Estudio e implementación de metaheurísticas para solucionar el problema de la selección de la solución deseada

E. Yeguas[1], M.V. Luzón[2], E. Barreiro[3], R. Joan-Arinyo[4]

[1]Escuela Politécnica Superior
Universidad de Córdoba
Campus Rabanales. Marie Curie (C3 - Anexo). 14071 Córdoba
eyeguas@uco.es

[2]E.T.S. Ingenierías Informática y Telecomunicación
Universidad de Granada
C/ Periodista Daniel Saucedo Aranda s/n. 18071 Granada
luzon@ugr.es

[3]Escuela Superior de Ingeniería Informática
Universidad de Vigo
Av. As Lagoas s/n, E-32004 Ourense
enrique@uvigo.es

[4]Grup d'Informàtica a l'Enginyeria
E.T.S. d'Enginyeria Industrial de Barcelona
Universitat Politècnica de Catalunya
Av Diagonal 647, 8a. 08028 Barcelona
robert@lsi.upc.edu

Resumen

Trabajos anteriores han demostrado la viabilidad de aplicar algunas Metaheurísticas al problema de la selección de la solución deseada. Por tanto, se ha decidido realizar un estudio de distintas metaheurísticas con el fin de aplicarlas al citado problema. Esto podría suponer un gran avance en la aplicación del CAD.
En este trabajo se realizará una descripción del marco de aparición del concepto Metaheurística. Prosigue con la formalización y descripción de las características asociadas a dicho concepto. Se establecen distintas clasificaciones basadas en las filosofías seguidas por los distintos algoritmos dentro del marco Metaheurística. Basándonos en una de ellas vamos describiendo cada una de las Metaheurísticas existentes, proponiendo implementaciones para los distintos algoritmos más representativos y analizando sus orígenes, evolución, variantes, estructura, eficacia y eficiencia teórica previa y potencialidad en general.

1. Introducción

Una de las ventajas más relevantes en el Diseño Asistido por Computador (CAD) ha sido el diseño basado en restricciones. El usuario describe un objeto mediante la definición de un conjunto de elementos geométricos tales como puntos, segmentos de línea y segmentos

circulares, y un conjunto de restricciones geométricas relativas a dichos elementos. La principal tarea de los sistemas CAD es comprobar si el conjunto de restricciones geométricas define de forma precisa el objeto y, en ese caso, determinar la posición y orientación de los elementos geométricos.

Cuando existe una solución, el usuario espera que el "*solver*[1]*"* le proporcione una determinada instancia y no cualquier instancia del espacio de soluciones. El problema de generar automáticamente la instancia esperada es un problema abierto y se conoce como el **problema de la selección de la solución deseada**. Existen diversas cuestiones que hacen que el problema tenga una dificultad elevada. De gran importancia resulta el hecho de que, hasta el momento, no se ha dado una definición aceptable del propio problema, es decir, hasta ahora no se sabe qué es aquello que el usuario espera y, por tanto, no se sabe cómo caracterizar la solución deseada.

Encontramos un pequeño acercamiento esperanzador a la solución del problema a partir de la definición por parte del usuario de una serie de predicados o restricciones adicionales que desea que cumpla la solución. A partir de la definición de tales predicados y mediante un proceso de transformación conveniente nos llevamos dicho problema al terreno de la optimización combinatoria. El problema se reformula y abre el horizonte a su solución mediante otras vías que, partiendo del consumo considerable de tiempo que supone la evaluación de una solución, tratarán que los algoritmos evalúen el mínimo número de soluciones posibles.

Entre los algoritmos aproximados se establecían típicamente los métodos constructivos y los métodos basados en búsqueda local. Sin embargo, en los últimos años (10-15 años) ha nacido un nuevo tipo de algoritmos aproximados que básicamente trata de combinar métodos heurísticos básicos en marcos de trabajo de alto nivel redirigidos a la exploración del espacio de búsqueda. Este tipo de métodos son hoy en día conocidos como *Metaheurísticas*. Las características fundamentales que se asocian a este tipo de métodos de búsqueda y que los hacen tan potentes son las siguientes:

- Son estrategias que guían el proceso de búsqueda.
- Su objetivo es explorar eficientemente el espacio de búsqueda para encontrar soluciones suboptimales.
- Las técnicas que las constituyen van desde la búsqueda local simple hasta procesos de aprendizaje complejo.
- Se tratan de algoritmos aproximados y no determinísticos.
- Incorporan mecanismos para evitar quedarse atrapados en áreas apetecibles del espacio de búsqueda.
- Los conceptos básicos asociados a las distintas Metaheurísticas permiten una descripción de nivel abstracto.
- No son específicas del problema.
- Hacen uso de conocimiento específico del dominio y/o experiencia anterior para sesgar y dirigir la búsqueda.

La aplicación de las Metaheurísticas a un problema con tanta necesidad de cómputo y con tanta necesidad de rapidez por parte del usuario como el de la selección de la solución deseada podría suponer un avance en el camino a la obtención de buenas soluciones a casos medianos del problema en poco tiempo, concretamente en el tiempo deseable para la aplicación desde un punto de vista práctico a los distintos terrenos profesionales.

---

[1] ***solver:*** nombre con que se designa comúnmente al software que se utiliza para resolver el *problema de resolución de restricciones geométricas*.

En este trabajo vamos a realizar un estudio de algunas Metaheurísticas con la intención de aplicarlas al problema de la selección de la solución deseada.

Comenzaremos con la descripción del marco de aparición del concepto metaheurística. Proseguiremos con la formalización y descripción de las características asociadas a dicho concepto. Se establecen distintas clasificaciones basadas en las filosofías seguidas por los distintos algoritmos dentro del marco metaheurística. Basándonos en una de ellas vamos describiendo cada una de las metaheurísticas existentes, proponiendo implementaciones para los distintos algoritmos más representativos y analizando sus orígenes, evolución, variantes, estructura, eficacia y eficiencia teórica previa y potencialidad en general.

A continuación realizaremos una somera exposición de en qué consisten, capacidad y posibilidades de las Metaheurísticas para resolver problemas de optimización combinatoria. Trataremos con una clasificación importante de las Metaheurísticas donde los algoritmos encuadrados tendrán diferencias con respecto a cómo abordan un problema, pero donde tendremos muchas semejanzas en sus estrategias y conceptos usados que permiten agruparlos desde un mismo punto de vista y bajo una misma bandera.

En la opinión de muchos autores los conceptos más importantes que se usan en las Metaheurísticas son los de *intensificación* y *diversificación*, conceptos contrarios y complementarios a la vez. Las Metaheurísticas que destacaremos serán comparadas desde el punto de vista de estos dos conceptos fundamentales, que también antes describiremos.

Concluiremos el estudio de las distintas Metaheurísticas subrayando puntos fuertes y débiles de las distintas aproximaciones con el objeto de desarrollar posibles hibridaciones combinando conceptos de cada una de las Metaheurísticas.

Finalmente, una vez enumeradas las distintas metaheurísticas, concretaremos cómo se ha resuelto la estructura y componentes genéricos de cada uno de sus algoritmos correspondientes, y realizaremos su implementación. En primer lugar comenzaremos indicando los aspectos aplicables a cualquier problema para cada algoritmo dentro de cada *Metaheurística* y posteriormente veremos aquellos que han necesitado ser adaptados al problema particular que resolvemos.

2. Metaheurísticas

Muchos problemas de optimización, de importancia tanto desde un punto de vista teórico como punto de vista práctico, tienen su solución en la elección de una "mejor" configuración de un conjunto de parámetros para lograr determinadas metas. Estos parecen dividirse de manera natural en dos categorías: aquellos para los que las soluciones son codificadas con variables continuas, y aquellos cuyas soluciones se codifican con variables discretas. Entre los últimos encontramos un tipo de problemas llamados problemas de optimización combinatoria. Según [PS82], en los problemas de optimización combinatoria, buscamos un objetivo dentro de un conjunto finito (o posiblemente contablemente infinito). Este objetivo es típicamente un entero, un subconjunto, una permutación, o una estructura de grafo.

Definición 2.1. Un *problema de Optimización Combinatoria* $P = (S,f)$ puede ser definido por:

- un conjunto de variables $X = \{x_1,..., x_n\}$;
- dominios variables $D_1,..., D_n$;
- restricciones entre variables;
- una función objetivo a ser minimizada o maximizada f: $D_1 \times ... \times D_n \rightarrow \Re^+$;

El conjunto de todas las posibles asignaciones o soluciones factibles es

$S = \{ s = \{(x_1, v_1),..., (x_n, v_n)\} \mid v_i \in D_i, i = 1,...,n \wedge s$ satisface todas las restricciones $\}$

S es llamado normalmente espacio de búsqueda (o soluciones), ya que cada elemento del conjunto puede ser visto como una solución candidata.

Para resolver un problema de optimización combinatoria se debe encontrar una solución $s \in S$ con valor mínimo de función objetivo, esto es, $f(s^*) \leq f(s)$ $\forall s \in S$ (para maximizar tendríamos $f(s^*) \geq f(s)$ $\forall s \in S$). s^* es llamada una solución globalmente óptima de (S,f) y el conjunto $S^* \subseteq S$ es llamado conjunto de soluciones globalmente óptimas.

Ejemplos de problemas de optimización combinatoria es el que nos ocupa: el problema de la selección de la solución deseada dentro de un problema de satisfacción de restricciones geométricas, y otros muy conocidos como el del viajante de comercio (TSP), el problema de asignación cuadrática (QAP), problemas de satisfacción de restricciones (CSPs) y problemas de planificación (scheduling). Debido a la importancia práctica de los problemas de optimización combinatoria, muchos algoritmos han sido desarrollados para su solución. Estos algoritmos pueden ser clasificados como algoritmos completos (exactos) o algoritmos aproximados. Los algoritmos exactos tienen garantizado encontrar para cada instancia finita de un problema de optimización combinatoria en tiempo limitado (ver [PS82], [NW88]). Sin embargo, para muchos problemas de optimización combinatoria que son *NP*-duros [GJ79], los métodos exactos necesitan un tiempo de computación exponencial en el peor de los casos e incluso para instancias del problema pequeñas estos algoritmos podrían tener un tiempo de ejecución demasiado alto para propósitos prácticos. Por lo tanto, el uso de métodos aproximados para resolver problemas de optimización combinatoria ha conseguido muchos adeptos en los últimos treinta años. En los métodos aproximados sacrificamos la garantía de encontrar soluciones óptimas por la gran posibilidad de encontrar buenas soluciones en una cantidad de tiempo significativamente reducida.

