat.puntos;
 while(solauxy!=NULL)
 {fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
  solauxy=solauxy->sig;
 }
 if(soluaux->dat.tipo=='P')
  fprintf(polunionagujeros,''%f %f\n'',2.0,2.0);
 else
  fprintf(polunionagujeros,''%f %f\n'',2.0,1.0);
 soluaux=soluaux->sig;
}
                                    //Pintamos los puntos
solauxy=ps;
while(solauxy!=NULL)
{fprintf(polunionagujeros,''%f %f\n'',solauxy->p.x,solauxy->p.y);
 solauxy=solauxy->sig;
}
fprintf(polunionagujeros,''%f %f\n'',2.0,3.0);
fprintf(polunionagujeros,''%f %f\n'',*apa,(*apa*100.0)/a);
fclose(polunionagujeros);
return(polvisi);
}
```

B.6 Códigos relacionados con el problema MinN-p-PvK(P)

Una vez implementadas las funciones necesarias para calcular los k puntos de máxima iluminación, (utilizando cada una de las heurísticas), se pueden utilizar para solucionar el problema MinN-p-PvK(P) realizando una búsqueda secuencial o binaria, del número mínimo k de luces necesarias para iluminar el polígono P o un porcentaje elevado de su área, como por ejemplo el 99%. Exponemos a continuación el código de cada una de estas estrategias, analizadas

en el Capítulo 8. Se muestran sólamente el código para ambas estrategias utilizando *simulated annealing-SA*, para solucionar cada una de las llamadas al problema MaxA-p-PvK(P, k). Para los otros dos métodos heurísticos, (GA y RS), se debe cambiar la llamada a la heurística correspondiente.

Estrategia secuencial

```
void Min_P_PVK_SA_Secuencial(lista *polig,double a)
{lista *mispuntos,*minube,*solauxy;
 poligono *mipoligonounion,*soluaux;
 double miarea,porcentaje;
 int mik,itera=0;
 FILE *polunionagujeros;
 mik=1;
 miarea=0;
 porcentaje=0;
                    //Empezamos con 1 luz y aumentamos hasta iluminar 99%
 while(porcentaje<99.0)
 {mipoligonounion=NULL;
  mispuntos=(lista*) malloc(sizeof(lista));
  mispuntos->sig=NULL;
  mipoligonounion=MaxRs_PVK(0,4.0*longitud(polig),0.95,&itera,polig,mipoligonounion,mispuntos,mik,a,&miarea);
  porcentaje=(miarea*100.0)/a;
  if(porcentaje<99.0)
  {mipoligonounion=Limpiar2(mipoligonounion);
   mispuntos=Limpiar(mispuntos);
 }
 printf(‘‘ \n\t\t\t Numero de puntos %d.- PORCENTAJE %.30f’’,mik,porcentaje);//getche();
 ++mik;
 }
}
```

Estrategia binaria

Implementamos ahora el mismo mecanismo de búsqueda,pero no secuencial. Empezaremos en el intervalo [1,n/3] y vamos mirando el punto medio y dividiendo el intervalo como en el método de la bisección, hasta tener un intervalo de longitud 1. En este caso daremos el extremo superior del intervalo,ya que si damos el inferior corremos el peligro de estar en un intervalo en el que el inferior no llega y el superior sí.

```
void Min_P_PVK_SA_Binaria(lista *polig,double a)
{lista *mispuntos,*minube,*solauxy;
 poligono *mipoligonounion,*soluaux;
 double miarea,porcentaje;
 int mia,mib,itera=0;
 FILE *polunionagujeros;
 mia=1;
 mib=longitud(polig)/3;
 miarea=a;
 porcentaje=100.0;
 while((mib-mia)!=1)
 {mipoligonounion=NULL;
  mispuntos=(lista*) malloc(sizeof(lista));
  mispuntos->sig=NULL;
  mipoligonounion=MaxRs_PVK(0,4.0*longitud(polig),0.90,&itera,polig,mipoligonounion,mispuntos,
                            (mia+mib)/2,a,&miarea);
  porcentaje=(miarea*100.0)/a;
  printf(‘‘\n\t\t\t Numero de puntos [%d-%d] ->%d.- PORCENTAJE %.30f’’,mia,mib,(mia+mib)/2,porcentaje);
  if(porcentaje<99.0)
   mia=(mia+mib)/2;
  else
   mib=(mia+mib)/2;
  mipoligonounion=Limpiar2(mipoligonounion);
```

```
  mispuntos=Limpiar(mispuntos);
 }
 mipoligonounion=NULL;
 mispuntos=(lista*) malloc(sizeof(lista));
 mispuntos->sig=NULL;
 mipoligonounion=MaxRs_PVK(0,longitud(polig),0.95,&itera,polig,mipoligonounion,mispuntos,mib,a,&miarea);
 porcentaje=(miarea*100.0)/a;
 printf('' \n\t\t\t Numero de puntos [%d,%d].- PORCENTAJE %.30f'',mia,mib,porcentaje);
 mipoligonounion=Limpiar2(mipoligonounion);
 mispuntos=Limpiar(mispuntos);
}
```

B.7 Implementaciones para el problema MaxA-p-Vor(N)

Incluimos en esta sección las implementaciones realizadas para el estudio heurístico del problema MaxA-p-Vor(C). Dividimos estos programas en tres apartados:

1. **Tipos da datos:** Incluimos los tipos de datos adicionales a los expuestos de forma general en la Sección B.2. Se determina la estructura necesaria para representar un *diagrama de Voronoi*, que será una lista enlazada de nodos que incluirán un punto, su *región de Voronoi* asociada y una lista de vecindades que permitirá optimizar la construcción.

2. **Construcción del diagrama de Voronoi:** Presentamos el código implementado para la construcción del *diagrama de Voronoi* de una nube de puntos. El algoritmo elegido ha sido un algoritmo incremental, ya que para solucionar el problema MaxA-p-Vor(N) nos interesará obtener la *región de Voronoi* $Rv(p, N \cup \{p\}, R)$ de un nuevo punto p, que se pretende añadir a la estructura, tal que su área sea máxima.

3. **Métodos aproximados para el problema** MaxA-p-Vor(N)**:** Exponemos las implementaciones realizadas para solucionar heurísticamente este problema mediante algoritmos basadas en *simulated annealing*, (SA), y *random search*, (RS).

Tipos de datos

Un *diagrama de Voronoi* será una lista enlazada de nodos. Cada nodo incluye las coordenadas de un punto p, la *región de Voronoi* asociada a este punto $Rv(p, N \cup \{p\}, R)$ y una lista de punteros a nodos, que contiene a todos los vecinos de p. Es importante que la estructura contenga información sobre las vecindades de un punto, pues esto permitirá optimizar la construcción del *diagrama de Voronoi*.

```
/*-------------------------------------------------
Definimos la estructura de datos voronoi
-------------------------------------------------*/

struct lista_elementos4
{punto p;                          //punto
 lista *zona;                      //región de voronoi asociada a p
 struct vecinos
 {struct lista_elementos4 *inf;    //Lista de punteros a vecinos
  struct vecinos *sig;
 }*vec;
struct lista_elementos4 *sig;      //siguiente nodo de voronoi
};
```

```
typedef struct lista_elementos4 voronoi;
```

Los siguientes códigos contienen los programas necesarios para construir el *diagrama de Voronoi* de un conjunto de puntos situados en el cuadrado unidad es decir el cuadrado de vértices $(0,0)$, $(1,0)$, $(1,1)$ y $(0,1)$.

Construcción del diagrama de Voronoi

Presentamos las implementaciones que permiten construir de forma incremental el *diagrama de Voronoi* asociado a un conjunto de puntos. Incluimos también de forma explicativa el conjunto de funciones necesarias para llegar al programa final `CalculaVoronoi`. En forma de pseudocódigo el algoritmo se describe en X.

```
/*-------------------------------------------------------------------------------------
Definimos una función que nos permitirá leer los datos introducidos por el usuario usando
la función generaconjunto de MatLab
-------------------------------------------------------------------------------------*/

void Lectura_CONJUNTO(lista *conjunto)
{FILE *conj;
 lista *aux;
 punto punt;
                                                    //Abrimos el fichero
conj=fopen(''c:\\PVGC\\Pruebas_Heur_P1\\Matlab\\conjunto.txt'',''r'');
fscanf(conj,''%lf %lf\n'',&punt.x,&punt.y);
conjunto->p.x=punt.x;                               //se guarda en la lista
conjunto->p.y=punt.y;
aux=conjunto;

while(feof(conj)==0)
{fscanf(conj,''%lf %lf\n'',&punt.x,&punt.y);
aux->sig=(lista*)malloc(sizeof(lista));
 aux=aux->sig; //Vamos generando la lista
 aux->p.x=punt.x;
 aux->p.y=punt.y;
}
                                                   //Cerramos la lista
fclose(conj);
aux->sig=NULL;
}

/*-------------------------------------------------------------------------------------
Vamos a diseñar una función que nos  diga si un punto ve los dos extremos de un  segmento
Diseñamos una primera función que mira si dos  zonas  son  vecinas, mirando  si comparten
algún lado de su borde.
-------------------------------------------------------------------------------------*/

bool vecinas(lista *zona1,lista *zona2)
{lista *aux1,*aux2;
aux1=zona1;
if((zona1==NULL)||(zona2==NULL))
  return(true);
 else
 {while(aux1->sig!=NULL)
  {aux2=zona2;
   while(aux2->sig!=NULL)
   {if((Iguales(aux1->p,aux2->sig->p)==true)&&(Iguales(aux1->sig->p,aux2->p)==true))
     return(true);
    aux2=aux2->sig;
   }                              //Ahora miramos el último lado de zona2

   if((Iguales(aux1->p,zona2->p)==true)&&(Iguales(aux1->sig->p,aux2->p)==true))
    return(true);
   aux1=aux1->sig;
  }                              //Ahora miramos el último lado de zona1
  aux2=zona2;
  while(aux2->sig!=NULL)
  {if((Iguales(aux1->p,aux2->sig->p)==true)&&(Iguales(zona1->p,aux2->p)==true))
    return(true);
```

```
   aux2=aux2->sig;
  }                              //Ahora miramos el último lado de zona2

   if((Iguales(aux1->p,zona2->p)==true)&&(Iguales(zona1->p,aux2->p)==true))
    return(true);
   return(false);
 }
}

/*-------------------------------------------------------------------------------------
Definimos ahora una función tal que dada una zona de voronoi, un punto y  los extremos de
un segmento con borde en la región de voronoi  transforma  la  región  de  voronoi en una
nueva, que es aquella de las dos en que el  segmeno divide a la zona y en la que se  está
el punto; ps1 y ps2 son dos punteros que  apuntan al primer extremo del segmento donde se
encuentran nuestro punto de intersección.
-------------------------------------------------------------------------------------*/

lista *nuevazona(lista *zona, punto p, punto s1, punto s2,lista *ps1,lista *ps2)
{lista *zonaux,*aux1,*aux2;
 punto p1,p2;
 char c;
                                //Hacemos circular zona
 aux1=zona;
 while(aux1->sig!=NULL)
  aux1=aux1->sig;
 aux1->sig=zona;
                                //Vamos a crear la primera zona
                                //Tomo un punto cualquiera s1 o s2 y lo guardo en zonaux
 zonaux=NULL;
 zonaux=(lista*)malloc(sizeof(lista));
 zonaux->p=s1;
 zonaux->sig=NULL;
 aux1=zonaux;
 aux2=ps1->sig;
                                //Ya está localizado, ahora seguimos hasta encontrar
                                //el otro extremo s2 y lo vamos guardando
 while(aux2!=ps2)
 {aux1->sig=(lista*)malloc(sizeof(lista));
  aux1=aux1->sig;
  aux1->p=aux2->p;
  aux1->sig=NULL;
  aux2=aux2->sig;
 }                              //Ahora guardamos el último aux2
 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p=aux2->p;
 aux1->sig=NULL;
                                //Ahora guardamos s2
 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p=s2;
 aux1->sig=NULL;