Entre los métodos aproximados básicos distinguimos usualmente entre métodos constructivos y métodos de búsqueda local. Los algoritmos constructivos generan soluciones desde la nada añadiendo componente a componente a una solución parcial inicialmente vacía para construir en el último paso la solución completa. Son típicamente los métodos aproximados más rápidos, pero sin embargo, devuelven a menudo soluciones de inferior calidad cuando son comparados con algoritmos de búsqueda local. Los algoritmos de búsqueda local parten de una solución inicial e iterativamente tratan de reemplazar la solución actual por una mejor solución en un vecindario de la solución actual definido apropiadamente. Dicho vecindario se define formalmente como sigue:

Definición 2.2. Una *estructura de vecindario* es una función $N : S \rightarrow 2^S$ que asigna a cada $s \in S$ un conjunto de vecinos $N(s) \subseteq S$. $N(s)$ es también llamado el vecindario de s.

Con la introducción de lo que es una estructura de vecindario podemos también definir el concepto de soluciones localmente mínimas o máximas.

Definición 2.3. Una *solución localmente mínima* (o *mínimo local*) con respecto a una estructura de vecindario N es una solución $\hat{s}$ tal que $\forall s \in N(\hat{s}) : f(\hat{s}) \leq f(s)$ (para que tuviésemos una solución localmente máxima o máximo local debería ocurrir que $\forall s \in N(\hat{s}) : f(\hat{s}) \geq f(s)$). Llamamos $\hat{s}$ una solución estrictamente mínima local si $\forall s \in N(\hat{s}) : f(\hat{s}) < f(s)$ (tendríamos una solución estrictamente máxima local si $\forall s \in N(\hat{s}) : f(\hat{s}) > f(s)$).

En los últimos 10 a 15 años un nuevo tipo de algoritmo aproximado ha emergido, el cual trata básicamente de combinar métodos heurísticos básicos en marcos de trabajo del más

alto nivel lanzándose a la exploración de un espacio de búsqueda. Este tipo de métodos son hoy en día comúnmente llamados *Metaheurísticas*. El término *metaheurística* deriva de la composición de dos palabras griegas. *Heurística* deriva del verbo *heuriskein* (*ευρισκειν*) que significa "encontrar", mientras que el sufijo "meta" significa "más allá de, en un nivel más alto".

Esta clase de algoritmos incluye los siguientes algoritmos y sus derivados:

- Optimización de colonias de hormigas (ACO).
- Computación evolutiva (EC) con su máximo representante: algoritmos genéticos (GA).
- Búsqueda local iterativa (ILS).
- Enfriamiento simulado (SA).
- Búsqueda tabú (TS).

Hay que tener en cuenta que las *Metaheurísticas* no se restringen únicamente a estos algoritmos sino que estos son los más representativos y comúnmente conocidos.

Hasta ahora, no hay una definición comúnmente aceptada del término *Metaheurística*. Ha sido justamente en los últimos años cuando algunos investigadores en el campo trataron de proponer una definición. A continuación resumimos algunas de ellas:

"Una metaheurística es formalmente definida como un proceso de generación iterativo que guía una heurística subordinada al combinar inteligentemente diferentes conceptos de exploración y explotación a lo largo de todo el espacio de búsqueda, aprendiendo estrategias que son usadas para estructurar la información con el objeto de encontrar soluciones eficientemente cercanas al óptimo". Esta cita ha sido tomada de una bibliografía acerca de *Metaheurísticas* de Osman y Laporte [OL96].

"*Una metaheurística es un proceso maestro iterativo que guía y modifica las operaciones de heurísticas subordinadas para producir soluciones de alta calidad de una manera eficiente. Puede manipular una solución simple completa (o incompleta) o una colección de soluciones en cada iteración. Las heurísticas subordinadas pueden ser procedimientos de alto o bajo nivel, o una simple búsqueda local, o un método de construcción".* Esta cita ha sido tomada de "Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization", de Voss, Martello, Osman y Roucairol [VMOR99].

"*Las metaheurísticas son típicamente estrategias de alto nivel las cuales guían y subyacen, más las heurísticas específicas del problema, para incrementar el rendimiento en la búsqueda. La principal meta es evitar las desventajas de la mejora iterativa y, en particular, el estancamiento al permitir a la búsqueda local escapar de los óptimos locales. Esto es logrado tanto permitiendo movimientos peores como generando nuevas soluciones iniciales para la búsqueda local de una forma más inteligente que la que proporciona justamente soluciones iniciales aleatorias. Muchos de los métodos pueden ser interpretados como la introducción de un sesgo tal que las soluciones de alta calidad son halladas rápidamente. Este sesgo puede ser de varias formas y puede ser denominado como sesgo de descenso (basado en la función objetivo), sesgo de memoria (basado en decisiones previas) o sesgo de experiencia (basado en rendimiento anterior). Muchas de las aproximaciones metaheurísticas dependen de decisiones probabilísticas hechas durante la búsqueda. Pero la principal diferencia con la búsqueda aleatoria pura es en los algoritmos metaheurísticos la aleatoriedad no es usada ciegamente sino de forma inteligentemente y sesgada".* Esta cita está tomada de la tesis PhD de Stützle [Stü99a].

Resumiendo, subrayamos las propiedades fundamentales que caracterizan a las *Metaheurísticas*:

- Las *Metaheurísticas* son estrategias que "guían" el proceso de búsqueda.
- La meta es explorar eficientemente el espacio de búsqueda para encontrar soluciones (sub)óptimas.
- Las técnicas que constituyen los algoritmos de *Metaheurísticas* van desde procedimientos simples de búsqueda local a complejos procesos de aprendizaje.
- Los algoritmos de *Metaheurísticas* son aproximados y no determinísticos.
- Incorporan mecanismos para evitar quedarse atrapados en áreas prometedoras cerradas del espacio de búsqueda.
- Los conceptos básicos de *Metaheurísticas* permiten una descripción de nivel abstracto.
- Las *Metaheurísticas* no son específicas del problema.
- Las *Metaheurísticas* hacen uso de conocimiento específico del dominio y / o experiencia de búsqueda (memoria) para sesgar la búsqueda.

Para enmarcar todo lo dicho podemos decir lo siguiente:

Las *Metaheurísticas* son conceptos de alto nivel para explorar espacios de búsqueda usando diferentes estrategias. Estas estrategias deberían ser escogidas de tal forma que existiera un equilibrio dinámico entre la explotación de la experiencia de búsqueda acumulada (que es comúnmente llamada *intensificación*) y la exploración del espacio de búsqueda (que es comúnmente denominada *intensificación*). Este equilibrio es necesario por una parte para rápidamente identificar regiones en el espacio de búsqueda con soluciones de alta calidad y por otra parte para no invertir demasiado tiempo en regiones del espacio de búsqueda que o bien han sido ya exploradas, o bien no proporcionan soluciones de alta calidad.

La estructura de las estrategias es altamente dependiente de la filosofía de la *Metaheurística* en sí. Posteriormente compararemos las estrategias usadas en las diferentes *Metaheurísticas*. Hay varias filosofías diferentes que aparecen en las *Metaheurísticas* existentes. Algunas de ellas pueden ser vistas como "extensiones" inteligentes de algoritmos de búsqueda local. La meta de esta clase de *Metaheurísticas* es la de escapar de mínimos locales para proseguir la exploración del espacio de búsqueda y moverse para encontrar otros esperanzadoramente mejores óptimos locales. Esto es por ejemplo verdadero para la Búsqueda Tabú, Búsqueda Local Iterativa, Búsqueda en Vecindario Variable, GRASP y Enfriamiento Simulado. Estas *Metaheurísticas* (también llamadas métodos de trayectoria) trabajan en uno o varias estructuras de vecindario distribuidas en los miembros (las soluciones) del espacio de búsqueda.

Podemos encontrar una filosofía diferente en algoritmos como Optimización a partir de Colonias de Hormigas y Computación Evolutiva. Éstos incorporan un componente de aprendizaje en el sentido de que implícitamente o explícitamente tratan de aprender correlaciones entre variables de decisión para identificar áreas de alta calidad en el espacio de búsqueda. Este tipo de *Metaheurística* efectúa en este sentido un muestreo sesgado del espacio de búsqueda. Por ejemplo, en Computación Evolutiva esto es logrado por una recombinación de soluciones y en Optimización a partir de Colonias de Hormigas esto es logrado por un muestreo del espacio de búsqueda en cada iteración según una distribución de probabilidad.

Hay varias aproximaciones para la clasificación de las *Metaheurísticas* según sus propiedades. A continuación resumiremos brevemente algunos criterios para las clasificaciones

y daremos una descripción de las *Metaheurísticas* básicas de hoy en día. Posteriormente se compararán las diferentes *Metaheurísticas* en el sentido de la implementación por parte de las mismas de los conceptos de *intensificación* y *diversificación* en el proceso de búsqueda. Puntualizaremos semejanzas y diferencias en términos de *intensificación* y *diversificación*.

3. Clasificación de las Metaheurísticas

Hay diferentes formas de clasificar y describir los algoritmos de *Metaheurísticas*. Dependiendo de las características seleccionadas para diferenciarlos, son posibles varias clasificaciones, siendo cada una de ellas el resultado de un punto de vista específico. En esta sección, describiremos brevemente las formas más importantes de clasificar *Metaheurísticas*.

Inspiradas en la naturaleza VS. No inspiradas en la naturaleza: Quizás, la forma más intuitiva de clasificar *Metaheurísticas* está basada en los orígenes del algoritmo. Hay algoritmos inspirados en la naturaleza, como Algoritmos Genéticos y Algoritmos de Hormigas, y algoritmos no inspirados en la naturaleza como Búsqueda Tabú y Búsqueda Local Iterativa. Esta clasificación no es muy significativa por las siguientes dos razones. Primero, muchos algoritmos híbridos recientes no se encuadran dentro de ninguna clase (o bien, en otro sentido, se encuadran en ambas al mismo tiempo). Segundo, es algunas veces difícil decir claramente el origen de un algoritmo.

Basados en población VS. Búsqueda de punto simple: Otra característica que puede ser usada para la clasificación de *Metaheurísticas* es la forma de recorrer el espacio: ¿Trabaja el algoritmo con una población o con una solución simple en cualquier momento? Aquellos algoritmos que trabajan con soluciones simples son llamados métodos de trayectoria e incluyen *Metaheurísticas* basadas en búsqueda local, como Búsqueda Tabú, Búsqueda Local Iterativa, y Búsqueda de Vecindario Variable. Todas ellas comparten la propiedad de describir una trayectoria en el espacio de búsqueda durante el proceso de búsqueda. Las *Metaheurísticas* basadas en población por el contrario ejecutan procesos de búsqueda que describen la evolución de un conjunto de puntos en el espacio de búsqueda.

Función objetivo estáctica VS. Función objetivo dinámica: Las *Metaheurísticas* pueden también ser clasificadas según la forma en que hacen uso de la función objetivo. Mientras que algunos algoritmos guardan la función objetivo dada en la representación del problema "tal como es", otros como la Búsqueda Local Guiada (GLS) la modifican durante la búsqueda. La idea tras esta aproximación es escapar de óptimos locales al modificar el paisaje de búsqueda. Según esto, durante la búsqueda la función objetivo es alterada al tratar de incorporar información recogida durante el proceso de búsqueda.

Una estructura de vecindario VS. Varias estructuras de vecindario: La mayoría de los algoritmos de *Metaheurísticas* trabajan en una estructura de vecindario simple. En otras palabras, el paisaje de fitness que es buscado no cambia en el curso del algoritmo. Otras *Metaheurísticas* como la Búsqueda de Vecindario Variable (VNS) usan un conjunto de estructuras de vecindario que dan la posibilidad de diversificar la búsqueda y abordar el problema saltando entre diferentes paisajes de fitness.

Métodos de memoria VS. Métodos sin memoria: Un rasgo muy importante para clasificar las *Metaheurísticas* es el uso que hacen de la historia de la búsqueda, que consiste en el uso o no de memoria. Los algoritmos sin memoria ejecutan un proceso de Markov, ya que la información que necesitan es sólo el estado actual del proceso de búsqueda. Hay varias formas diferentes de hacer uso de la memoria. Usualmente diferenciamos entre memoria a corto y largo plazo. La primera guarda usualmente pista de movimientos recientemente ejecutados, soluciones visitadas o, en general, decisiones tomadas. La segunda es usualmente una acumulación de parámetros sintéticos e índices sobre la búsqueda. El uso de la memoria es

reconocido actualmente como uno de los elementos fundamentales de una *Metaheurística* poderosa.

A continuación describiremos las *Metaheurísticas* más importantes y que hemos utilizado en nuestro estudio, siguiendo la clasificación de búsqueda de punto simple vs. búsqueda basada en población, que divide a las *Metaheurísticas* en **métodos de trayectoria** y **métodos basados en población**. Esta elección, motivada porque dicha categorización, permite una descripción más clara de los algoritmos. Además, una tendencia actual es la hibridación de métodos en la dirección de la integración de algoritmos de búsqueda de punto simple en algoritmos basados en población.

4. Métodos de trayectoria

En esta sección describiremos las *Metaheurísticas* llamadas *métodos de trayectoria*, ya que su proceso de búsqueda está caracterizado por una trayectoria en el espacio de búsqueda. En lo siguiente nos referiremos a un problema de maximización o minimización P con una función objetivo f, un conjunto de soluciones factibles $S = \{s_1, s_2, ... \}$ y una estructura de vecindario N.

El proceso de búsqueda de los *métodos de trayectoria* puede ser visto como la evolución en tiempo (discreto) de un sistema dinámico discreto [[Bar97], [Dev89]]. El algoritmo parte de un estado inicial (la solución inicial) y describe una trayectoria en el espacio de estados. El sistema dinámico depende de la estrategia usada: los algoritmos simples generan una trayectoria compuesta por dos partes: una fase de tránsito seguida por un atractor (un punto fijo, un ciclo o un atractor complejo). Los algoritmos con estrategias avanzadas generan trayectorias más complicadas que no pueden ser subdivididas en tales dos fases. Las características de la trayectoria perfilan el comportamiento del algoritmo y su efectividad con respecto a la instancia que está atacando. Es de valor subrayar que el dinamismo es el resultado de la combinación de algoritmo, representación del problema e instancia.

Primero describiremos los algoritmos de búsqueda local básicos y después entraremos dentro de estrategias más complejas, finalizando con los algoritmos que definen estrategias generales y pueden incluir otros *métodos de trayectoria* como componentes.

4.1. Búsqueda local básica y su mejora iterativa

Búsqueda local básica

La búsqueda local básica es llamada usualmente *mejora iterativa (ascensión de colinas)*, ya que cada movimiento es ejecutado solamente si la solución que produce es mejor que la solución actual. El algoritmo para tan pronto como encuentra un mínimo local. El algoritmo visto a alto nivel queda esquematizado en la siguiente figura: 1.

Algoritmo de Ascensión de Colinas:

```
s ← GenerarSolucionInicial();
repite
     s ← Mejorar (s, N(s));
hasta (no hay mejora posible en el vecindario);
```

Figura 1.: Algoritmo de Búsqueda Local básica

La función `Mejorar (s, N(s))` puede ser o bien una función del primer mejor o del mejor absoluto del vecindario. La primera estudia el vecindario $N(s)$ y escoge la primera

solución que mejora la función objetivo. La segunda de ellas explora exhaustivamente el vecindario y devuelve la solución con el menor o mayor valor de función objetivo (según sea nuestro problema: minimización o maximización). Ambos métodos paran en un mínimo local, por lo que su rendimiento depende fuertemente de la definición de *S, f* y *N*.

Búsqueda local básica múltiple (MLS)

Teniendo en cuenta las debilidades y la simplicidad de la búsqueda local simple donde un solo arranque conducía a una solución se pensó en una mejora inmediata de múltiple arranque. De ahí llegamos al algoritmo básico de búsqueda multiarranque donde en vez de una única búsqueda local se ejecutan varias en el algoritmo y se toma la mejor solución resultante.

Una búsqueda con arranque múltiple consta de dos fases: Global y Local.

- Fase Global: se genera una solución de la región factible.
- Fase Local: se aplica una búsqueda local desde la solución generada en el paso anterior que acabará en un mínimo local con respecto a la estructura de entorno considerada.

Estos pasos se reiteran hasta que se satisfaga algún criterio de parada.

El mínimo local (máximo) obtenido con menor (mayor) valor de la función objetivo es la solución propuesta por el algoritmo.

En nuestro caso, en el de múltiple arranque básico, la fase global supone generar una solución aleatoria y la fase local ejecutar una búsqueda local sobre la solución anterior. Se trata de encadenar diversas búsquedas locales independientes y obtener un resultado propio de una mayor exploración del espacio de búsqueda y propio de la localización de un mayor número de óptimos locales que con la búsqueda local básica.

La aleatoriedad sigue jugando aquí un papel importante, ya que depende del punto de partida el hallazgo de más o menos zonas prometedoras distintas.

Presentamos en la figura siguiente el algoritmo comentado (figura 2)

Procedimiento Búsqueda con Arranque Múltiple

```
Genera (Solución Actual);
Mejor Solución := Solución Actual;
Repite
      Búsqueda Local(Solución Actual);
      Si Mejor(f(Solución Actual),f(Mejor Solución))
      Entonces Mejor Solución := Solución Actual;
      Genera (Solución Actual);
Hasta (criterio de parada);
```

Figura 2.: Algoritmo de Búsqueda Local Multiarranque (MLS)

El rendimiento de los procedimientos de mejora iterativa en problemas de optimización combinatoria es normalmente bastante satisfactorio, así que se han desarrollado varias técnicas para evitar que los algoritmos queden atrapados en óptimos locales o para escapar de ellos. A continuación, describiremos los más importantes y exitosos.

4.2. Enfriamiento Simulado (Simulated Annealing)

Está comúnmente extendido que el *Enfriamiento Simulado* (Simulated Annealing) es la más antigua de las Metaheurísticas y seguramente uno de los primeros algoritmos que tiene una estrategia explícita para evitar óptimos locales. Los orígenes del algoritmo los encontramos en mecanismos estadísticos (algoritmo de Metrópolis). Se establece el paralelismo entre un proceso termodinámico y la optimización combinatoria. Dicho proceso termodinámico supone la cristalización de un sólido, proceso delicado y que ha de ser convenientemente regulado. En este proceso (de enfriamiento) el objetivo es llevar las moléculas de un estado inicial donde la temperatura es alta a un estado final (microestado fundamental a 0º K). La transición entre estados ha de ser suave y ha de ir redirigida con la temperatura permitiendo la estabilización progresiva de las moléculas en cada estado intermedio.

El algoritmo de Metrópolis ha sido readaptado para los problemas de optimización combinatoria. La distribución de las moléculas del sólido pasa a ser una solución factible, la configuración fundamental óptima pasa a ser la solución óptima y la energía de la configuración pasa a ser el coste de la solución. La temperatura se mantiene como elemento regulador. El algoritmo resultante pertenece a un ámbito completamente distinto: la optimización combinatoria, pero lleva en su nombre sus orígenes.

El algoritmo de *Enfriamiento Simulado* fue presentado inicialmente como algoritmo de búsqueda para problemas de optimización combinatoria en [KGV83] y [Cer85]. La idea fundamental es la de permitir movimientos hacia soluciones con valores de función objetivo más altos (movimientos ascendentes) en caso de problemas de minimización o movimientos hacia soluciones con valores de función objetivo más bajos (movimientos descendentes) en caso de problemas de maximización. De esta forma se intenta escapar de óptimos locales. La probabilidad de hacer tal movimiento se va decrementando a lo largo de la búsqueda. Dicho de otra forma, el algoritmo de *Enfriamiento Simulado* es un Método de Vecindario caracterizado por la *función de aceptación de soluciones vecinas* que se va a ir adaptando a lo largo de las iteraciones, utilizando una variable llamada *Temperatura*. El algoritmo de alto nivel se describe a continuación en pseudocódigo en la figura 3.