/*-------------------------------------------------
Ya está construida la primera zona. Si el  punto p es
interior a esta zona ya hemos terminado, en caso con-
trario debemos construir la otra zona
-------------------------------------------------*/

 if(fmod (cortes(p,zonaux),2.0)!=0.0)
 {aux1=zona;
  while(aux1->sig!=zona)
   aux1=aux1->sig;
  aux1->sig=NULL;
  zona=Limpiar(zona);
  return(zonaux);
 }
 else
  zonaux=Limpiar(zonaux);

/*-------------------------------------------------
Quito el punto del polígono mirando los elementos del
Si llegamos hasta aquí es que la primera zona  no  ha
valido y  debemos  calcular la  segunda. Repetimos el
mismo proceso, pero cambiar s1 y s2
-------------------------------------------------*/

 zonaux=NULL;
 zonaux=(lista*)malloc(sizeof(lista));
 zonaux->p=s2;
```

```
 zonaux->sig=NULL;
 aux1=zonaux;
 aux2=ps2->sig;
 while(aux2!=ps1)
 {aux1->sig=(lista*)malloc(sizeof(lista));
  aux1=aux1->sig;
  aux1->p=aux2->p;
  aux1->sig=NULL;
  aux2=aux2->sig;
 }
                                  //Ahora guardamos el último aux2
 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p=aux2->p;
 aux1->sig=NULL;
                                  //guardamos s1
 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p=s1;
 aux1->sig=NULL;
 aux1=zona;
 while(aux1->sig!=zona)
  aux1=aux1->sig;
 aux1->sig=NULL;
 zona=Limpiar(zona);
 return(zonaux);
}/*fin nuevazona*/

/*-------------------------------------------------------------------------------------
Veamos ahora una función tal que dado un punto y su zona de voronoi asociada y otro punto
devuelve como solución dos puntos s1 y s2 que son los puntos de intersección de la región
de voronoi asociada al primer punto con la mediatriz del  segmento que une los dos puntos
iniciales. Además en aux1 nos devuelve un puntero al primer estremo del segmento que con-
tiene dicho punto, e igual con le segundo.
-------------------------------------------------------------------------------------*/

void puntosinterseccion(punto p1,lista *zonap1,punto p2,punto *s1,punto *s2,lista **ps1,lista **ps2,int *i)
{punto m1,m2,p;
 lista *aux;
 double x,y;
 char c;                          //Determinamos dos puntos m1 y m2 que pertenecen a la mediatriz de p1 y p2
 m1.x=(p1.x+p2.x)/2;
 m1.y=(p1.y+p2.y)/2;
 m2.x=((p1.x+p2.x)/2)+p1.y-p2.y;
 m2.y=((p1.y+p2.y)/2)+p2.x-p1.x;
 *ps1=NULL;
 *ps2=NULL;
 aux=zonap1;
 *i=0;
 while((aux->sig!=NULL)&&(*i!=2))
 {if(cortanrectasbis(m1,m2,aux->p,aux->sig->p,&x,&y)==true)
  {if(*i==0)
   {(*s1).x=x;
    (*s1).y=y;
    *ps1=aux;
    *i=*i+1;                      //Miramos lado a lado
   }
   else
   {(*s2).x=x;
    (*s2).y=y;
    *ps2=aux;
    *i=*i+1;
   }
  }
  aux=aux->sig;
 }                                //Ahora miramos el último lado
 if((cortanrectasbis(m1,m2,aux->p,zonap1->p,&x,&y)==true)&&(*i!=2))
 {(*s2).x=x;
  (*s2).y=y;
  *ps2=aux;
  *i=*i+1;
 }
}

/*-------------------------------------------------------------------------------------
La siguiente función elimina de la lista de vecinas de un nodo de voronoi la referencia a
una vecina vec
-------------------------------------------------------------------------------------*/
```

```
voronoi *eliminarvecina(voronoi *nodo, voronoi *veci)
{voronoi::vecinos *aux5,*aux6;
 aux5=nodo->vec;                 //Buscamos el nodo a eliminar
 if(aux5!=NULL)
 {if(nodo->vec->inf==veci)
  {aux5=nodo->vec;
   nodo->vec=nodo->vec->sig;
   aux5->sig=NULL;
   free(aux5);
  }
  else
  {aux6=aux5;
   aux5=aux5->sig;
   while(aux5->inf!=veci)
   {aux6=aux5;
    aux5=aux5->sig;
   }                             //Ahora lo eliminamos
   aux6->sig=aux5->sig;
   aux5->sig=NULL;
   free(aux5);
  }
 }
 return(nodo);
}

/*-------------------------------------------------------------------------------------
La siguiente funcion hace lo contrario que la anterior. Añade un vecino  a  la  lista  de
vecinos de un nodo.
-------------------------------------------------------------------------------------*/

voronoi *sumavecino(voronoi *vor, voronoi *veci)
{voronoi::vecinos *aux;
                                 //Añadimos a cada uno como vecino del otro
 if(vor->vec==NULL)
 {vor->vec=(voronoi::vecinos*)malloc(sizeof(voronoi::vecinos));
  vor->vec->inf=veci;
  vor->vec->sig=NULL;
 }
 else
 {aux=vor->vec;
  while(aux->sig!=NULL)
   aux=aux->sig;
  aux->sig=(voronoi::vecinos*)malloc(sizeof(voronoi::vecinos));
  aux=aux->sig;
  aux->inf=veci;
  aux->sig=NULL;
 }
 return(vor);
}

/*-------------------------------------------------------------------------------------
Diseñamos ahora una función tal que dada un voronoi y un nodo de él actualiza su lista de
vecindades.
-------------------------------------------------------------------------------------*/

voronoi *actualizavecindades(voronoi *vor)
{voronoi *aux,*aux2;
 voronoi::vecinos *aux1,*aux3;
                                 //Buscamos el nodo donde se encuentra el punto
 aux=vor;
 aux1=vor->vec;
 while(aux1!=NULL)
 {if(vecinas(aux->zona,aux1->inf->zona)==false)
  {aux3=aux1->sig;
   aux2=aux1->inf;
   aux1->inf=eliminarvecina(aux1->inf,aux);
   aux1=aux1->sig;
   aux=eliminarvecina(aux,aux2);
  }
  else
   aux1=aux1->sig;
 }
 return(vor);
}

/*-------------------------------------------------------------------------------------
Diseñamos una función que nos da la longitud del diagrama de  voronoi. Esto es  necesario
pues si el diagrama de voronoi es un solo punto no se precisa hacer nada
-------------------------------------------------------------------------------------*/

int longvoronoi(voronoi *vor)
```

```
{voronoi *aux;
 int i;
 aux=vor;
 i=0;
 while(aux!=NULL)
 {aux=aux->sig;
  ++i;
 }
 return(i);
}

/*-----------------------------------------------------------------------------------
Diseñamos ahora dos funciones para limpiar memoria. Limpiar3  limpia una lista de vecinos
y Limpiar4 limpia una lista de voronoi
-----------------------------------------------------------------------------------*/

voronoi::vecinos *Limpiar3(voronoi::vecinos *vec)
{voronoi::vecinos *aux;
 if(vec!=NULL)
 {aux=vec;
  vec=vec->sig;
  free(aux);
  vec=Limpiar3(vec);
 return(vec);
 }                              //Limpiar3(aux);
}

voronoi *Limpiar4(voronoi *vor)
{voronoi *aux;
 if(vor!=NULL)
 {aux=vor;
  vor=vor->sig;
  aux->zona=Limpiar(aux->zona);
  aux->vec=Limpiar3(aux->vec);
  free(aux);
  vor=Limpiar4(vor);
  return(vor);
 }

}
```

A continuación presentamos las funciones que calcularán el *diagrama de Voronoi* de un conjunto de puntos. La primera de ellas, (`anadepunto`), añade un punto con su *región de Voronoi* a un *diagrama de Voronoi* ya existente. La segunda, (`CalculaVoronoi`), calcula del *diagrama de Voronoi* de un conjunto de puntos.

```
/*-----------------------------------------------------------------------------------
La siguiente función modifica el diagrama de voronoi al introducir un nuevo punto.
-----------------------------------------------------------------------------------*/

voronoi *anadepunto(voronoi *vor,punto p)
{voronoi *aux,*aux2,*visitas,*listavecinos;
 lista *aux1,*ps1,*ps2,*pe1,*pe2;
 punto s1,s2,e1,e2;
 voronoi::vecinos *aux3,*aux4;
 char c;
 int cort,v;
                                 //Creamos una zona de memoria para guardar el nodo asociado al punto p
 aux=(voronoi*)malloc(sizeof(voronoi));
                                //El nodo siguiente es NULL y guardamos el cuadrado unidad
 aux->p=p;
 aux->sig=NULL;
 aux->vec=NULL;
 aux->zona=(lista*)malloc(sizeof(lista));
 aux->zona->p.x=0.0;
 aux->zona->p.y=0.0;
 aux->zona->sig=NULL;
 aux1=aux->zona;
                                //Ya hemos guardado el (0,0), ahora los otros tres puntos
aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p.x=1.0;
 aux1->p.y=0.0;
 aux1->sig=NULL;
                                //Repetimos
```

```
 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p.x=1.0;
 aux1->p.y=1.0;
 aux1->sig=NULL;

 aux1->sig=(lista*)malloc(sizeof(lista));
 aux1=aux1->sig;
 aux1->p.x=0.0;
 aux1->p.y=1.0;
 aux1->sig=NULL;
                                //Ahora lo añadimos al final de la lista de voronoi
 if(vor==NULL)
  vor=aux;
 else
 {aux2=vor;
  while(aux2->sig!=NULL)
    aux2=aux2->sig;
  aux2->sig=aux;
 }

/*-----------------------------------------------------
Tenemos guardado al final de la lista  de voronoi  el
nodo correspondiente a p. Ahora buscamos  en que zona
de voronoi está y vamos actualizándolo todo
-----------------------------------------------------*/

aux2=vor;
while(fmod(cortes(p,aux2->zona),2.0)==0.0)
 aux2=aux2->sig;
                                //Nodo localizado. Trabajamos primero en la zona inicial
if(longvoronoi(vor)>1)
{puntosinterseccion(aux2->p,aux2->zona,p,&e1,&e2,&pe1,&pe2,&cort);
 if(cort==2)
  aux2->zona=nuevazona(aux2->zona,aux2->p,e1,e2,pe1,pe2);
 puntosinterseccion(aux->p,aux->zona,aux2->p,&s1,&s2,&ps1,&ps2,&cort);
 if(cort==2)
  aux->zona=nuevazona(aux->zona,aux->p,s1,s2,ps1,ps2);
                                //Guardamos nodos visitados
 visitas=NULL;
 visitas=(voronoi *)malloc(sizeof(voronoi));
 visitas->zona=NULL;
 visitas->sig=NULL;
 visitas->vec=NULL;
 visitas=sumavecino(visitas,aux2);
                                //Nodos a visitar
 listavecinos=NULL;
 listavecinos=(voronoi *)malloc(sizeof(voronoi));
 listavecinos->zona=NULL;
 listavecinos->sig=NULL;
 listavecinos->vec=NULL;
 aux3=aux2->vec;
 while(aux3!=NULL)
 {listavecinos=sumavecino(listavecinos,aux3->inf);
  aux3=aux3->sig;
 }
 while(listavecinos->vec!=NULL)
 {aux3=listavecinos->vec;
  listavecinos->vec=listavecinos->vec->sig;
 aux2=aux3->inf;
  puntosinterseccion(aux2->p,aux2->zona,p,&e1,&e2,&pe1,&pe2,&cort);
 switch(cort)
  {case 0:free(aux3);
          break;
   case 2:aux2->zona=nuevazona(aux2->zona,aux2->p,e1,e2,pe1,pe2);
          puntosinterseccion(aux->p,aux->zona,aux2->p,&s1,&s2,&ps1,&ps2,&cort);
          if(cort==2)
          aux->zona=nuevazona(aux->zona,aux->p,s1,s2,ps1,ps2);
          visitas=sumavecino(visitas,aux2);
          free(aux3);
          aux3=aux2->vec;
          while(aux3!=NULL)
          {v=0;
           aux4=visitas->vec;
           while(aux4!=NULL)
           {if(aux3->inf==aux4->inf) ++v;
            aux4=aux4->sig;
           }
           aux4=listavecinos->vec;
           while(aux4!=NULL)
```