Enfriamiento Simulado (Simulated Annealing)

```
s ← GeneraSolucionInicial(); T ← T0;
Mientras (no se alcance la condición de finalización) hacer
{
      Genera un vecino s'∈ N (s) aleatoriamente;
      Si Mejor(f(s'),f(s)) entonces
            s ← s' {s' reemplaza a s};      sino
             Aceptar s' como nueva solución con probabilidad
            p(T,s',s);
      Actualizar(T);}
```

Figura 3.: Algoritmo de Enfriamiento Simulado (Simulated Annealing)

El algoritmo comienza generando una solución inicial (o bien aleatoria o bien construida heurísticamente) e inicializando el parámetro que ha sido llamado temperatura *T*. Entonces repite el proceso de búsqueda hasta que la condición de finalización se alcanza. Son posibles varios criterios de terminación: un tiempo máximo de CPU, un número máximo de iteraciones, se ha encontrado una solución *s* con $f(s)$ menor que un valor umbral predefinido, o se alcanza un máximo número de iteraciones sin mejora. Las instrucciones en el bucle interno son muy simples: se genera una solución *s*' aleatoria del vecindario ($s' \in N(s)$) y ésta se acepta como nueva solución actual dependiendo de $f(s)$, $f(s')$ y *T*. Para problemas de minimización, tenemos que *s*' reemplaza a *s* si $f(s') < f(s)$ o, en el caso de que $f(s') \geq f(s)$, con una probabilidad que está en función de *T* y $f(s') - f(s)$. La probabilidad es calculada generalmente siguiendo la

distribución de Boltzmann: $e^{-\frac{f(s')-f(s)}{T}}$. Por el contrario, para problemas de maximización, tenemos que *s*' reemplaza a *s* si $f(s') > f(s)$ o, en el caso de que $f(s') \leq f(s)$, con una probabilidad que está en función de *T* y $f(s) - f(s')$. La probabilidad calculada siguiendo la distribución de Boltzmann quedaría ahora: $e^{-\frac{f(s)-f(s')}{T}}$.

La temperatura *T* es decrementada durante el proceso de búsqueda. Así al principio de la búsqueda la probabilidad de aceptar movimientos ascendentes (descendentes en el caso de maximización) es alta y se va decrementando gradualmente, convergiendo a un algoritmo de mejora iterativo simple. Este proceso es análogo al proceso de enfriamiento de metales y vidrio, que asume una configuración de baja energía cuando han sido refrescados tras un plan de enfriamiento apropiado. Con respecto al proceso de búsqueda, esto significa que el algoritmo es el resultado de dos estrategias combinadas: desplazamiento aleatorio y mejora iterativa. En la primera fase de la búsqueda, el sesgo hacia la mejora es bajo y permite la exploración del espacio de búsqueda; este componente variable es lentamente decrementado hasta llevar a la búsqueda a converger a un óptimo (local). La probabilidad de aceptar movimientos ascendentes en caso de minimización o descendentes en caso de maximización, está controlada por dos factores: la diferencia de las funciones objetivo y la temperatura. Por una parte, a una temperatura fija, cuanto más alta sea la diferencia entre $f(s')$ y $f(s)$, más baja será la probabilidad de aceptar un movimiento de *s* a *s'*. Por otra parte, cuanto más alto sea el valor de *T*, más alta será la probabilidad de realizar movimientos ascendentes en la minimización y movimientos descendentes en la maximización.

La elección de un plan apropiado de enfriamiento es crucial para la ejecución del algoritmo. El plan de enfriamiento define el valor de *T* en cada iteración *k*, $T_{k+1}=Q(T_k, k)$, donde $Q(T_k, k)$ es una función de la temperatura en el paso previo y del número de iteración. Los resultados teóricos en cadenas de Markov no homogéneas [AKL97] declaran que bajo particulares condiciones en el plan de enfriamiento, el algoritmo converge en probabilidad a un mínimo global para $k \rightarrow \infty$. Más precisamente tal y como se establece en la ecuación 2.1.

$$\exists \Gamma \in \Re \quad \text{tq.} \quad \lim_{k\to\infty} \textit{Prob}\,[\text{óptimo global encontrado tras } k \text{ pasos o iteraciones }] = 1$$

$$\text{sii} \qquad \sum_{k=1}^{\infty} \exp\left(\frac{\Gamma}{T_k}\right) = \infty \qquad (1)$$

Un plan de enfriamiento particular que completa la hipótesis para la convergencia es la única que sigue una ley logarítmica (ecuación 2.2):

$$T_{k+1} = \frac{\Gamma}{\log(k + k_0)} \qquad (2)$$

Desafortunadamente, los planes de enfriamiento que garantizan la convergencia a un óptimo global no son factibles en la práctica porque necesitan tiempo infinito. Por lo tanto, las aplicaciones adoptan los planes de enfriamiento más rápidos. Uno de los más usados sigue una ley geométrica (ecuación 2.3):

$$T_{k+1} = \alpha\, T_k \qquad (3)$$

donde $\alpha \in]0,1[$, que corresponde a un descenso exponencial de la temperatura.

La regla de enfriamiento puede variar durante la búsqueda, con el propósito de proporcionar equilibrio entre diversificación e intensificación. Por ejemplo, al principio de la

búsqueda, *T* puede ser constante o seguir un decremento lineal, para muestrear el espacio de búsqueda; después, *T* puede seguir una regla como la geométrica, para converger a un óptimo local al final de la búsqueda. Se usan también planes no monótonos [[Osm93], [LM86]], que alternan fases de enfriamiento y recalentamiento, proporcionando así una estrategia oscilante para controlar la diversificación y la intensificación.

El plan de enfriamiento y la temperatura inicial deberían ser adaptados a la instancia particular del problema, ya que el coste de escapar de mínimos locales depende de la estructura del espacio de búsqueda. Una forma simple de determinar empíricamente la temperatura de inicio es la de muestrear inicialmente el espacio de búsqueda con una búsqueda aleatoria para evaluar grosamente el promedio y la varianza para los valores de la función objetivo. Sin embargo, pueden ser implementados esquemas más elaborados todavía [Ing96].

El *Enfriamiento Simulado* (Simulated Annealing) ha sido aplicado a varios problemas de optimización combinatoria, desde el viajante de comercio (TSP) a diseño de layout y problemas de estructura de las proteínas [[AL97], [Ing96], [Fle95]]. Es hoy en día usado también como un componente en las *Metaheurísticas*, además de ser usado como algoritmo de búsqueda en sí mismo.

El proceso dinámico descrito por el *Enfriamiento Simulado* (Simulated Annealing) es una cadena de Markov [Fel68], ya que sigue una trayectoria en el espacio de estados donde el estado sucesor es escogido dependiendo sólo del incumbente. Por lo tanto, este proceso es sin memoria.

Parámetros y componentes específicos del algoritmo ES

Selección de la solución inicial

La selección de la solución inicial tiene normalmente dos vertientes significativas:

- Uso de técnicas eficientes para su obtención: p. ej. Algoritmo greedy.
- Uso de conocimiento experto.

Mecanismo de transición

El mecanismo de transición supone tres pasos fundamentales:

- Generación de una nueva solución.
  - Definición del conjunto de vecinos.
  - Selección de un elemento de dicho conjunto.
- Cálculo de la diferencia de costos.
- Mecanismo de aceptación de una nueva solución (basado en criterio de Metrópolis).

Secuencia de enfriamiento

La secuencia de enfriamiento cubre los siguientes apartados:

1. Valor inicial del parámetro de control (temperatura)

No parece conveniente considerar valores fijos independientes del problema.

Una propuesta comúnmente usada es la siguiente…

$T_0 = (\mu / -\ln(\varphi)) \cdot c(S_0)$ *Minimización*

$T_0 = (\mu / -\ln(\varphi)) \cdot (1/c(S_0))$ *Maximización* (*4*)

donde :

T_0: temperatura inicial.

μ: porcentaje en tanto por 1 que indica lo que se está dispuesto a perder.

φ: probabilidad con que se aceptará soluciones dentro del porcentaje μ.

$c(S_0)$: costo de la solución de partida que inicializa el algoritmo.

En esta propuesta indicamos, por tanto, el tanto por uno φ de probabilidad de que una solución sea un μ por uno peor que la solución inicial S_0 .

2. Función de enfriamiento de energía

Se tienen diferentes mecanismos de enfriamiento.

a. Enfriamiento basado en sucesivas temperaturas descendentes fijadas por el usuario.

b. Enfriamiento con descenso constante de temperatura.

c. Descenso con una constante proporcional

$T_{k+1}=\alpha \cdot T_k$, k=1,2,… (*5*)

donde:

T_{k+1} : temperatura en el nivel o iteración de enfriamiento k+1.

α: constante pequeña cercana a 1 (suele tomar valores entre 0.8 y 0.99)

T_k : temperatura en el nivel o iteración de enfriamiento k.

d. Criterio de Boltzmann

$T_k = T_0 / (1 + \log(k))$ (*6*)

donde:

T_k : temperatura en el nivel o iteración de enfriamiento k.

T_0: temperatura inicial.

k: número de nivel o iteración de enfriamiento.

e. Esquema de Cauchy

$T_k = T_0 / (1 + k)$ (*7*)

donde:

T_k : temperatura en el nivel o iteración de enfriamiento k.

T_0: temperatura inicial.

k: número de nivel o iteración de enfriamiento.

Existe una modificación que permite controlar el número de iteraciones…

$$T_{k+1} = T_k / (1 + \beta \cdot T_k)$$

$$\beta = (T_0 - T_f) / M \cdot T_0 \cdot T_f \qquad (8)$$

donde:

T_{k+1} : temperatura en el nivel o iteración de enfriamiento k+1.

T_k : temperatura en el nivel o iteración de enfriamiento k.

T_0: temperatura inicial.

T_f : temperatura final.

M: número de iteraciones de enfriamiento del algoritmo.

3. Velocidad de enfriamiento

El número de vecinos a generar L(T) debe ser suficientemente grande como para que el sistema llegue a alcanzar su estado estacionario para esa temperatura.

Se establecen dos posibilidades fundamentales para L(T)…

- Número fijo.
- Número fijo de vecinos generados (n) y número fijo de vecinos acertados (m).
  - Un enfriamiento ocurre si se cumple n o si se cumple m.
  - El valor de m normalmente debe ser mucho menor que el de n.
  - Si el valor de n es nulo el algoritmo termina.

4. Temperatura final

En teoría deberíamos tener que la temperatura final debería ser 0. Sin embargo, en la práctica, ocurre que cuando la temperatura está por debajo de un valor final T_f, o después de un número determinado de iteraciones, el algoritmo termina.

Algoritmos de Enfriamiento Simulado Paralelos

Los algoritmos paralelos de búsqueda suponen una alternativa apetecible a la hora del hallazgo de buenas soluciones. Permiten, entre otras posibilidades, las siguientes, frente al típico algoritmo secuencial:

1. Preservar la calidad de las soluciones reduciendo el tiempo de ejecución.
2. Incrementar la calidad de las soluciones sin aumentar el tiempo de cálculo:
   a. Aumentando las iteraciones con una paralelización efectiva.
   b. Introduciendo mayor diversidad en el proceso de búsqueda y evitando así la convergencia prematura (*ventajas de un diseño paralelo ejecutado secuencialmente*).

El algoritmo de *Enfriamiento Simulado* presenta varios modelos que permiten su ejecución en paralelo. Todos los algoritmos se pueden paralelizar, pero en las ***Metaheurísticas*** que describiremos en este trabajo, solamente comentaremos la paralelización de aquellos algoritmos que se han utilizado como tales en la práctica en el estudio sobre nuestro problema.

Veamos por tanto algunos de los modelos que se establecen para la ejecución paralela del algoritmo de *Enfriamiento Simulado*. Dicha ejecución paralela puede llevarse también a través de un diseño paralelo ejecutado secuencialmente.

Modelo clásico. Ejecuciones independientes

Este modelo es el más simple y el que inicialmente sale a relucir para la paralelización de todo algoritmo. Consiste en la ejecución por parte de cada procesador (real o ficticio) de un algoritmo independiente de *Enfriamiento Simulado*. Al final del proceso se selecciona la mejor solución.

Estrategia de División

Este modelo va a ser un poco más elaborado. En este caso, cada procesador ejecuta un algoritmo de *Enfriamiento Simulado* también. Pero, cada k iteraciones se escoge uno de los múltiples estados (solución actual o solución actual y mejor solución) y se copia en todos los procesadores. Se continúa la búsqueda en todos los procesadores a partir de la solución seleccionada.

Aquí aparece la comunicación entre los procesadores, algo que amplía las posibilidades de la búsqueda. Al final del proceso se selecciona la mejor solución.

Configuración maestro/esclavo

Seguimos en la tónica de modelo elaborado. Ahora, se dispone de un procesador maestro, y el resto de procesadores actúan como esclavos. Cada uno de los procesadores tiene su función propia en base a su categoría y que a continuación se relata:

Procesador Maestro

- Procesa el estado (solución actual) y distribuye este estado a los procesadores esclavos.

Procesadores Esclavos

- Generan un vecino y aplican el criterio de selección.
- Cuando un procesador consigue un vecino aceptado, lo pasa al procesador maestro y este realiza su labor.
- Se utiliza un criterio de selección entre las soluciones aceptadas por los procesadores esclavos.

La comunicación ahora es distinta y un poco más refinada.

Existen así mismo, otros modelos que hibridan los anteriormente comentados.

4.3. Búsqueda Tabú

La *Búsqueda Tabú* está entre las heurísticas más citadas y usadas para los problemas de optimización combinatoria. Las ideas básicas de la *Búsqueda Tabú* fueron introducidas en primer lugar en [Glo86] y esbozada independientemente en [Han86]. Una descripción del método y sus conceptos puede ser encontrada en [GL97]. Estamos ante una técnica de búsqueda por entornos/vecindario de propósito general que incorpora memoria adaptativa y exploración sensible. La *Búsqueda Tabú* usa explícitamente la historia de la búsqueda, tanto para evitar óptimos locales como para implementar una estrategia exploratoria. Primero describiremos una versión simple de la *Búsqueda Tabú*, para introducir los conceptos básicos. Después explicaremos un algoritmo más aplicable y finalmente discutiremos algunas mejoras.

Procedimiento Búsqueda Tabú Simple

```
s ← GenerarSolucionInicial();
ListaTabu ← Ø
Mientras (no se alcance la condicion de terminacion) hacer
{
      s ← Mejor_mejora(s,N(s)\ListaTabu);
      Actualizar(ListaTabu);
}
```

Figura 4.: Algoritmo de Búsqueda Tabú Simple

El algoritmo de *Búsqueda Tabú* simple (ver figura 4) aplica una búsqueda local de mejor mejora como ingrediente básico y usa una memoria de corto plazo para escapar de óptimos locales y evitar ciclos. La memoria de corto plazo es normalmente conocida como lista tabú. Ésta guarda pista de las soluciones más recientemente visitadas y prohíbe moverse hacia ellas. El vecindario de la solución actual está así restringido a las soluciones que no pertenecen a la lista tabú. En lo siguiente nos referiremos a este conjunto como *conjunto de soluciones permitidas*. La nueva solución generada es entonces añadida a la lista y una de las soluciones de la lista es descartada (usualmente en un orden FIFO). Debido a esta restricción dinámica de soluciones permitidas en un vecindario, la *Búsqueda Tabú* puede ser considerada como una técnica de búsqueda de vecindario dinámico [Stü99a]. La condición de terminación puede ser un tiempo máximo de CPU, un número máximo de iteraciones, la generación de una solución s con $f(s)$ menor que un valor umbral predefinido, o un máximo número de iteraciones sin mejora. El algoritmo puede también terminar si el conjunto de soluciones permitidas está vacío, esto es, si todas las soluciones en $N(s)$ están prohibidas por la lista tabú. Hay que decir sin embargo que, son posibles otras estrategias de escape para evitar parar cuando el conjunto de soluciones permitidas está vacío: por ejemplo, se puede elegir la solución menos visitada recientemente, incluso si es tabú.

El uso de una lista tabú previene de volver a soluciones recientemente visitadas, previniendo por lo tanto de bucles infinitos (son posibles los ciclos de alto orden, ya que la lista tabú tiene una longitud finita l que es más pequeña que la cardinalidad del espacio de búsqueda) y fuerza a la búsqueda a aceptar incluso movimientos descendentes en problemas de maximización o movimientos ascendentes en problemas de minimización. La longitud l de la lista tabú (tenencia tabú) controla la memoria del proceso de búsqueda. Con pequeña tenencia tabú la búsqueda se concentrará en áreas pequeñas del espacio de búsqueda. Por el contrario, una tenencia tabú grande fuerza al proceso de búsqueda a explorar regiones más amplias, porque prohíbe revisitar un número más alto de soluciones. La tenencia tabú puede ser variada durante la búsqueda, llevando a constituir algoritmos más robustos. Se puede encontrar un ejemplo en [Tai91], donde la tenencia tabú es periódicamente reinicializada de forma aleatoria en el intervalo $[l_{min}, l_{max}]$. Un uso más avanzado de la tenencia tabú dinámica se presenta en [[BT94], [BP97]], donde la tenencia tabú es incrementada si hay evidencia para que se produzca la repetición de soluciones (necesitándose así una más alta diversificación), mientras que es decrementada si no hay mejoras (necesitándose así aumentar la intensificación).

La implementación de la memoria de corto plazo como una lista de soluciones visitadas no es práctica, porque manejar una lista de soluciones es altamente ineficiente. Por lo tanto, en vez de soluciones en sí mismas, se guardan los *atributos* de las mismas. Los atributos son usualmente componentes de soluciones, como movimientos y diferencias entre dos soluciones. Si puede ser considerado más de un atributo, se introduce una lista tabú por cada uno de ellos. El conjunto de atributos y listas tabú relacionadas definen las *condiciones tabú* que son usadas para filtrar el vecindario de una solución y generar el *conjunto de soluciones permitidas*.

Almacenar movimientos en vez de soluciones completas es mucho más eficiente, pero introduce una pérdida de información, ya que prohibir un movimiento significa asignar el estado tabú a probablemente más de una solución. Así, es posible que soluciones no visitadas de buena calidad sean excluidas del conjunto de soluciones permitidas. Para solucionar este problema, se definen los criterios de aspiración que permiten incluir una solución en el conjunto de soluciones permitidas incluso si está prohibida por condiciones tabú. Los criterios de aspiración definen las condiciones de aspiración que son usadas para construir el conjunto de soluciones permitidas. El criterio de aspiración más comúnmente usado selecciona las soluciones que son mejores que la actual. El algoritmo completo, tal y como ha sido descrito, se muestra en la siguiente figura:

Procedimiento Búsqueda Tabú

```
s ← GeneraSolucionInicial();
InicializaListasTabu(LT1,LT2,...,LTr);
k ← 0;
mientras (no se alcancen las condiciones de terminacion)
hacer
{
      SolucionesPermitidas(s,k) ← { z ∈ N(s) |
                                    ninguna condición tabú ha
                                    sido violada y ha sido
                                    satisfecha una condición
                                    de aspiración al menos } ;
      s ← MejorMejora(s,SolucionesPermitidas(s,k));
      ActualizarListasTabuYCondicionesAspiracion();
      k ← k + 1;
}
```

Figura 5.: Algoritmo de Búsqueda Tabú

Las listas tabú son sólo una de las posibles formas de tomar ventaja de la historia de la búsqueda, y constituyen una memoria de corto plazo. La información recopilada durante el proceso de búsqueda completo puede ser muy útil, especialmente para una guía estratégica del algoritmo. Este tipo de memoria de largo plazo es usualmente añadida a la *Búsqueda Tabú* al aludir a cuatro principios: *recencia*, *frecuencia*, *calidad* e *influencia*. La memoria basada en recencia guarda para cada solución (o atributo) la iteración más reciente en la que ocurrió. Ortogonalmente, la memoria basada en frecuencia guarda pista de cuántas veces ha sido visitada cada solución (atributo). Esta información identifica las regiones (o los subconjuntos) del espacio de soluciones donde la búsqueda fue restringida, o donde permaneció durante un alto número de iteraciones. Este tipo de información sobre el pasado es usualmente utilizada para diversificar la búsqueda. El tercer principio, calidad, es una guía para aprender y extraer información de la historia de búsqueda para identificar componentes de buenas soluciones. Esta información puede ser útilmente integrada en la construcción de la solución inicial. Otras metaheurísticas (como por ejemplo, *Optimización* a partir de *Colonias* de *Hormigas*) usan explícitamente este principio para aprender buenos bloques de construcción de soluciones de alta calidad. Finalmente, la influencia es una propiedad respecto de las elecciones hechas

durante la búsqueda y puede ser usada para indicar qué elecciones han mostrado ser las más críticas.

Tenemos por tanto que, la memoria en la *Búsqueda Tabú* tiene dos componentes fundamentales:

- Memoria de corto plazo: Guarda información que permite guiar la búsqueda de forma inmediata a partir de la lista de soluciones candidatas, desde el comienzo del procedimiento.

- Memoria de largo plazo: Guarda información que permite guiar la búsqueda a largo plazo, después de una primera etapa en la que se aplica la memoria a corto plazo. Con la información guardada en esta memoria se suele intensificar la búsqueda volviendo a visitar buenas soluciones o diversificar la búsqueda visitando nuevas zonas a explorar.