```
           {if(aux3->inf==aux4->inf) ++v;
            aux4=aux4->sig;
           }
                              //Añadimos vecinos
           if(v==0)
            listavecinos=sumavecino(listavecinos,aux3->inf);
           aux3=aux3->sig;
          }
         break;
  }
 }

/*-------------------------------------------------
Ya hemos terminado. Ahora solamente falta por  actua-
lizar las vecindades de las zonas  de voronoi  que se
encuentran en visitas. Añadimos a cada nodo  visitado
aux y actualizamos vecindades.
-------------------------------------------------*/

 aux3=visitas->vec;
 while(aux3!=NULL)
 {aux3->inf=actualizavecindades(aux3->inf);
  aux3=aux3->sig;
 }
aux3=visitas->vec;
 while(aux3!=NULL)
 {aux=sumavecino(aux,aux3->inf);
  aux3->inf=sumavecino(aux3->inf,aux);
  aux3=aux3->sig;
}
//Ahora ya limpiamos visitas para la siguiente pues sus datos ya no se van a utilizar
 visitas=Limpiar4(visitas);
 listavecinos=Limpiar4(listavecinos);
 }
 return(vor);
}

/*-----------------------------------------------------------------------------------------
La siguiente función devuelve como solución el diagrama de voronoi de una nube de puntos.
-----------------------------------------------------------------------------------------*/

voronoi *CalculaVoronoi(voronoi *vor,lista *nube,double *a)
{lista *aux;
 voronoi *aux1;
 FILE *fvoronoi;
 vor=NULL;
 aux=nube;
 while(aux!=NULL)
 {vor=anadepunto(vor,aux->p);
 aux=aux->sig;
 }

/*-------------------------------------------------
Lo vamos a guardar ahora todo en el fichero  fvoronoi
que será voronoi.txt en  disco duro. Lo haremos igual
que los polígonos con agujeros. Marcamos 2.0 2.0 para
indicar que es un polígono.Las siguientes coordenadas
son siempre las del punto correspondiente a  la  zona
que acabamos de pintar
-------------------------------------------------*/

fvoronoi=fopen(''c:\\PVGC\\Pruebas_Heur_P2\\Matlab\\voronoi.txt'',''w+'');
aux1=vor;
while(aux1!=NULL)
{aux=aux1->zona;
 while(aux!=NULL)
 {fprintf(fvoronoi,''%f %f\n'',aux->p.x,aux->p.y);
  aux=aux->sig;
 }
 fprintf(fvoronoi,''%f %f\n'',2.0,2.0);
 fprintf(fvoronoi,''%f %f\n'',aux1->p.x,aux1->p.y);
 fprintf(fvoronoi,''%f %f\n'',2.0,3.0);
 if(aux1->sig==NULL)
 {*a=area_poligonobis(aux1->zona);
  fprintf(fvoronoi,''%lf %lf\n'',*a,(*a)*100);
 }
  aux1=aux1->sig;
 }
 fclose(fvoronoi);
```

```
 return(vor);
}/*fin construcción de voronoi*/
```

A continuación exponemos los códigos de cada una de las heurísticas construidas para el problema MaxA-p-Vor(N). Previamente exponemos dos funciones que nos ayudan a estudiar el área de la *región de Voronoi* asociada a un punto del cuadrado unidad. La primera de ellas copia un *diagrama de Voronoi* y la segunda es la función *fitness* asociada a un punto en un *diagrama de Voronoi*.

```
/*-----------------------------------------------------------------------------------
La siguiente función copia un diagrama de voronoi.
-----------------------------------------------------------------------------------*/

voronoi *copiarvoronoi(voronoi *vor,voronoi *cop)
{voronoi *aux,*auxcop,*aux1;
 lista *auxzona;
 voronoi::vecinos *auxvecinos;
 int i=1;

 aux=vor;
 cop=(voronoi*)malloc(sizeof(voronoi));
 cop->p=aux->p;
 cop->sig=NULL;
 cop->zona=NULL;
 cop->vec=NULL;
                                //Copiamos la primera zona
 auxzona=aux->zona;
 while(auxzona!=NULL)
 {cop->zona=Anadir(cop->zona,auxzona->p);
  auxzona=auxzona->sig;
 }
                                //Ahora copiamos el resto de las zonas
 auxcop=cop;
 aux=aux->sig;
 while(aux!=NULL)
 {auxcop->sig=(voronoi*)malloc(sizeof(voronoi));
  auxcop=auxcop->sig;
  auxcop->p=aux->p;
  auxcop->sig=NULL;
  auxcop->zona=NULL;
  auxcop->vec=NULL;
                                //Copiamos la zona
   auxzona=aux->zona;
  while(auxzona!=NULL)
  {auxcop->zona=Anadir(auxcop->zona,auxzona->p);
   auxzona=auxzona->sig;
  }
 aux=aux->sig;
 }

/*-----------------------------------------------------------------------------------
Ahora copiamos los vecinos.Debe tenerse en cuenta que primero se deben  copiar  todas las
zonas para luego hacer referencia a las  direcciones donde se encuentran los vecinos pero
en la copia, no en la original.
-----------------------------------------------------------------------------------*/

                                //Copiamos vecinos
 auxcop=cop;
 aux=vor;
 while(aux!=NULL)
 {auxvecinos=aux->vec;
  while(auxvecinos!=NULL)
  {aux1=cop;
   while(Iguales(aux1->p,auxvecinos->inf->p)==false)
    aux1=aux1->sig;
   auxcop=sumavecino(auxcop,aux1);
   auxvecinos=auxvecinos->sig;
  }
  aux=aux->sig;
  auxcop=auxcop->sig;
 }
 return(cop);
}
```

```
/*-------------------------------------------------------------------------------------
Diseñamos ahora dos funciones para limpiar memoria. Limpiar3  limpia una lista de vecinos
Diseñamos la función fitness, tal que dado un daigrama de voronoi y un  nuevo  punto  nos
devuelve el área de la zona del punto si lo añadimos al diagrama. Se ha debido  copiar el
diagrama, para que el antiguo no se modifique. Mucho cuidado pues las listas  de  vecinos
son punteros y nos se pueden copiar directamente, pues copiamos direcciones  del  voronoi
que se quiere preservar. Se debe mirar donde están los vecinos en la copia y copiar estas
direcciones nuevas en las listas de vecinos.
-------------------------------------------------------------------------------------*/

double FitnessVor(voronoi *vor,punto p)
{voronoi *copia,*auxv;
 double area;
 copia=NULL;
 copia=copiarvoronoi(vor,copia);
 copia=anadepunto(copia,p);
 auxv=copia;
 while(auxv->sig!=NULL)
  auxv=auxv->sig;
 area=areapolvisi(p,auxv->zona);
 copia=Limpiar4(copia);
 return(area);
}
```

Implementaciones para la heurística simulated annealing-SA

Los códigos para SA siguen la misma estructura que los expuetos para el problema MaxA-p-Pv1(P).

```
voronoi *MaxSA_VOR(int tipor,double tpi,double dt,int *itera,voronoi *vorini,lista *conjunto,punto *p, double
*a)
{unsigned long int i,k;
 long hora;
 int horasistema;
 punto p1,p2;
 double areaux,areauy,delta,tk,U,U1,U2,div;
 FILE *rastro,*datoscurva;
 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''w+'');
 datoscurva=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\datoscurva.txt'',''w+'');

                              //Generamos una primera solución

 hora=time(NULL); //Generamos una semilla de sistema
 horasistema=(unsigned int) hora/2;
 srand(horasistema);
                             //Creamos el punto/
 p1.x=(1.0*rand())/(RAND_MAX);
 p1.y=(1.0*rand())/(RAND_MAX);
                             //Calculamos el área de p1
 areaux=FitnessVor(vorini,p1);
 fprintf(rastro,''%f %f\n'',p1.x,p1.y);

                            //Lo guardamos en rastro
                            //Guardamos datos para curva crecimiento

 fprintf(datoscurva,''%lf\n'',areaux*100.0);
 k=1;
 div=10.0;
 tk=tpi;                     //TEMPERATURA INICIAL
 do                         //Generamos una nueva solución
 {i=1;
  do
  {do
   {U1=(1.0*rand())/(RAND_MAX);
    U2=(1.0*rand())/(RAND_MAX);
    p2.x=p1.x+((sqrt(-(2.0*log(U1)))*sin(2.0*pi*U2))/div);
    p2.y=p1.y+((sqrt(-(2.0*log(U1)))*cos(2.0*pi*U2))/div);
   }while((p2.x>=1)||(p2.x<=0)||(p2.y>=1)||(p2.y<=0));
   areauy=FitnessVor(vorini,p2);
   *itera=*itera+1;
                            //Calculamos delta
   delta=areauy-areaux;
```

```
                              //Estudiamos el cambio
   if(delta>=0)
   {areaux=areauy;
    p1=p2;
   }                          //Sino depende de la temperatura
   else
   {U=(1.0*rand())/(RAND_MAX);
    if(U<exp((1000*delta)/tk))
    {printf(''*'');
     areaux=areauy;
     p1=p2;
    }
   }
   ++i;
   fprintf(rastro,''%f %f\n'',p2.x,p2.y);
   fprintf(datoscurva,''%lf\n'',(*a)*100.0);
 }while ((i)<=(tk));
  switch(tipor)
  {case 0:tk=dt*tk;break;
   case 1:tk=(tpi)/log(1+k);break;
   case 2:tk=(tpi)/(1+k); break;
   case 3:tk=(tpi)/exp(k); break;
   case 4:tk=(tpi)/(exp(exp(k)));break;
   case 5:tk=(0.9)*tk;break;
  }
  k=k+1;
  div=div/0.999;
  printf(''\n\t%lf.....%lf\t'',tk,areaux);
 }while(tk>0.025);
 *p=p1;                           //Actualizamos la salida
 *a=areaux;
 conjunto=Anadir(conjunto,*p);
 vorini=CalculaVoronoi(vorini,conjunto,&*a);
 fclose(rastro);
 fclose(datoscurva);
 return(vorini);

}
```

Implementaciones para la heurística random search-RS

Asímismo, las implementaciones para RS son las siguientes:

```
voronoi *MaxRS_VOR(lista *nube,voronoi *vorini,lista *conjunto,punto *p, double *a)
{lista *aux;
 double areaux=0.0;
 int i=1;
 FILE *rastro,*datoscurva;
 rastro=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\rastro.txt'',''w+'');
 datoscurva=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\datoscurva.txt'',''w+'');
 *a=0.0;
 (*p).x=0.0;
 (*p).y=0.0;
 aux=nube;
 while(aux!=NULL)
 {areaux=FitnessVor(vorini,aux->p);
  printf(''\n\t\t Punto %d analizado (%lf,%lf) Area %lf'',i,aux->p.x,aux->p.y,areaux);
  fprintf(rastro,''%lf %lf\n'',aux->p.x,aux->p.y);
  fprintf(datoscurva,''%lf\n'',areaux*100.0);
  if(areaux>(*a))
  {*p=aux->p;
   *a=areaux;
  }
  aux=aux->sig;
  ++i;
}
 conjunto=Anadir(conjunto,*p);
 vorini=CalculaVoronoi(vorini,conjunto,&*a);
 fclose(rastro);
 fclose(datoscurva);
 return(vorini);

}
```

B.8 Funciones de cálculos generales y programa principal.

Las técnicas heurísticas implementadas necesitan un conjunto de funciones auxiliares para su ejecución. Estas funciones están relacionadas con el manejo de estructuras geométricas clásicas. Así, se necesita saber cuando dos rectas o dos segmentos se cortan, la longitud de un segmento, etc. Algunas de estas funciones están contenidas en secciones anteriores, sin embargo, otras más generales se presentan a continuación junto programa principal diseñado para la ejecución de todos los métodos. Debe recordarse que no se han incluido todas estas funciones, (para reducir la extensión de este apéndice), y que sólamente se exponen las que se han considerado más relevantes.