Estrategias de intensificación y diversificación

Uno de los componentes fundamentales y más significativos dentro de la *Búsqueda Tabú* viene proporcionado por la posible reinicialización de la búsqueda. Dicha reinicialización supone proseguir la ejecución del algoritmo en busca de mejores soluciones una vez que la ejecución actual se encuentra en una fase de estancamiento. Permitimos así mismo la exploración y explotación de otras regiones del espacio en base a la historia de la ejecución del algoritmo que ha quedado almacenada en la memoria.

Teniendo en cuenta la potencialidad que esta posibilidad supone, se han establecido las estrategias de intensificación y diversificación para la reinicialización y que a continuación pasamos a comentar (normalmente se utiliza una estrategia oscilante entre las dos, según convenga)…

Estrategias de intensificación

Están basadas en la modificación de reglas de elección para que se favorezcan combinaciones de movimientos y características de solución que históricamente hayan sido buenas. Pueden iniciar un regreso a regiones atractivas y buscar en ellas más extensamente.

Dos variantes comunes son las siguientes…

- Introducir una medida de diversificación para asegurar que las soluciones registradas difieran una de otra en un grado deseado. Borrar toda memoria de corto plazo para reanudar el proceso desde la mejor de las soluciones registradas.

- Mantener una lista secuencial de longitud limitada que añade al final una nueva solución sólo si en su momento fue mejor que cualquier otra previamente vista. El último miembro de la lista es siempre el escogido (y suprimido) como base para reanudar la búsqueda. La memoria de corto plazo que acompañó a esta solución también es guardada.

Estrategias de diversificación

Están diseñadas para conducir la búsqueda hacia nuevas regiones. Suelen estar basadas en modificaciones de las reglas de elección para llevar a la solución atributos (o movimientos) que no hayan sido usados frecuentemente. Para ello se suele modificar la información sobre el

problema (las valoraciones de las funciones objetivo) penalizando soluciones con valores de los atributos muy visitados.

La *Búsqueda Tabú* ha sido aplicada a muchos problemas de optimización combinatoria; las aplicaciones más exitosas son la Búsqueda Tabú Robusta al problema de la asignación cuadrática (QAP) [Tai91], la Búsqueda Tabú Reactiva al problema MAXSAT [BP97], y otras a encaminamiento, planificación y problemas de asignación [DLM99].

4.4. Métodos de Búsqueda Local Explorativa o Multiarranque

En esta sección presentaremos métodos de trayectoria más actuales: los algoritmos de *búsqueda local multiarranque*. Una búsqueda con arranque múltiple consta de dos fases: global y local. En la fase global se genera una solución de una región factible del espacio de búsqueda. En la fase local se aplica una búsqueda local desde dicha solución generada que acabará en un óptimo local con respecto a la estructura de entorno considerada. Dichas fases se alternan hasta que se satisface el criterio de parada del algoritmo. El óptimo local obtenido con mejor valor de la función objetivo es la solución propuesta por el algoritmo.

A continuación presentaremos distintos algoritmos de *búsqueda multiarranque*, partiendo de que el algoritmo esencial ya ha sido presentado al hablar de la búsqueda local múltiple o MLS en el apartado 2.4.1.2. Los algoritmos a comentar serán el ILS (Búsqueda Local Iterativa) y el VNS (Búsqueda de Vecindario Variable).

ILS (Búsqueda Local Iterativa)

Pasamos, en primer lugar, a hablar de la Búsqueda Local Iterativa (*ILS*), el esquema más general de las estrategias exploratorias. Por una parte, su generalidad la hace un marco de trabajo para otras *Metaheurísticas* (como VNS). Por otra parte, otras *Metaheurísticas* pueden ser fácilmente incluidas en la misma como subcomponentes. *ILS* es una *Metaheurística* simple pero poderosa [[Stü99a], [Stü99b], [RMS02], [MOF91]]. Aplica la búsqueda local a una solución inicial hasta que encuentra un óptimo local; entonces muta la solución y reinicia la búsqueda local. La importancia de la perturbación o mutación es obvia: un cambio muy pequeño en la solución puede no hacerle posible escapar de la atracción del óptimo local encontrado, y por otro lado, una modificación demasiado grande nos dispone cercanos a una simple reinicialización aleatoria.

Una búsqueda local es efectiva si es capaz de encontrar buenos óptimos locales, esto es, si puede encontrar las zonas de atracción de puntos fijos correspondientes a aquellos óptimos. Cuando el espacio de búsqueda es ancho y/o cuando las zonas de atracción de óptimos locales buenos son pequeñas, un simple algoritmo multiarranque es casi inútil. Una búsqueda efectiva puede ser designada como una trayectoria sólo en el conjunto de óptimos locales S^*, en vez de en el conjunto S de todos los estados. Desafortunadamente, la única forma de introducir una estructura de vecindario en S^* sería construyendo (o al menos muestreando) la zona de atracción de óptimos locales involucrada en la búsqueda. Como esto no es factible en la práctica, una trayectoria a lo largo de los óptimos locales $s_1^*, s_2^*, \ldots, s_t^*$ puede ser ejecutada sin introducir explícitamente una estructura de vecindario, al aplicar el siguiente esquema:

1. Ejecutar búsqueda local desde un estado inicial s hasta que un óptimo local s^* sea encontrado.
2. Mutar o perturbar s^* y obtener s'.
3. Ejecutar la búsqueda local desde s' hasta encontrar un óptimo local $s^{*'}$.
4. Teniendo como base un criterio de aceptación decidir si asignar a s^* la solución $s^{*'}$.
5. Ir al paso 2.

El objetivo de la perturbación de s^* es el de producir un punto de inicio para la búsqueda local tal que ésta finalice en un óptimo local distinto de s^*, pero normalmente de mayor calidad que un óptimo local alcanzable a partir de una reinicialización aleatoria. El criterio de aceptación actúa como un contrapeso, ya que filtra y da realimentación a la mutación, dependiendo de las características del nuevo óptimo local.

El algoritmo de alto nivel para *ILS* queda establecido en la figura 6, tal y como aparece en [RMS02].

Procedimiento ILS Básico

```
s0 ← GeneraSolucionInicial();
s* ← BusquedaLocal(s0);
mientras (no se alcance la condición de finalización) hacer
{
      s' ← Mutacion(s*, historia);
      s*' ← BusquedaLocal(s');
      s* ← AplicarCriterioAceptacion(s*, s*', historia); }
```

Figura 6.: Algoritmo ILS Básico

El diseño de los algoritmos *ILS* tiene varios grados de libertad en la elección de la solución inicial, mutación y criterios de aceptación. La historia de la búsqueda juega un papel clave, ya que puede ser explotada tanto como memoria a corto plazo como a largo plazo.

La construcción de soluciones iniciales debería ser rápido sin ser costoso computacionalmente. Las soluciones iniciales deberían ser un punto de inicio bueno para la búsqueda local. La forma más rápida de producir una solución inicial es generarla aleatoriamente; sin embargo, esta es la forma más fácil para los problemas donde cada posible asignación es una solución factible, mientras que en otros casos la construcción de una solución factible requiere también comprobación de restricciones. También pueden ser adoptados métodos constructivos basados en heurísticas. Es importante subrayar que una solución inicial es considerada un buen punto de inicio dependiendo de la búsqueda local particular aplicada y de la estructura del problema, ya que el objetivo del diseñador del algoritmo es encontrar un equilibrio entre velocidad y calidad de soluciones.

La perturbación es normalmente no determinística, para evitar ciclos. Su característica más importante es la magnitud, definida a grandes rasgos como la cantidad de cambios hechos en la solución actual. La magnitud puede ser o bien fija o bien variable. En el primer caso, la distancia entre s^* y s' se mantiene constante independientemente del tamaño del problema. Sin embargo, una magnitud dinámica es en general más efectiva, ya que experimentalmente se ha encontrado que en la mayoría de los problemas, cuanto mayor es el tamaño del problema, más grande debería ser la magnitud de la mutación. Son posibles esquemas más sofisticados: por ejemplo, la magnitud puede ser adaptativa (incrementarse cuando se necesita más diversificación y decrementarse cuando la intensificación sea preferible). El algoritmo VNS que luego veremos pertenece a esta categoría. Una elección ortogonal con respecto a la magnitud es el tipo de mutación o perturbación. Puede ser aleatoria, o producida por un algoritmo (por ejemplo, una búsqueda local distinta a la usada en el algoritmo principal).

El tercer componente importante es el criterio de aceptación. Hay dos casos extremos: aceptar el nuevo óptimo local solo en caso de mejora o siempre aceptar el nuevo estado. En realidad, hay varias posibilidades. Por ejemplo, es posible adoptar un tipo de plan de enfriamiento: aceptar todos los nuevos óptimos locales que supongan mejora y aceptar también aquellos que no supongan mejora con una probabilidad que esté en función de una temperatura T y la diferencia de los valores de las función objetivo…

Minimización

$$\Pr ob\left[Accept(s^*, s', historia)\right] = \begin{cases} 1 & \text{si } f(s^{*'}) < f(s^*) \\ \exp\left(\dfrac{-f(s^{*'}) + f(s^*)}{T}\right) & \text{en otro caso} \end{cases} \quad (9)\text{(a)}$$

Maximización

$$\Pr ob\left[Accept(s^*, s', historia)\right] = \begin{cases} 1 & \text{si } f(s^{*'}) > f(s^*) \\ \exp\left(\dfrac{f(s^{*'}) - f(s^*)}{T}\right) & \text{en otro caso} \end{cases} \quad (9)\text{(b)}$$

El plan de enfriamiento puede ser o bien monótono (sin incrementarse a lo largo del tiempo) o bien no monótono (estableciendo un equilibrio entre diversificación e intensificación). El plan no monótono es particularmente efectivo si explota la historia de la búsqueda. Cuando la intensificación parece ya no ser efectiva, se necesita una fase de diversificación y se incrementa la temperatura.

ILS Basada en Poblaciones

En el algoritmo *ILS* se va trabajando con una solución en cada una de las iteraciones. Sin embargo, existe la posibilidad de trabajar con una población de soluciones en vez de con una sola, o sea, aplicar varios *ILS* en paralelo entre los que exista una comunicación. A continuación veremos dos posibles modelos muy utilizados en los que se utiliza tal estrategia.