```
/*-------------------------------------------------------------------------------------
La siguiente función calcula el producto vectorial de dos vectores  son  extremos  en  un
punto.
-------------------------------------------------------------------------------------*/

double vectorial (punto p1,punto p2,punto p3)
{punto u,v;
                              //Calculamos el primer vector
u.x=(p2.x-p1.x);
 u.y=(p2.y-p1.y);
                              //Calculamos el segundo vector
 v.x=(p3.x-p1.x);
 v.y=(p3.y-p1.y);
 return((u.x*v.y)-(v.x*u.y));
}

/*-------------------------------------------------------------------------------------
Una función que estudia si dos segmentos se cortan.
-------------------------------------------------------------------------------------*/

bool cortan(punto p1,punto p2, punto p3, punto p4)
{double v1,v2,w1,w2;
 v1=p2.x-p1.x;
 v2=p2.y-p1.y;
 w1=p4.x-p3.x;
 w2=p4.y-p3.y;
if((v1*w2)==(v2*w1))
                              //Son paralelas
  return(false);
 else                         //Miramos si hay 2 iguales

  if(((p1.x==p3.x) && (p1.y==p3.y))||((p1.x==p4.x)&&(p1.y==p4.y))||
    ((p2.x==p3.x)&&(p2.y==p3.y))||((p2.x==p4.x)&&(p2.y==p4.y)))
  return(true);
  else
   if((vectorial(p1,p3,p4)==0)||(vectorial(p2,p3,p4)==0)||
     (vectorial(p3,p1,p2)==0)||(vectorial(p4,p1,p2)==0))
    return(true);
   else
    if((vectorial(p1,p3,p4)/vectorial(p2,p3,p4)>0)||
      (vectorial(p3,p1,p2)/vectorial(p4,p1,p2)>0))
     return(false);
    else
     return(true);
}

void delay(long i)
{long x,y,z;
for(x=0;x<=i;++x)
for(y=0;y<=i;++y)
for(z=0;z<=i;++z);
}

/*-------------------------------------------------------------------------------------
Función Es_Ordenada_vertice determina que la ordenada de un determinado  punto no coinci-
da con la de ningún vértice, para que no se puedan producir cortes en vértices y de esta
manera puntos que están fuera del polígono podrían incluirse en la nube de puntos.
-------------------------------------------------------------------------------------*/
```

```
bool Es_Ordenada_vertice(punto punt,lista *polig)
{unsigned long int i;
 lista *aux;
 i=0;
 aux=polig;
 while(aux!=NULL)
 {if(punt.y==aux->p.y) ++i;
   aux=aux->sig;
 }
 if(i==0)
  return(false);
 else
  return(true);
}

/*-----------------------------------------------------------------------------------
La función  cortes determina el número de cortes entre los lados de un polígono y el seg-
mento que  une un punto generado con el punto que está en la misma ordenada y la abcisa
en 1.
-----------------------------------------------------------------------------------*/

int cortes(punto punt, lista *polig)
{lista *aux;
 int c=0;
 punto punt1;
 aux=polig;
 punt1.x=2.0;
                                //Los extremos se consideran sólo una vez
 punt1.y=punt.y;
 if(aux!=NULL)
 {while (aux->sig!=NULL)
  {if((cortan(punt,punt1,aux->p,aux->sig->p)==true)&&(punt.y!=aux->p.y)) ++c;
    aux=aux->sig;
  }

                                //Como el último lado del polígono no lo tenemos en
                                //cuenta se debe estudiar por separado
 if((cortan(punt,punt1,aux->p,polig->p)==true)&&(punt.y!=aux->p.y)) ++c;
 }
 return(c);
}

/*-----------------------------------------------------------------------------------
La siguiente función crea una nube de puntos interiores a un polígono.  Será necesaria en
la heurística random-search.
-----------------------------------------------------------------------------------*/

void creanube (lista *nube,lista *polig,unsigned long int n)
{unsigned long int i;
 lista *aux;
 punto punt;
                                //Fichero para guardar la nube
 int horasistema;
 int hora;
FILE *fnube;
 fnube=fopen(''c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\nube.txt'',''w+'');
 hora=time(NULL);
                                //Generamos una semilla de sistema

 horasistema=(unsigned int) hora/n;
 srand(horasistema);
                                //Creamos el primer punto
 punt.x=(1.0*rand())/(RAND_MAX);
 punt.y=(1.0*rand())/(RAND_MAX);
while((fmod(cortes(punt,polig),2.0)==0.0)||(Es_Ordenada_vertice(punt,polig)==true))
 {punt.x=(1.0*rand())/(RAND_MAX);
  punt.y=(1.0*rand())/(RAND_MAX);
 }
nube->p.x=punt.x;
 nube->p.y=punt.y;
 nube->sig=NULL;
                                //Guardamos en la nube

 fprintf(fnube,''%f %f\n'',punt.x,punt.y);
aux=nube;
i=2;
 while (i<=n)
 {punt.x=(1.0*rand())/(RAND_MAX);
```

```
  punt.y=(1.0*rand())/(RAND_MAX);
                                //Genero un punto y lo verifico
  while((fmod(cortes(punt,polig),2.0)==0.0)||(Es_Ordenada_vertice(punt,polig)==true))
  {punt.x=(1.0*rand())/(RAND_MAX);
   punt.y=(1.0*rand())/(RAND_MAX);
  }
  if (i==n)
  {aux->sig=(lista*)malloc(sizeof(lista));
   aux=aux->sig;
   aux->p.x=punt.x;
   aux->p.y=punt.y;
   aux->sig=NULL;
   fprintf(fnube,''%f %f\n'',punt.x,punt.y);
  }
  else
  {aux->sig=(lista*)malloc(sizeof(lista));
   aux=aux->sig;
   aux->p.x=punt.x;
   aux->p.y=punt.y;
   fprintf(fnube,''%f %f\n'',punt.x,punt.y);
  }
  ++i;
 }
fclose(fnube);
}

/*-------------------------------------------------------------------------------------
Esta función simplemente visualiza los datos de una lista.
-------------------------------------------------------------------------------------*/

void ver (lista *p)
{lista *aux; unsigned long int i=1;
 aux=p;
 printf(''\n*****************'');
 while (aux!=NULL)
 {printf(''\n (%f , %f) %c'',aux->p.x,aux->p.y,aux->p.tipo);
  aux=aux->sig;
  ++i;
 }
}

/*-------------------------------------------------------------------------------------
Otra función para ver un polígono general.
-------------------------------------------------------------------------------------*/

 void ver2 (poligono *p)
 {poligono *aux; unsigned long int i=1;
  aux=p;
  printf(''\n*****************'');
  while (aux!=NULL)
  {printf(''\n Polígono %d'',i);
   ver(aux->dat.puntos);
   aux=aux->sig;
   ++i;
  }
 }

/*-------------------------------------------------------------------------------------
Construimos funciones que nos permitirá tener menús con las diferentes técnicas  que ire-
mos onstruyendo.
-------------------------------------------------------------------------------------*/

void Menu1(int *op)
{system(''cls'');
 printf(''\n\n\t\t============================================================'');
 printf(''\n\t\t OPCIONES GENERALES DE IMPLEMENTACION:'');
 printf(''\n\t\t-------------------------------------------------------\n'');
 printf(''\n\t\t 1.- Algoritmos Pv1.\n'');
 printf(''\n\t\t 2.- Algoritmos Pvk.\n'');
 printf(''\n\t\t 3.- Solucion MIN_P_Pvk_RS.\n'');
 printf(''\n\t\t 4.- Solucion MIN_P_Pvk_SA .\n'');
 printf(''\n\t\t 5.- Solucion MIN_P_Pvk_GNT.\n'');
 printf(''\n\t\t-------------------------------------------------------'');
 printf(''\n\t\tOpcion?..'');
 scanf(''%d'',op);
```

```
}
                                //Genético en PvK
void Menu2(int *op)
{system(''cls'');
 printf(''\n\n\t\t========================================================'');
 printf(''\n\t\t MENU DE OPCIONES IMPLEMENTADAS PV1:'');
 printf(''\n\t\t----------------------------------------------------\n'');
 printf(''\n\t\t 1.- Algoritmo MaxRS_PV1.'');
 printf(''\n\t\t 2.- Algoritmo MaxSA_PV1.'');
 printf(''\n\t\t 3.- Algoritmo MaxGRAD_PV1.'');
 printf(''\n\t\t 4.- Superficie de Areas por Mallas.\n'');
 printf(''\n\t\t 5.- Por la Descomposicion S.\n'');
 printf(''\n\t\t----------------------------------------------------'');
 printf(''\n\t\tOpcion?..'');
 scanf(''%d'',op);
}

void Menu3(int *op)
{system(''cls'');
 printf(''\n\n\t\t========================================================'');
 printf(''\n\t\t MENU DE OPCIONES IMPLEMENTADAS PVK:'');
 printf(''\n\t\t----------------------------------------------------\n'');
 printf(''\n\t\t 1.- Construccion Union.'');
 printf(''\n\t\t 2.- Algoritmo MaxRS_PVK.'');
 printf(''\n\t\t 3.- Algoritmo MaxGNT_PVk.'');
 printf(''\n\t\t 4.- Algoritmo MaxSA_PVk.'');
 printf(''\n\t\t----------------------------------------------------'');
 printf(''\n\t\tOpcion?..'');
 scanf(''%d'',op);
 }

void MenuResultados(int *op)
{system(''cls'');
 printf(''\n\n\t\t========================================================'');
 printf(''\n\t MENU DE RESULTADOS DE TODAS OPCIONES IMPLEMENTADAS:'');
 printf(''\n\t\t----------------------------------------------------\n'');
 printf(''\n\t\t 1.- Resultados MaxRS_PV1.'');
 printf(''\n\t\t 2.- Resultados MaxSAT0-Tp1=100-Tr=0.9-0.005_PV1.'');
 printf(''\n\t\t 3.- Resultados MaxRs_Casos.'');
 printf(''\n\t\t 4.- Resultados MaxS_PV1.\n'');
 printf(''\n\t\t----------------------------------------------------'');
 printf(''\n\t\tOpcion?..'');
 scanf(''%d'',op);

}
```

A continuación incluimos el código del programa principal, o función principal:

```
/*-------------------------------------------------------------------------------------
SOFTWARE HEURISTICAS : Tesis Doctoral.
-------------------------------------------------------------------------------------*/
                                //Constantes
#define Tamcierre 1000
#define pi 3.1416
#define Tampobla 325
                                //Ficheros include
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stddef.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>
                                //Ficheros aux.
#include ''tipos.h''
#include ''prototipos.h''
#include ''general.h''
#include ''rpg.h''
#include ''Max_PV1.h''
#include ''Max_PV2.h''
#include ''Max_PVK.h''
```

```
#include ''resultados.h''
#include ''temporal.h''
#include ''voronoi.h''
#include ''superficie.h''
#include ''reduccion.h''