Modelo reemplazar el peor:

Se parte de una población de soluciones. Se aplica el algoritmo *ILS* básico sobre cada una de esas soluciones sobre un cierto número de iteraciones. Se reemplaza la peor solución encontrada por la mejor y se obtiene la nueva población inicial de soluciones.

Este esquema se repite hasta que se cumpla el criterio de parada determinado. La motivación asociada a este esquema es ir concentrando gradualmente la búsqueda alrededor de la mejor solución de la población.

Modelo EE(μ+λ):

Partimos de μ soluciones. A partir de mutación obtenemos λ individuos. Aplicamos búsqueda local a los λ individuos durante un cierto número de iteraciones. De los λ individuos resultantes tomamos los μ mejores y obtenemos la nueva población inicial. Obviamente, tendremos que el parámetro $\lambda > \mu$.

Debido a su flexibilidad y alto nivel de abstracción, *ILS* puede ser considerada el marco de trabajo básico para la mayoría de las *Metaheurísticas*. Su aplicación va desde problemas como el TSP o MAXSAT a problemas de planificación (para un resumen de las aplicaciones ver [RMS02]).

VNS (Búsqueda basada en Entornos Cambiantes)

La Búsqueda de Vecindario Variable (*VNS*) [[HM99], [HM01]] es una *Metaheurística* que aplica explícitamente una estrategia basada en estructuras de vecindario dinámicamente cambiantes. El algoritmo es muy general y existen muchos grados de libertad para diseñar variantes e instanciaciones particulares.

En un primer paso debe ser definido un conjunto de estructuras de vecindario. Estos vecindarios pueden ser escogidos arbitrariamente, pero se suele definir una secuencia de vecindarios con cardinalidad incremental: $|N_1| < |N_2| < \ldots < |N_{kmax}|$. Después de que se genera una solución inicial, el índice de vecindario es inicializado y el algoritmo itera hasta que se alcanza una condición de parada. El algoritmo de alto nivel se establece en la siguiente figura…

Procedimiento VNS Básico

```
SeleccionConjuntoEstructurasVecindario N_k , k=1,…,k_max ;
s ← GeneraSolucionInicial();
mientras (no se alcance condición de finalización) hacer
{
      k ← 1;
      mientras (k < k_max) hacer
      {
            s' ← SeleccionaPuntoInicioVecindario(N_k(s));
                  {fase de sacudida}
            s''← BusquedaLocal(s');
            si f(s'')<f(s) entonces
            {
                  s ← s'';
                  k ← 1;
            }
            sino
                  k ← k+1;
      }
}
```

Figura 7.: Algoritmo VNS Básico

El ciclo principal del *VNS* está compuesto de tres fases: sacudida, búsqueda local y movimiento. En la fase de sacudida se selecciona aleatoriamente una solución *s*' en el k-ésimo vecindario de la actual solución *s*. Entonces, *s*' se hace el punto de inicio de la búsqueda local. La búsqueda local puede usar cualquier estructura de vecindario y no está restringida al conjunto definido para el *VNS*. Al final del proceso de búsqueda local (terminado tan pronto como se verifica una condición de terminación predefinida) la nueva solución *s*'' se compara con *s* y, si es mejor, reemplaza a *s* y el algoritmo comienza de nuevo con k=1. En otro caso, k es incrementado y comienza una nueva fase de sacudida usando un vecindario diferente.

El objetivo de la fase de sacudida es el de proporcionar un buen punto de partida para la búsqueda local. El punto de inicio debería pertenecer a la zona de atracción de un óptimo local diferente del actual, pero no debería ser "demasiado lejano" de *s*, ya que en otro caso el algoritmo degeneraría en un simple multiarranque. Además, eligiendo *s*' en el vecindario de la actual mejor solución es probable producir una solución que mantenga algunos rasgos buenos de la actual.

El proceso de cambiar vecindarios en el caso de que no haya mejoras corresponde a una diversificación de la búsqueda. En particular, la elección de vecindarios de cardinalidad creciente conduce a una diversificación progresiva. La efectividad de esta estrategia de vecindario dinámica puede ser explicada por el hecho de que un mal lugar en el espacio de búsqueda dado por un vecindario podría ser un buen lugar en el espacio de búsqueda dado por otro vecindario. Además, una solución que es localmente óptima con respecto a un vecindario no es probablemente localmente óptima con respecto a otro vecindario. Este hecho está siendo explotado por una búsqueda local llamada Descenso de Vecindario Variable (VND), esquematizado en el algoritmo mostrado en la página siguiente…

Procedimiento Descenso de Vecindario Variable (VND)

```
SeleccionConjuntoEstructurasVecindario N_k , k=1,…,k_max ;
s ← GeneraSolucionInicial();
mientras (no se alcance condición de finalización) hacer {
      k ← 1;
      mientras (k < k_max) hacer   {
            s' ← MejorMejora(N_k(s));
            si Mejor(f(s'),f(s)) entonces
                  s ← s';
            sino
                  k ← k+1;   }}
```

Figura 8.: Algoritmo VND

En VND se aplica una búsqueda local de mejor mejora, y, en caso de que se encuentre un óptimo local, la búsqueda procede con otra estructura de vecindario. Como puede observarse en lo descrito anteriormente, la elección de las estructuras de vecindario es el punto crítico de *VNS* y VND. Los vecindarios escogidos deberían mostrar y representar diferentes propiedades y características del espacio de búsqueda, esto es, las estructuras de vecindario deberían dar diferentes abstracciones del espacio de búsqueda. Una variante de *VNS* es obtenida al seleccionar los vecindarios de tal forma que se permita producir una descomposición del problema. El algoritmo es llamado Búsqueda de Descomposición de Vecindario Variable (VNDS) y es descrito en la página siguiente…

Procedimiento Búsqueda por Descomposición de Vecindario Variable

```
SeleccionConjuntoEstructurasVecindario N_k , k=1,…,k_max ;
s ← GeneraSolucionInicial();
mientras (no se alcance condición de finalización) hacer
{
      k ← 1;
      mientras (k < k_max) hacer
      {
            s' ← SeleccionaPuntoInicioVecindario(N_k(s));
                  {s y s' difieren en un conjunto de k
                  atributos}
            s''← BusquedaLocal(s',atributos);
                  {solo se mueve involucrando a los k
                  atributos permitidos}
            si Mejor(f(s''),f(s)) entonces
            {
                  s ← s'';
                  k ← 1;
            }
            sino
                  k ← k+1;
      }
}
```

Figura 9.: Algoritmo VNDS

VNDS sigue el esquema usual VNS, pero las estructuras de vecindario y la búsqueda local son definidas en subproblemas. Para cada solución, todos los atributos (usualmente variables) se mantienen fijos excepto *k* de ellos. Para cada *k* se define una estructura de

vecindario N_k. La búsqueda local solo supone cambios en las variables que pertenecen al subproblema al que se aplica.

La decisión de ejecutar o no un movimiento puede ser variada también. El criterio de aceptación basado en mejoras supone un descenso orientado fuertemente pendiente y podría no ser adecuado para explorar efectivamente el espacio de búsqueda. Por ejemplo, cuando los óptimos locales están agrupados, VNS puede encontrar rápidamente el mejor óptimo en un agrupamiento, pero no tiene guía para abandonar ese agrupamiento y encontrar otro. SVNS extiende VNS al proporcionar un criterio de aceptación más flexible que tiene también en cuenta la distancia desde la solución actual. El algoritmo es esquematizado en la siguiente figura.

Procedimiento SVNS

```
SeleccionConjuntoEstructurasVecindario N_k , k=1,…,k_max ;
s ← GeneraSolucionInicial();
mientras (no se alcance condición de finalización) hacer
{
      k ← 1;
      mientras (k < k_max) hacer
      {
            s' ← SeleccionaPuntoInicioVecindario(N_k(s));
                  {fase de sacudida}
            s''← BusquedaLocal(s');
            si Mejor(f(s''),f(s)) entonces
                  s ← s'';
            si Mejor(f(s'')-αρ(s,s''),f(s)) entonces
            {
                  s ← s'';
                  k ← 1;
            }
            sino
                  k ← k+1;
      }
}
```

Figura 10.: Algoritmo SVNS Básico

El nuevo criterio de aceptación es el siguiente: además de siempre aceptar mejoras, las soluciones peores pueden ser aceptadas si son distantes de la actual menos de un valor $\alpha\rho(s,s'')$. $\rho(s,s'')$ mide la distancia entre s y s'' y α es un parámetro que cuantifica la importancia de la distancia entre las dos soluciones en el criterio de aceptación.

Concluyendo, *VNS* y sus variantes han sido aplicadas con éxito en varios problemas de optimización combinatoria: como MAXSAT y problemas gráficos [HM01].

5. Métodos basados en población

En los métodos basados en población tratamos en cada iteración del algoritmo con un conjunto (una población) en vez de manipular una solución simple. Al trabajar con una población de soluciones, este tipo de algoritmos proporcionan una forma de exploración del espacio de búsqueda natural e intrínseca a esa agrupación de soluciones que viaja en la búsqueda. De esta forma, el rendimiento final del algoritmo depende fuertemente de la forma en que se manipula a la población. Los métodos basados en población más estudiados en optimización combinatoria son Computación Evolutiva (EC) y Optimización a partir de Colonias de Hormigas (ACO). En los algoritmos de Computación Evolutiva se modifica una población de individuos a partir de operadores de recombinación y mutación, y en Optimización a partir de Colonias de Hormigas se usa una colonia de hormigas artificiales para construir soluciones guiada por los rastros de feromona e información heurística.

5.1. Computación evolutiva (EC)

Los algoritmos de *Computación Evolutiva* (EC) están inspirados en la capacidad de la naturaleza para evolucionar los seres vivientes bien adaptados a su entorno. Los algoritmos de *Computación Evolutiva* pueden ser caracterizados a grandes rasgos como modelos computacionales de procesos evolutivos. En cada iteración se aplica un número de operadores a los individuos de la población actual para generar los individuos de la población de la siguiente

generación (iteración). Normalmente, tenemos operadores para recombinar dos o más individuos para producir nuevos individuos llamados operadores de recombinación o cruce, y tenemos operadores que causan una autoadaptación de los individuos denominados operadores de mutación. La fuerza de prosperidad en los algoritmos evolutivos es la selección de individuos basada en su fitness (esto puede ser el valor de una función objetivo o el resultado de un experimento de simulación, o alguna otra clase de medida). Los individuos con un fitness más alto tienen una probabilidad más alta de ser elegidos como miembros de las siguientes iteraciones de la población (o como padres para la generación de nuevos individuos). Esto corresponde al principio de supervivencia del más adaptado en la evolución natural. Es la capacidad de la naturaleza para adaptarse a un entorno cambiante la que dio la inspiración para los algoritmos de *Computación Evolutiva* (EC).