/*-------------------------------------------------------------------------------------
Función principal. Llama a todas las funciones que ejecutan las heurísticas.
-------------------------------------------------------------------------------------*/

void main()
{lista *poligonoreal,*poligono1real,*poligono2real,*poligonounion,*indik;
 lista *nubereal;
 lista2 *nubereal2;
 lista *conjunto;
 lista *polvisireal,*polvisireal2;
 poligono *poliunionreal;
 punto puntoreal;
                                //Declaración variables

 double aunion,area1,area2,ainter,apreal,apvisi,tempini,decretemp,cireal,redreal;
 unsigned long int tipo,vertices,puntos,individuos,antsreal,antsinireal,itereal;
 int opcion,opcion1,opcion2,opcion3,opcion4,tiporreal,horasistema,i,k,nrectas,iteraciones;
 int tamano,verti,llevamos;
 char c,nombre[100];
 punto p1,p2;
 long hora;
 voronoi *diagramavoronoi;
 double avoronoi;
apvisi=0;
system(''cls'');
 printf(''\n\n\t\t==========================================='');
 printf(''\n\t\t\tMETODOS APROXIMADOS EN PROBLEMAS GEOMETRICOS'');
 printf(''\n\t\t\t-------------------------------------------'');
 printf(''\n\n\t\t\t 1.- PROGRAMA DE PRUEBAS P1'');
 printf(''\n\n\t\t\t 2.- PROGRAMA DE RESULTADOS P1\n\n'');
 printf(''\n\t\t\t\t 3.- PROGRAMA DE PRUEBAS P2'');
 printf(''\n\n\t\t\t 4.- PROGRAMA DE RESULTADOS P2\n\n'');
 printf(''\n\t\t\t\t 5.- GENERACION ALEATORIA DE POLIGONOS\n'');
 printf(''\n\t\t\t-------------------------------------------'');
 printf(''\n\t\t\t==========================================='');
 printf(''\n\t\tOpcion?..'');
 scanf(''%d'',&opcion);
if(opcion==1)
 {do
  {system(''cls'');
   printf(''\n\t\t\t PROGRAMA DE PRUEBAS P1:'');
   printf(''\n\t\t\t---------------------------------'');
   printf(''\n\t\t\t * Generar poligono S/N?..'');
   fflush(stdin);
   c=toupper(getche());
   if((c=='S'))
   {printf(''\n\n\t\t\t * Vertices? ..'');
    scanf(''%ld'',&vertices);
    printf(''\n\t\t\t * Tipo: General (1) Monotono (2)'');
    scanf(''%ld'',&tipo);
                                //Semilla aleatoria
    hora=time(NULL);
    horasistema=(unsigned int) hora/2;
    srand(horasistema);
                                //Conjunto inicial
   conjunto=(lista*) malloc(sizeof(lista));
    Conjunto_Inicial(conjunto,vertices);
    poligonoreal=(lista*) malloc(sizeof(lista));
    switch(tipo)
    {case 1:RPG(poligonoreal,conjunto);break;
     case 2:RPG_MONOTONO(poligonoreal,conjunto);break;
   }
  }
   else
   {printf(''\n\n\t\t\t * Pinte el poligono y presione enter'');
    getche();
    poligonoreal=(lista*) malloc(sizeof(lista));
    Lectura_RPG(poligonoreal);
  }
   printf(''\n\t\t.....................................................'');
```

```
printf(''\n\t\t * Poligono Generado'');
apreal=area_poligono(poligonoreal);
printf(''\n\t\t * Poligono Triangulado'');
printf(''\n\t\t * El AREA del Poligono Generado es %lf'',apreal);
printf(''\n\t\t * El AREA del Poligono Visibilidad %lf'',apvisi);
printf(''\n\t\t.....................................................'');
c=getche();
Menu1(&opcion1);
switch(opcion1)
{case 1:Menu2(&opcion2);
 switch(opcion2)
 {case 1:printf(''\n\t\t 1.-ALGORITMO RANDOM-SEARCH:'');
         printf(''\n\t\t ----------------------'');
         printf(''\n\n\t\t * Puntos Interiores? ..'');
         scanf(''%ld'',&puntos);
         nubereal=(lista*) malloc(sizeof(lista));
         creanube(nubereal,poligonoreal,puntos);
         polvisireal=(lista*) malloc(sizeof(lista));
         polvisireal->sig=NULL;
         polvisireal=MaxRS_PV1(nubereal,poligonoreal,polvisireal,&puntoreal,&apvisi,apreal);
         printf(''\n\n\t\t * Poligono de visibilidad calculado'');
         printf(''\n\n\t\t - Punto de area maxima........\t(%lf,%lf)'',puntoreal.x,puntoreal.y);
         printf(''\n\n\t\t - Area maxima.................\t%lf'',apvisi);
         nubereal=Limpiar(nubereal);
         polvisireal=Limpiar(polvisireal);
         break;
  case 2:printf(''\n\t\t 2.-ALGORITMO SIMULATED ANNEALING:'');
         printf(''\n\t\t ----------------'');
         printf(''\n\t\t * Tipo de Reducción? ..'');
         scanf(''%d'',&tiporreal);
         printf(''\n\t\t * Temperatura Inicial? ..'');
         scanf(''%lf'',&tempini);
         decretemp=0;
         if(tiporreal==0)
          {printf(''\n\t\t * Decremento Temperatura? ..'');
           scanf(''%lf'',&decretemp);
          }
         polvisireal=(lista*) malloc(sizeof(lista));
         system(''cls'');
         printf(''\n\n\t======================================================================'');
         printf(''\n\t MaxSA_PV1 DIAGRAMA DE ACEPTACION DE MOVIMIENTOS:'');
         printf(''\n\n\t======================================================================'');
         printf(''\n\t\t\t'');
          polvisireal=MaxSA_PV1(tiporreal,tempini,decretemp,&iteraciones,poligonoreal,polvisireal,
                                &puntoreal,&apvisi,apreal);
         printf(''\n\n\t\t * Poligono de visibilidad calculado'');
         printf(''\n\t\t - Punto de area maxima = (%lf,%lf)'',puntoreal.x,puntoreal.y);
         printf(''\n\t\t - Area maxima %lf'',apvisi);
         polvisireal=Limpiar(polvisireal);
         break;
  case 3:printf(''\n\t\t 3.-ALGORITMO GRADIENTE:'');
         printf(''\n\t\t ----------------------'');
         polvisireal=(lista*) malloc(sizeof(lista));
          polvisireal->sig=NULL;
         polvisireal=MaxGRAD_PV1(&iteraciones,poligonoreal,polvisireal,&puntoreal,&apvisi,apreal);
         printf(''\n\n\t\t * Poligono de visibilidad calculado'');
         printf(''\n\n\t\t - Punto de area maxima........\t(%lf,%lf)'',puntoreal.x,puntoreal.y);
         printf(''\n\n\t\t - Area maxima.................\t%lf'',apvisi);
         polvisireal=Limpiar(polvisireal);
         break;
  case 4:printf(''\n\t\t 4.-CALCULO DE LA SUPERFICIE DE AREAS POR MALLAS:'');
         printf(''\n\t\t ------------------------------------------'');
         SuperficieBis(poligonoreal);
         printf(''\n\n\t\t * Superficie finalizada por mallas'');
         break;
  case 5:printf(''\n\t\t 9.-DESCOMPOSICION S:'');
         printf(''\n\t\t ----------------------'');
         polvisireal=(lista*) malloc(sizeof(lista));
         polvisireal->sig=NULL;
         polvisireal=MaxS_PV1(&iteraciones,poligonoreal,polvisireal,&puntoreal,&apvisi,apreal);
         printf(''\n\n\t\t * Poligono de visibilidad calculado'');
         printf(''\n\n\t\t - Punto de area maxima........\t(%lf,%lf)'',puntoreal.x,puntoreal.y);
```

```
        printf(''\n\n\t\t - Area maxima.................\t%lf'',apvisi);
        polvisireal=Limpiar(polvisireal);
        break;
}break;
case 2:Menu3(&opcion3);
switch(opcion3)
{case 1:printf(''\n\t\t 1.-PROBANDO LA UNION:'');
        printf(''\n\t\t -----------------'');
        printf(''\n\t\t\t * Generar nube S/N?..'');
        fflush(stdin);
        c=toupper(getche());
       if((c=='S'))
        {printf(''\n\n\t\t\t * Puntos Interiores? ..'');
         scanf(''%ld'',&puntos);
         printf(''\n Nube de %ld'',puntos);
         nubereal=NULL;
         nubereal=(lista*) malloc(sizeof(lista));
         creanube(nubereal,poligonoreal,puntos);
         poliunionreal=NULL;
         poliunionreal=Construye_Union(poliunionreal,poligonoreal,nubereal,apreal,&apvisi);
        }
        else
        {printf(''\n\n\t\t\t * Pinte la nube y presione enter'');
         getche();
         nubereal=(lista*) malloc(sizeof(lista));
         Lectura_NUBE(nubereal);
         printf(''\n\n\t\t\t * Nube Leida\n'');
         poliunionreal=NULL;
         poliunionreal=Construye_Union(poliunionreal,poligonoreal,nubereal,apreal,&apvisi);
        }
        break;
 case 2:printf(''\n\t\t 2.-ALGORITMO RANDOM-SEARCH_PVK:'');
        printf(''\n\t\t ---------------------------------'');
        printf(''\n\n\t\t\t * Valor de k? ..'');
        fflush(stdin);
        scanf(''%d'',&k);
        printf(''\n\n\t\t\t * Cantidad de k-elementos? ..'');
        scanf(''%ld'',&puntos);
        nubereal=NULL;
        nubereal=(lista*) malloc(sizeof(lista));
        creanube(nubereal,poligonoreal,k*puntos);
        poliunionreal=NULL;
        indik=(lista*) malloc(sizeof(lista));
        indik->sig=NULL;
        poliunionreal=MaxRS_PVK(nubereal,poligonoreal,poliunionreal,indik,k,apreal,&apvisi);
        break;
 case 3:printf(''\n\t\t 3.-ALGORITMO GENETICO_PVK:'');
        printf(''\n\t\t ----------------------------'');
        printf(''\n\n\t\t\t * Valor de k? ..'');
        fflush(stdin);
        scanf(''%d'',&k);
        poliunionreal=NULL;
        indik=(lista*) malloc(sizeof(lista));
        indik->sig=NULL;
        poliunionreal=MaxGNT_PVK(&iteraciones,poligonoreal,poliunionreal,indik,k,apreal,&apvisi);
        break;
 case 4:printf(''\n\t\t 4.-ALGORITMO SIMULATED ANNEALING_PVK:'');
        printf(''\n\t\t -------------------------------------'');
        printf(''\n\n\t\t\t * Valor de k? ..'');
        fflush(stdin);
        scanf(''%d'',&k);
        printf(''\n\t\t * Tipo de Reducción? ..'');
        scanf(''%d'',&tiporreal);
        printf(''\n\t\t * Temperatura Inicial? ..'');
        scanf(''%lf'',&tempini);
        decretemp=0;
        if(tiporreal==0)
        {printf(''\n\t\t * Decremento Temperatura? ..'');
         scanf(''%lf'',&decretemp);
        }
        poliunionreal=NULL;
        indik=(lista*) malloc(sizeof(lista));
        indik->sig=NULL;
        system(''cls'');
        printf(''\n\n\t======================================================================'');
```

```
            printf(''\n\t MaxRS_PVK DIAGRAMA DE ACEPTACION DE MOVIMIENTOS:'');
            printf(''\n\n\t======================================================================'');
            printf(''\n\t\t\t'');
             poliunionreal=MaxRS_PVK(tiporreal,tempini,decretemp,&iteraciones,poligonoreal,
                                poliunionreal,indik,k,apreal,&apvisi);
            break;
     case 5:printf(''\n\t\t 5.-ALGORITMO DEL GRADIENTE MaxGrad_PVK:'');
            printf(''\n\t\t ----------------------------------'');
            printf(''\n\n\t\t\t * Valor de k? ..'');
            fflush(stdin);
            scanf(''%d'',&k);
            poliunionreal=NULL;
            indik=(lista*) malloc(sizeof(lista));
            indik->sig=NULL;
            poliunionreal=MaxGRAD_PVK(poligonoreal,poliunionreal,indik,k,apreal,&apvisi);
            break;
    }break;
    case 3:printf(''\n\t\t 3.- SOLUCIÓN MIN_P_PVK_RS:'');
           printf(''\n\t\t ------------------------'');
           Min_P_PVK_RS(poligonoreal,apreal);
           break;
    case 4:printf(''\n\t\t 4.- SOLUCIÓN MIN_P_PVK_SA:'');
           printf(''\n\t\t ------------------------'');
           Min_P_PVK_SA(poligonoreal,apreal);
           break;
    case 5:printf(''\n\t\t 6.- SOLUCIÓN MIN_P_PVK_GNT:'');
           printf(''\n\t\t ------------------------'');
           Min_P_PVK_GNT(poligonoreal,apreal);
           break;
  }/*del 1ºswitch*/
  printf(''\n\n\t * Desea repetir el programa S/N.. '');
  fflush(stdin);
  c=toupper(getchar());
  if(c=='N')
  poligonoreal=Limpiar(poligonoreal);
 }while(c!='N');
}                          //fin de PROGRAMA DE PRUEBAS
else

//PROGRAMA DE RESULTADOS.ESTAS FUNCIONES NO SE INCLUYEN*/
//LLAMADAS ITERATIVAS A MÉTODOS.

if(opcion==2)
{MenuResultados(&opcion4);
 switch(opcion4)
 {case 1:Resultados_MaxDet_PV1();break;
  case 2:Resultados_MaxRST0_PV1();break;
  case 4:Resultados_MaxCasos();break;
  case 5:Resultados_MaxS_PV1();break;
}
}                          /*fin PROGRAMA DE RESULTADOS*/

else
 if(opcion==3)
 {do
  {system(''cls'');
   printf(''\n\t\t\t PROGRAMA DE PRUEBAS P2:'');
   printf(''\n\t\t\t--------------------------------'');
    printf(''\n\t\t\t * Generar conjunto de puntos S/N?..'');
   fflush(stdin);
   c=toupper(getche());
   if((c=='S'))
   {printf(''\n\n\t\t\t * Puntos? ..'');
    scanf(''%ld'',&puntos);
                               //Genero un conjunto de puntos del cual calcular el diagrama de voronoi
    hora=time(NULL);
    horasistema=(unsigned int) hora/2;
    srand(horasistema);
    conjunto=(lista*) malloc(sizeof(lista));
    Conjunto_Inicial(conjunto,puntos);
   printf(''\n-Conjunto generado'');
   }
   else                        //El conjunto se leerça del fichero

   {printf(''\n\n\t\t\t * Pinte el conjunto y presione enter '');
    getche();
```

```
    conjunto=(lista*) malloc(sizeof(lista));
    Lectura_CONJUNTO(conjunto);
   }                            //Ahora ya calculamos el diagrama de voronoi

   diagramavoronoi=NULL;
   diagramavoronoi=CalculaVoronoi(diagramavoronoi,conjunto,&avoronoi);
   system(''cls'');
   printf(''\n\n\t\t=======================================================');
   printf(''\n\t\t MENU DE OPCIONES IMPLEMENTADAS MaxA-P-Vor:'');
   printf(''\n\t\t-------------------------------------------------------\n'');
   printf(''\n\t\t 1.- Algoritmo MaxRS_VOR.'');
   printf(''\n\t\t 2.- Algoritmo MaxSA_VOR.'');
   printf(''\n\t\t 3.- Calculo de la Superficie de Voronoi.'');
   printf(''\n\n\t\t-------------------------------------------------------\n'');
   printf(''\n Opcion...'');
   scanf(''%d'',&opcion);
   switch(opcion)
   {case 1:printf(''\n\t\t 1.-ALGORITMO RANDOM SEARCH:'');
           printf(''\n\t\t ----------------------'');
           printf(''\n\n\t\t * Puntos a analizar? ..'');
           scanf(''%ld'',&puntos);
           hora=time(NULL);
           horasistema=(unsigned int) hora/2;
           srand(horasistema);
           nubereal=(lista*) malloc(sizeof(lista));
           Conjunto_Inicialbis(nubereal,puntos);
           diagramavoronoi=MaxRS_VOR(nubereal,diagramavoronoi,conjunto,&puntoreal,&avoronoi);
           printf(''\n\n\t\t * Zona de area maxima calculada'');
           printf(''\n\n\t\t - Punto de area maxima.....\t(%lf,%lf)'',puntoreal.x,puntoreal.y);
           printf(''\n\n\t\t - Area maxima.............\t%lf'',avoronoi);
           nubereal=Limpiar(nubereal);
           break;
    case 2:printf(''\n\t\t 2.-ALGORITMO SIMULATED ANNEALING:'');
           printf(''\n\t\t ----------------------------'');
           printf(''\n\n\t\t * Tipo de Reducción? ..'');
           scanf(''%d'',&tiporreal);
           printf(''\n\n\t\t * Temperatura Inicial? ..'');
           scanf(''%lf'',&tempini);
           decretemp=0;
           if(tiporreal==0)
           {printf(''\n\t\t * Decremento Temperatura? ..'');
            scanf(''%lf'',&decretemp);
           }
           system(''cls'');
           printf(''\n\n\t=====================================================================');
           printf(''\n\t MaxSA_VOR DIAGRAMA DE ACEPTACION DE MOVIMIENTOS:'');
           printf(''\n\n\t=====================================================================');
           printf(''\n\t\t\t'');
           iteraciones=0;
           diagramavoronoi=MaxSA_VOR(tiporreal,tempini,decretemp,&iteraciones,diagramavoronoi,conjunto,
                                     &puntoreal,&avoronoi);
           printf(''\n\n\t\t * Zona de area maxima calculada'');
           printf(''\n\n\t\t - Punto de area maxima.....\t(%lf,%lf)'',puntoreal.x,puntoreal.y);
           printf(''\n\n\t\t - Area maxima.............\t%lf'',avoronoi);
           printf(''\n\n\t\t - Iteraciones.............\t%d'',iteraciones);
           break;
    case 3:SuperficieVor(diagramavoronoi);
           break;
   }
   diagramavoronoi=Limpiar4(diagramavoronoi);
   conjunto=Limpiar(conjunto);
   printf(''\n\n\t * Desea repetir el programa S/N.. '');
   fflush(stdin);
   c=toupper(getchar());
   if(c=='N')
    conjunto=Limpiar(conjunto);
  }while(c!='N');
}/*fin construcción diagrama de Voronoi y por tanto opción 3*/

//  Opciones de resultados para MaxA-P-Vor(N).
//ESTAS FUNCIONES NO SE INCLUYEN EN EL APÉNDICE

else
 if(opcion==4)
```

```
 {system(''cls'');
  printf(''\n\n\t\t======================================================='');
  printf(''\n\t MENU DE OPCIONES IMPLEMENTADAS RESULTADOS MaxA-P-Vor:'');
  printf(''\n\t\t-------------------------------------------------------\n'');
  printf(''\n\t\t 1.- Resultados Algoritmo MaxRS_VOR.'');
  printf(''\n\t\t 2.- Resultados Algoritmo MaxSA_VOR.'');
  printf(''\n\n\t\t-------------------------------------------------------\n'');
  printf(''\n Opcion...'');
  scanf(''%d'',&opcion);
  switch(opcion)
  {case 1:printf(''\n\t\t 1.- RESULTADOS ALGORITMO RANDOM SEARCH:'');
          ResultadosMaxRS_VOR();
          break;
   case 2:printf(''\n\t\t 1.- RESULTADOS ALGORITMO SIMULATED ANNEALING:'');
          ResultadosMaxSA_VOR();
          break;
 }
 }
else
 if(opcion==5)
 {do
  {system(''cls'');
   printf(''\n\t\t\t======================================'');
   printf(''\n\t\t\t PROGRAMA DE GENERACION POLIGONOS:'');
   printf(''\n\t\t\t--------------------------------------'');
   printf(''\n\n\t\t\t * Cantidad de poligonos?..'');
   fflush(stdin);
   scanf(''%d'',&tamano);
   printf(''\n\n\t\t\t * Vertices de cada poligono?..'');
   scanf(''%d'',&verti);
   printf(''\n\n\t\t\t * Nombre del fichero?..'');
   fflush(stdin);
   gets(nombre);
   printf(''\n\n\t\t\t * Cuantos llevamos?..'');
   scanf(''%d'',&llevamos);
   Generar_Poligonos(tamano,verti,nombre,llevamos);
   poliunionreal=NULL;
   poliunionreal=Lectura_Poligonos(poliunionreal,nombre);
   ver2(poliunionreal);
  printf(''\n\n\t * Desea repetir el programa S/N.. '');
   fflush(stdin);
   c=toupper(getchar());
 }while(c!='N');
 }
}/*fin main*/
```

B.9 Subrutinas de visualización de datos con MatLab

Incluimos a continuación los programas implementados con el paquete matemático MatLab para ver los resultados obtenidos por los programas en C. Estos programas permiten visualizar elementos tales como polígonos generados, polígonos de visibilidad, unión de polígonos de visibilidad, superficie de áreas, etc. También permiten dibujar polígonos y nubes de puntos interiores al polígono. Para reducir el código expuesto se incluyen sólamente las subrutinas de visualización básicas. De esta manera para visualizar las soluciones de cada heurísticas, aunque realmente son programas distintos, pues los ficheros de lectura son distintos, sólo incluímos la visualización de la solución de una de las heurísticas para cada problema. Además, algunos códigos de visualización que se consideran auxiliares tampoco se incluyen.

Dividimos los programas en cuatro bloques:

- Programas de visualización general
- Programas relacionados con el problema MaxA-p-Pv1(P).

- Programas para visualizar resultados de los problema MaxA-p-Pvk(P, k) y MinN-p-Pvk(P).
- Programas de visualización para MaxA-p-Vor(N).

Programas de visualización general

► Incluimos un pequeño código que permite visualizar el polígono generado con nuestro generador aleatorio RPG.

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\poligono.txt');
nc=size(A,1);
x=[];y=[];
for i=1:nc
 x(i)=A(i,1);
 y(i)=A(i,2);
end
fill(x,y,'y'),title('Poligono Generado');
hold on;

grid;
```

► El siguiente código se utilizará para generar un polígono y una nube mediante el ratón. De esta forma podemos estudiar el comportamiento de polígonos que no han sido generados con RPG. El código para generar o pintar un polígono es el siguiente:

```
clf;axis([0,1,0,1]);grid
title('Introducir Poligono en sentido positivo FIN: Boton derecho raton');
i=1;bot=[];z=[];t=[];
[fi,texto]=fopen('c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\poligono.txt','w');
[x,y,bot]=ginput(1);
while(bot(1)==1)
 fprintf(fi,'%f %f\n',x(1),y(1));
 z(i)=x(1);
 t(i)=y(1);
 line([z],[t]);
 i=i+1;
 [x,y,bot]=ginput(1);
end
st=fclose(fi);
axis(axis);
fill(z,t,'c'),title('Poligono Generado');
hold on;

grid;
```

► La siguiente subrutina permite generar una nube de puntos interior a un polígono:

```
pintarpoligono;
title('Introducir Nube FIN: Boton derecho raton');
i=1;bot=[];z=[];t=[];
[fi,texto]=fopen('c:\\santiago\\discos\\doctorado\\PVGC\\Pruebas_Heur_P1\\Matlab\\nube.txt','w');
[x,y,bot]=ginput(1);
while(bot(1)==1)
 fprintf(fi,'%f %f\n',x(1),y(1));
 z(i)=x(1);
 t(i)=y(1);
 plot(x(1),y(1),'.');
 i=i+1;
 [x,y,bot]=ginput(1);
end
st=fclose(fi);
axis(axis);
title('Nube Generada');

hold on;
```

- El siguiente programa nos permite visualizar una triangulación conseguida según un algoritmo de triangulación Scan de Graham [68]. Esta triangulación será necesaria para calcular el área del polígono P y así, el porcentaje de área iluminada por un conjunto de k puntos interiores a él, es decir, nos permitirá calcular el valor de la función objetivo para el problema MaxA-p-Pvk(P, k).

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\triangulacion.txt');
nc=size(A,1);
x=[];y=[];
i=1;
while(i<nc)
 x=[];y=[];
 for j=1:3
  x(j)=A(i,1);
  y(j)=A(i,2);
  i=i+1;
 end
 if(i==nc)
  fill(x,y,'c'),grid,title(['Triangulacion Generada. Area Total: ',num2str(A(i,1))]);
 else
  fill(x,y,'c'),grid,title('Triangulacion Generada: ');
 hold on;
 end
end
for j=1:3
 x(j)=A(i,1);
 y(j)=A(i,2);
end
fill(x,y,'c'),title(['Triangulacion Generada. Area Total: ',num2str(A(nc,1))]);
grid;
```

Programas relacionados con el problema MaxA-p-Pv1(P)

- El siguiente código permite visualizar el polígono de visibilidad solución de cualquiera de las heurísticas diseñadas para MaxA-p-Pv1(P). Se muestra también el área iluminada y el punto solución:

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\polvisidet.txt');
nc=size(A,1);
x=[];y=[];
for i=1:nc-2
 x(i)=A(i,1);
 y(i)=A(i,2);
end
fill(x,y,'c'),grid,title('Poligono de Visibilidad Generado');
hold on;
i=i+1;
plot(A(i,1),A(i,2),'*');
i=i+1;
title(['Poligono de Visibilidad Generado Area: ',num2str(A(i,1)),' Porc: ',num2str(A(i,2)),'%']);
hold on;
grid;
```

- Para visualizar la descomposición $\mathcal{S}$ de un polígono es regiones de visibilidad se ha implementado el siguiente código:

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\nube.txt');
nf=size(A,1);
for i=1:nf
 plot(A(i,1),A(i,2),'.');
 if A(i,3)==1
  j=i+1;
 while (j<=nf)&(A(j,3)==2)
```

```
  line([A(i,1),A(j,1)],[A(i,2),A(j,2)]);
  j=j+1;
 end
 i=j;
 end
 hold on;
end
hold on;

grid;
```

- El siguiente programa permite visualizar por pantalla la superficie de áreas construida para un determinado polígono P:

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\superficie.txt');
nc=size(A);
x=0:0.005:1;
y=0:0.005:1;
[X,Y]=meshgrid(x,y);
Z=[];
for i=1:nc
 for j=1:nc
  Z(i,j)=A(i,j)+0.5;
 end
end
surfl(X,Y,Z);
shading interp;
colormap gray;
hold on;contour(X,Y,Z);

title('Superficie de Areas');
```

- Finalmente incluimos un código que permite visualizar el conjunto de puntos visitas por una heurísticas en todas sus iteraciones.

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\rastro.txt');
nf=size(A,1);
hold on;
for i=2:nf
 plot(A(i,1),A(i,2));
end

hold off;
```

Programas para los problemas MaxA-p-Pvk(P,k) y MinN-p-Pvk(P)

- El siguiente código nos permite visualizar el polígono unión de k polígonos de visibilidad, (debe recordarse que puede ser un polígono con varias componentes conexas con agujeros). Este programa será necesario para visualizar el resultado de los métodos diseñados para los problemas MaxA-p-Pvk(P,k) y MinN-p-Pvk(P).

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P1\Matlab\polunionagujeros.txt');

             %Primero pintamos poligonos principales
nc=size(A,1);
x=[];y=[];
j=1;
for i=1:nc-1
 if(A(i,1)~=2.0)
  x(j)=A(i,1);
  y(j)=A(i,2);
  j=j+1;
 else
  if(A(i,2)==2.0)
   fill(x,y,'c'),title('Poligono Generado');
  end
```

```
  x=[];y=[];
  j=1;
  hold on;
 end
end
              %Ahora pintamos los agujeros

nc=size(A,1);
x=[];y=[];
j=1;
for i=1:nc-1
 if(A(i,1)~=2.0)
  x(j)=A(i,1);
  y(j)=A(i,2);
  j=j+1;
 else
  if(A(i,2)==1.0)
   fill(x,y,'w'),title('Poligono Generado');
 end
 x=[];y=[];
 j=1;
 hold on;
 end
end
hold on;
i=i+1;
title(['Poligono de Visibilidad Generado Area: ',num2str(A(i,1)),' Porc: ',num2str(A(i,2)),'%']);

grid;
```

Programas de visualización para MaxA-p-Vor(N)

- El siguiente código dibuja el *diagrama de Voronoi* de una nube de puntos, construido para visualizar las soluciones de las heurísticas diseñadas en el Capítulo 9 para el problema MaxA-p-Vor(N).

```
A=load('c:\santiago\discos\doctorado\PVGC\Pruebas_Heur_P2\Matlab\voronoi.txt');

              %Primero pintamos las regiones y a continuacion los puntos
nc=size(A,1);
x=[];y=[];
j=1;
for i=1:nc-1
 if(A(i,1)~=2.0)
  x(j)=A(i,1);
  y(j)=A(i,2);
  j=j+1;
 else
  if(A(i,2)==2.0)
   if(i==nc-3)
    fill(x,y,'w'),title('Diagrama de Voronoi Generado');
   else
    fill(x,y,'g'),title('Diagrama de Voronoi Generado');
   end
  end
  x=[];y=[];
  j=1;
  hold on;
 end
end
              %Ahora la nube de puntos

nc=size(A,1);
x=[];y=[];
j=1;
for i=1:nc-1
 if(A(i,1)~=2.0)
  x(j)=A(i,1);
  y(j)=A(i,2);
  j=j+1;
 else
  if(A(i,2)==3.0)
   if(i==nc-1)
    plot(x,y,'.');
   else
    plot(x,y,'.');
```

```
   end
  end
  x=[];y=[];
  j=1;
 end
end
hold on;
i=i+1;
title([‘Diagrama de Voronoi Area: ’,num2str(A(i,1)),‘ Porc: ’,num2str(A(i,2)),‘%’]);
```

Bibliografía

[1] Abellanas, M.; Hernández, G.; Lodares, D.: *"Kernels and Depth of a Convex Polygon"*, Sixth SIAM Conference on Discrete Mathematics, Vancouver, (Canadá), 1992.

[2] Ahn, H-K.;Cheng, S-W.; Cheong, O.; Golin, M.; Oostrum, R.: *"Competitive facility location along a highway"*, Proc. 7th Annu. Int. Conf. (COCOON 2001), Lectures Notes Comput. Sci.(2108):237-246, 2001.

[3] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: *"The Design and Analysis of Computer Algorithms"*, Addison Wesley, 1974.

[4] Atiyah, M.; Sutcliffe, P.: *"Polyhedra in Physics, Chemistry and Geometry"*,arXiv: math-ph/0303071 v1, to appear in the Milan Journal of Mathematics.

[5] Avis, D.; Toussaint, G.T.: *"An Efficient Algorithm for Decomposing a Polygon into Star-Shaped Pieces",* Pattern Recognition, 13, pp. 295-298, 1981.

[6] Auer T,; Held M.: *"Heuristics for the Generation of Random Polygons"*, Proc. 8th Cand. Conf. Comput. Geom., pp. 38-44, Ottawa, Canada, Aug 1996. Carleton University Press.

[7] Back, T.: *"Evolutionary Algorithms in Theory and Practice"*, Oxford Press, 1996.

[8] Belleville, P.; Bose, P.; Czyzowicz, J.; Urrutia, J.; Zaks, J.: *"K-guarding polygons in the plane"*, Proceedings Sixth Canadian Conference on Computational Geometry, (Saskatoon, Canada, 1994), pp. 381-386.

[9] García, B.: *"Uso del Sistema de Colonia de Hormigas para Optimizar Circuitos Lógicos Combinatorios",* Maestría, Universidad de Veracruz, 2001.

[10] Bentley, J. L.; Ottmann T.A.:"*Algorithms for reporting and counting geometric intersections"*, IEEE Transactions on Computers 28, pp. 643-647, 1979.

[11] Berger, M.: *"Geometry"*, Springer-Verlag, 1987.

[12] Berg de, M.; Kreveld van, M.; Overmars, M.; Schwarzkopf, O.: *"Computational Geometry. Algorithms and Applications"*, Springer, 1997.

[13] Bjorling-Sachs, I.; Souvaine, D.: *"ATight Bound for Guarding General Polygons with Holes"*, Tech. Report LCSR-TR-165, Dept. of Comp. Science, Rutgers University, 1991.

[14] Bondy, J. A.; Murty, U.S.R.: *"Graph theory with applications"*, Elsevier Science, New York, 1976.

[15] Bose P.; Lubiw A.; Munro J.I.: *"Efficient visibility queries in simple polygons",* Proceedings of the 4th Canadian Conference on Computational Geometry, St. Johns, Nfld, pp. 23-28, 1992.

[16] Box, G. E. P.; Muller, M. E.: *"A Note on the Generation of Random Normal Deviates*", Ann. Math. Stat. 28, pp. 610-611, 1958.

[17] Brassard, G.; Bratley, P.: *"Fundamentos de Algoritmia"*, Prentice Hall 2000.

[18] Brodal, G.S.; Jacob, R.: *"Dynamic Planar Convex Hull",* Preprint sumitted to Elsevier Science, 2003.

[19] Bullnheimer, B.; Hartl, R.F.: *"An Improved Ant System Algorithm for the Vehicle Routing Problem"*, Sixth Viennese workshop on Optimal Control, Dynamic Games, Nonlinear Dynamics and Adaptive Systems, Vienna (Austria), May 21-23, 1997, también en: *"Annals of Operations Research"*, (Dawid, Feichtinger and Hartl (eds.)): Nonlinear Economic Dynamics and Control, 1999.

[20] Chazelle, B.; Edelbrunner, H.: *"An optimal algorithm for intersecting line segments in the plane"*, Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science, pp. 217-225, 1988.

[21] Chazelle, B.: *"Triangulating a Simple Polygon in Linear Time"*, Discrete and Computational Geometry, 6, pp. 485-524, 1991.

[22] Cheong, O.; Har-Peled, S.; Linial, N.; Matoušek, J.: *"The one-round Voronoi game"*, Proc. 18th Annu. ACM Symp. on Computational Geometry, 2002.

[23] Cheong, O.; Efrat, A.; Har-Peled, S.: *"On Finding a Guard that Sees Most and a Shop that Sells Most",* ACM-SIAM Symposium on Discrete Algorithms (SODA), New Orleans, LA, January 11-13, 2004.

[24] Chvátal, V.: *"A Combinatorial Theorem in Plane Geometry"*, Journal of Combinatorial Theory, Serie B, 18, pp. 39-41, 1975.

[25] Clacerol, M.: *"Geometric Problems on Computational Morphology"*, Ph. D. Thesis in preparation.

[26] Coll N.; Hurtado F.; Sellarés J. A.: *"Approximating planar subdivisions. Aplications to Voronoi diagrams",* X Encuentros de Geometría Computacional, Sevilla, 2003.

[27] Cook, S.A.: "*The complexity of theorem-proving procedures"*. En Association for Computing Machinery, editor, Proc. 3rd Ann. ACM Symp. on Theory of Computing, pp. 151-158. New York, 1971.

[28] Davis, L.; Benedikt, M.: *"Computational Models of Space: Isovist and Isovist Fields"*, Computer and Image Processing, 11, pp. 49-72, 1979.

[29] Davis, L.: *"Handbook of Genetic Algorithms"*, Van Nostrand Reinhold, 1991.

[30] Dehne, F.; Klein, R.; Seidel, R.: *"Maximizing a Voronoi Region: The Convex Case"*,

[31] Díaz, A.; Glover, F.; Ghaziri, H.M.; González, J.L.; Laguna, M.; Moscato, P.; Tseng, F.T.: *"Optimización Heurística y Redes Neuronales"*, Ed. Paraninfo, 1996.

[32] Dorigo, M.: "*Behavoir of Real Ants",* http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html, IRIDIA, Université Libre de Bruxelles, Belgium.

[33] Dorigo, M.: *"Optimizacion, Learning and Natural Algorithms"*, Ph.D.Thesis, Politecnico di Milano, Italy, 1992.

[34] Dorigo, M.; Maniezzo, V.; Colorni A.: *"The Ant System: Optimization by a Colony of Cooperating Agents"*, IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1): pp. 29-41, 1996.

[35] Dowsland, K.A.: *"Simulated Annealing"*, en Modern Heuristic Techniques for Combinatorial Problems. Ed. C. R. Reeves, Blackwell Scientific Pub., Oxford, 1993.

[36] Edelsbrunner, H.; O'Rourke, J.; Welzl, E.: *"Stationing Guards in Rectilinear Art Galleries"*, Computer Vision, Graphics and Image Processing, 27, pp. 167-176, 1984.

[37] Efrat, A.; Har-Peled, S.: *"Locating guards in art galleries",* 2nd IFIP Internat. Conf. Theo. Comp. Sci., pp. 181-192, 2002.

[38] Epstein, P.: *"Generating Geometric Objects at Random"*, Master's thesis, CS Dept., Carleton University, Ottawa K1S5B6, Canada, 1992.

[39] ElGindy, H.; Avis, D.: *"A linear algorithm for computing the visibility polygon from a point",* Journal of Algorithms, 2, pp. 186-197, 1981.

[40] Estévez Valencia P.: *"Optimización Mediante Algoritmos Genéticos"*, Anales del Instituto de Ingenieros de Chile, Agosto 97, pp. 83-92.

[41] Even, S.; Itai, A.; Shamir, A.: *"On the complexity of timetable and multicommodity flow problems", SICOMP,* 5(4):691:703, 1976.

[42] Fisk, F.: *"A Short Proof of Chvátal's Watchman Theorem"*, Journal of Combinatorial Theory, Serie B, 24, pp. 374, 1978.

[43] Fogel, D.: *"Evolutionay Computation"*, IEEE Press, 1995.

[44] Foley, Van Dam , Feiner, Hughes, Phillips, "*Introduction to Computer Graphics"*, Ed. 1^a, Addison-Wesley, 1994.

[45] Garey, M.R.; Johnson, D.S.: "*Computers and intractability. An guide to the theory of NP-completeness"*, W.H. Freeman and Company 1979

[46] García J.: *"Problemas Algorítmicos-Combinatorios de Visibilidad ",* Tesis Doctoral, UPM, 1995.

[47] Gelatt, C.D.; Kirkpatrick, S.; Vecchi, M.P.: *"Optimazation by simulated annealig"*, Science, 220: pp. 671-680, 1983.

[48] Geman, S.; Geman, D.: *"Stochastic relaxation, Gibbs distribution, and the Bayesian restoration of images",* IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-6: pp. 721-741, 1984.

[49] Ghosh, S.K.: *"Approximation algorithms for Art Gallery Problems"*, Proceedings of the Canadian Information Processing Society Congress, 1987.

[50] Ghosh, S.K.: *"Computing a viewpoint of a set of points inside a polygon"*, Lecture Notes in Computer Science, No. 338, pp. 96-104, Springer Verlag, 1988.

[51] Ghosh, S. K.; Bhattacharya, A.; Sarkar, S.: *"Exploring an unknown polygonal environment with bounded visibility"*, Proceedings ICCS 2001: International Conference Computational Science, San Francisco, CA, USA, May 28-30, 2001, LNCS vol. 2073.

[52] Glover, F.: *"Tabu Search: A Tutorial",* Center of Applied Artificial Intelligence, University of Colorado, USA.

[53] Glover, F.; Kochenberger, G.: *"State of the Art Handbook in Metaheuristics"*, Kluwer 2002.

[54] Glover, F.; Laguna M.: "*Tabu Search",* Kluwer Academic Publishers 1997.

[55] Goldberg, D.E.: *"Genetic Algorithms in Search, Optimization and Machine Learning"*, Addison-Wesley, 1995.

[56] Hearn, Baker : "*Gráfica por Computadora*", Ed 2^a, Prentice-Hall, 1995, (corresponde a Ed 2^o, inglés, 1994).

[57] Hochbaum, D.; Pathria, A.: *"Analysis of the greedy approach in covering problems"*, Naval Research Quarterly, 45: pp. 615-627, 1998.

[58] Hoffmann, F.: *"On the Rectilinear Art Gallery Problem"*, Proc. International Colloquium on Automata, Languages and Programming, 90. pp. 717-728, 1990.

[59] Hoffmann, F.; Kaufmann, M.; Kriegel, K.: *"The Art Gallery Theorems for Polygons with Holes"*, Proc. Foundations of Comp. Science, 91, pp. 39-48, 1991.

[60] Holland, J.H.: *"Adaptation in Natural and Artificial Systems"*, University of Michigan Press, 1975.

[61] Ingber, L.: *"Very fast simulated re-annealing",* Mathl. Comput. Modelling, 12(8): pp. 967-973, 1989.

[62] Jesus de, M.C.: *"Minimal Steiner Trees Approximations using Genetic Algorithms"*, PhD. thesis, University of Sevilla, Departament of Applied Mathematics I, 2000.

[63] Joe, B.; Simpsom, R.B.: *"Corrections to Lee's visibility polygon algorithm"*, BIT, 27, pp. 458-473, 1986.

[64] Kaelbling, L.P.; Littman, M.L.: *"Reinforcement Learning: A Survey",* Journal of Artifical Intelligence 4: pp. 237-285, 1996.

[65] Kahn, J.; Klawe, M.; Kleitman, D.: *"Traditional Galleries Require Fewer Watch-men"*, SIAM J. Algebraic and Discrete Methods, 4, pp. 194-206, 1983.

[66] Karp, R.M.: *"Reducibility among combinatorial problems",* R.E. Miller y J.W. Thatcher, editors, Complexity of Computer Computations, pp. 85-103. Plenum Press, New York, 1972.

[67] Ke, Y.; O'Rourke, J.: *"Computing the Kernel of a Point Set in a Polygon",* Proceedings WADS 89, Ottawa, Canada, August 17-19.

[68] Kong, X.; Everett, H.; Toussaint, G.T.: *"The Graham scan triangulates simples polygons"*, Pattern Recognition Letters, vol. 11, November 1990, pp. 713-716.

[69] Koza, J.R.: *"Genetic Programming"*, MIT Press, 1992.

[70] Kuc, R.; Siegel, M., *"Efficient Representation of Refecting Structures for a Sonar Navigation Model"*, Proceedings IEEE International Conference on Robotics and Automation, 1987.

[71] Lee D.T.; Lin A.K.: *"Computational complexity of art gallery problem"*, IEEE Trans. Info. Th. IT-32, pp. 415-421, 1979.

[72] Lee, D.T.: *"Visibility of a Simple Polygon"*, Computer Vision, Graphics and Image Processing, 22, pp. 207-221, 1983.

[73] Lozano-Pérez, T.; Wesley, M.A.: *"An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles"*, Comm. ACM, 22, pp. 560-570, 1979.

[74] Michalewicz, Z.: *"Genetic Algorithms+Data Structures=Evolution Programs"*, Springer-Verlag, Second Edition, 1994.

[75] Munro, J.; Overmars, M.: *"Variations on Visibility"*, Proc. 3rd ACM Symposium on Computational Geometry, 1987.

[76] Nemhauser, G.L.; Wolsey, L.A.: *"Integer and Combinatorial Optimization"*, Wiley, 1988.

[77] Ntafos, S.: *"Watchman routes under limited visibility ",* Comput. Geom. Theory Appl., vol.1, n.3, pp.149-170, 1992.

[78] Ntafos, S. C.; Tsoukalas, M. Z.: "*Optimum placement of guards",* Info. Sciences, 76: pp. 141-150, 1994.

[79] Okabe, A.; Boots, B.; Sugihara, K.; Chiu, S. N.: *"Spacial Tessellations: Concepts and Applications of Voronoi Diagrams"*, John Wiley & Sons, Chichester, UK, 2000.

[80] O'Rourke, J.: *"Art Gallery Theorems and Algorithms"*, Osford University Press, 1987.

[81] O'Rourke, J.; Virmani, M.: *"Generating Random Polygons"*, Technical Report 011, CS Dept., Smith College, Northampton, MA 01063, July 1991.

[82] O'Rourke, J.: *"Computational Geometry in C "*, Cambridge University Press, 1995.

[83] Otfried Cheong and René van Oostrum, *"The Visibility Region of Points in a Simple Polygon",* Proc. 11th Canad. Conf. Comput. Geom., (1999), pp. 87-90.

[84] Pippenger, N.; Fischer, M.J.: *"Relations Among Complexity Measures", JACM,* Vol 26, No. 2, 1979, pp. 361-381.

[85] Reeves, C.R.: *"Modern Heuristic Techniques for Combinatorial Problems"*, McGraw-Hill, 1995.

[86] Feo T.A.; Resende, M.G.C.: *"Greedy Randomize Adaptive Search Procedures"*, Journal of Global Optimizacion 6: pp. 109-133, 1995.

[87] Reyes Columé, P.: *"Problemas de Etiquetado: Complejidad Computacional"*, Tesis Doctoral, Universidad de Sevilla, Departamento de Matemática Aplicada I, 2002.

[88] Sack, J.-R.; Urrutia, J., *"Handbook of Computational Geometry",* Elsevier, First Edition, 2000.

[89] Savage, J.E.: *"The Complexity of Computing",* Krieger, 1976.

[90] Shermer, T.: *"Visibility Properties of Polygons"*, Tesis Doctoral, McGill University, 1989.

[91] Shermer, T.: *"Recent Results in Art Galleries"*, Proceedings of the IEEE, 1992.

[92] Schuchart, D.; Hecker H.: "*Two NP-hard problems for ortho-polygons"*, Math. Logiv Quart. 41, pp. 261-267, 1995.

[93] Schuierer, S.; Rawlins, G.; Wood, D.: "Towards a General Theory of Visibility", Proc. 2nd Canadian Conference on Computational Geometry, 1990.

[94] Sorkin, G.: *"Theory and Practice of Simulated Annealing on Special Energy Landscapes"*, Ph. D. thesis, University of California at Berkely, Berkeley, CA, USA, 1992.

[95] Sung-Ho Kim; Jung-Heum Park; Seung-Hak Choi; Sung Yong Shin; Kyung-Yong Chwa, *"An optimal algorithm for finding the edge visibility polygon under limited visibility",* Information Processing Letters, Vol. 53, Issue 6, pp. 359-365, 1995.

[96] Szu, H. H.; Hartley, R. L.: "*Fast simulated annealig"*, Physic Letters A, 122: pp. 157-162, 1987.

[97] Toussaint, G.T.: *"Computing geodesic properties inside a simple polygon",* Invited Paper, Revue D'Intelligence Artificielle, vol. 3, No. 2, 1989, pp. 9-42.

[98] Tukey, J.W.: *"Mathematics and the picturing of data"*, Proceedings of the International Congress of Mathematicians, 2, (1975), pp. 523-531.

[99] Urrutia, J.: *"Art Gallery and Illumination Problems "* en "Handbook on Computational Geometry", Elsevier (J. R. Sack and J. Urrutia ed.), 1999.

[100] Wood, D.; Yamamoto, P.: *"Dent and Staircase Visibility"*, Proc. 5th Canadian Conference on Computational Geometry, pp. 297-302, 1993.

[101] Yachida, M.: *"3-D Data Adquisition by Multiple Views"*, Robotic Research: The Third International Symposium, MIT Press, 1986.

Índice Alfabético