Ha habido una variedad de algoritmos de *Computación Evolutiva* (EC) ligeramente diferentes propuestos a lo largo de los años. Básicamente éstos se encuadran dentro de tres categorías diferentes que han sido desarrolladas de manera independiente las unas de las otras. Estos son Programación Evolutiva (EP) desarrollada por Fogel et al. en 1966 [Fog62] [FOW66], Estrategias Evolutivas (ES) propuestas por Rechenberg en 1973 [Rec73] y Algoritmos Genéticos iniciados por Holland en 1975 [Hol75] (ver [Gol89] y [Mit98] para más literatura). La Programación Evolutiva emana del deseo de generar inteligencia para la máquina. Mientras que la Programación Evolutiva fue propuesto originalmente para operar en representaciones discretas de máquinas de estado finitas, la mayoría de las variantes actuales son usadas para problemas de optimización continua. Los más recientes tienen cabida también para la mayoría de las variantes actuales de Estrategias Evolutivas, mientras que la mayoría de las aplicaciones de los Algoritmos Genéticos son resolver problemas de optimización combinatoria. A lo largo de los años ha habido muchos estudios y trabajos acerca de métodos de *Computación Evolutiva*.

En lo siguiente, nos concentraremos en los algoritmos de *Computación Evolutiva* orientados a la resolución de problemas de optimización combinatoria y en concreto en aquellos que hemos utilizado para la resolución del problema del que nos ocupamos en este trabajo: **El Problema de Resolución de Restricciones Geométricas – Selección de la Solución deseada**.

Para hacer esto seguiremos un trabajo realizado por Hertz et al. [HK00], que da una buena visión de los diferentes componentes de los algoritmos de *Computación Evolutiva* y de las posibilidades para definirlos. A continuación mostramos la estructura básica de todo algoritmo de *Computación Evolutiva*…

Procedimiento Computación Evolutiva (EC)

```
P ← GeneraPoblacionInicial();
Evalua(P);
mientras (no alcancen las condiciones de terminacion)
hacer
{
      P' ← Recombinar(P);
      P''← Mutar(P');
      Evaluar(P'');
      P  ← Seleccionar(P'' ∪ P);
}
```

Figura 11.: Esquema básico de Algoritmo de Computación Evolutiva

En este algoritmo, *P* denota la población de individuos. Una población de hijos es generada por operadores de mutación y recombinación y los individuos para la siguiente población son seleccionados de la unión de la población antigua y de la población de hijos. Los principales rasgos de un algoritmo de *Computación Evolutiva* son:

- ***Descripción de los individuos:*** los algoritmos de *Computación Evolutiva* manejan poblaciones de individuos. Estos individuos no son necesariamente soluciones del problema considerado. Pueden ser soluciones parciales, o conjuntos de soluciones, o cualquier objeto que pueda ser transformado en una o más soluciones de una manera estructurada. Lo más comúnmente usado en problemas de optimización combinatoria es la representación de las soluciones como cadenas de bits o como permutaciones de *n* números enteros. También son posibles las estructuras de árbol o estructuras más complejas para problemas de optimización más complejos.

- ***Proceso de evolución:*** En cada iteración tiene que decidirse qué individuos entrarán en la población de la siguiente iteración. Esto es hecho a partir de un esquema de selección. La estrategia de sólo escoger entre los hijos como individuos para la siguiente población es llamada *reemplazamiento generacional*. Es posible transferir individuos de la población actual en la siguiente población; en ese caso, estamos tratando con un proceso de evolución de estado continuo. La mayoría de los algoritmos de *Computación Evolutiva* trabajan con poblaciones de tamaño fijo manteniendo siempre al menos el mejor individuo en la población actual. Es también posible tener un tamaño de población variable. En el caso de un tamaño de población continuamente reducido, la situación donde se deja un único individuo en la población (o no pueden ser encontradas parejas de cruce para ningún miembro de la población) puede ser uno de los criterios de parada del algoritmo.

- ***Estructura de vecindario:*** Una función de vecindario $N_{ec} : I \rightarrow 2^I$ sobre el conjunto de individuos I asigna a cada individuo $i \in I$ un conjunto de individuos $N_{ec}(i) \subseteq I$ a los que se disponen actuar como patrones de recombinación para i para crear descendencia. Si un individuo puede ser recombinado con cualquier otro individuo (como por ejemplo en el Algoritmo Genético básico) hablamos de una población desestructurada, en otro caso hablamos de una población estructurada. Un ejemplo para un algoritmo de *Computación Evolutiva* usando poblaciones estructuradas es el Algoritmo Genético Paralelo propuesto por Mühlenbein [Müh91].

- ***Fuentes de información:*** La forma más común de fuentes de información para crear descendencia (por ejemplo, nuevos individuos) es una pareja de padres (cruce de dos padres). Pero hay también operadores de recombinación que recombinan más de dos individuos para crear un nuevo individuo (cruce multiparental), ver [ERR94]. Los desarrollos más recientes usan incluso estadísticas de población para generar los individuos de la siguiente población. Un ejemplo es el operador de recombinación llamado Cruce Simulado de Bits (BSC) [Sys93] que usa una distribución a lo largo del espacio de búsqueda dada por la población actual para generar la siguiente población.

- ***Infactibilidad:*** Una característica importante de un algoritmo de *Computación Evolutiva* es la forma en la que trata con las soluciones no factibles. Cuando se recombinan individuos, podría ocurrir que el individuo resultante no sea factible. Hay básicamente tres formas diferentes de manejar tal situación. Lo más simple es rechazar los individuos no factibles. Por otro lado, para muchos problemas podría

ser muy difícil encontrar individuos factibles. Por lo tanto, la estrategia de penalizar a los individuos no factibles en la función que mide la calidad de un individuo es a veces más apropiado (o incluso inevitable). La tercera estrategia consiste en intentar reparar una solución no factible (ver [ER97] para un ejemplo).

- ***Estrategia de instensificación:*** En muchas aplicaciones fue demostrado que usar algoritmos de mejora para mejorar el fitness de los individuos resulta bastante beneficioso. Los algoritmos de *Computación Evolutiva* que usan un algoritmo de búsqueda local para cada individuo de la población son llamados a menudo *Algoritmos Meméticos* [[Mos89], [Mos99]]. Mientras que el uso de una población asegura una exploración del espacio de búsqueda, el uso de las técnicas de búsqueda local ayuda a identificar rápidamente "buenas" áreas en el espacio de búsqueda. Otra estrategia de intensificación es el uso de operadores de recombinación que tratan explícitamente de combinar "buenas" partes de individuos (en vez de, por ejemplo, un cruce simple de cadenas de bits a partir de un punto). Esto también concentra la búsqueda ejecutada por el algoritmo de *Computación Evolutiva* en áreas de individuos con ciertas propiedades "buenas" (ver [[GDK91], [Kem96], [WHP98], [Har99]] como ejemplos).

- ***Estrategia de diversificación:*** Una de las mayores dificultades de los algoritmos de *Computación Evolutiva* (especialmente al aplicar la búsqueda local) es la convergencia prematura hacia soluciones subóptimas. El mecanismo más común para diversificar el proceso de búsqueda es el uso de un operador de mutación. La simple forma de un operador de mutación supone ejecutar una perturbación aleatoria de un individuo, introduciendo un tipo de ruido. Una forma más elaborada de aplicar diversificación es la de introducir sistemáticamente nuevos individuos provenientes de áreas del espacio de búsqueda no exploradas apenas. Esto se puede lograr a partir de la construcción de individuos en base a alguna memoria de historia que haya guardado pista de por ejemplo la frecuencia con que aparecen ciertos componentes de las soluciones en las soluciones o individuos de generaciones pasadas.

Así acaba la lista de características del algoritmo de *Computación Evolutiva*. Esta lista por supuesto no es exhaustiva y pueden ser válidas también otras formas de describir los algoritmos de *Computación Evolutiva*. En las siguientes secciones vamos a introducir algunos de los métodos basados en poblaciones existentes, concretamente los que hemos utilizado para estudiar el problema correspondiente a este trabajo.

5.2. Algoritmos Genéticos (AG)

Fundamentos de Algoritmos Genéticos

Los *Algoritmos Genéticos* son procedimientos adaptativos para la búsqueda de soluciones en espacios complejos, inspirados en los procesos de evolución natural y evolución genética [HLV98]. La idea básica consiste en mantener una población de cromosomas o individuos, donde cada cromosoma es una solución candidata a un problema concreto. Asociado a cada cromosoma existe un valor de bondad o adaptación que describe la adecuación al problema de la solución que representa. La población evoluciona con el tiempo por medio de un proceso de competición y variación controlada. El procedimiento de competición, denominado mecanismo de selección, utiliza las adaptaciones de los cromosomas para determinar cuáles de

ellos se usarán para crear otros nuevos, mientras que estos nuevos cromosomas se obtendrán a través de la aplicación de operadores genéticos sobre los cromosomas seleccionados en la competición.

Una de las principales ventajas que presenta este tipo de algoritmos es su capacidad para explotar la información acumulada sobre un espacio de búsqueda y, de este modo, dirigir las siguientes búsquedas hacia los mejores subespacios. Por esto se aplican sobre espacios grandes, complejos y parcialmente definidos donde las técnicas clásicas de búsqueda no son apropiadas.

Cuando se trabaja con *Algoritmos Genéticos* se emplea un vocabulario muy específico que hace uso de términos propios de la genética. Así, se denominan genotipos a los propios cromosomas, fenotipos a las soluciones representadas por los cromosomas, genes a las unidades de un cromosoma, loci a las posiciones de los cromosomas y alelos a los posibles estados de un gen.

Estructura de un Algoritmo Genético

Un algoritmo genético comienza con una población de cromosomas generados aleatoriamente, y va obteniendo mejores cromosomas gracias a la aplicación de los operadores genéticos. La evolución se produce sobre la población en forma de selección natural. Durante sucesivas iteraciones, denominadas generaciones, se evalúa la adaptación o adecuación de los cromosomas como soluciones, y en base a esta evaluación se forma una nueva población de cromosomas usando un mecanismo de selección y operadores genéticos específicos, tales como el cruce y la mutación. Para calcular el valor de adaptación de un cromosoma es necesaria una función de adaptación. Todo problema a resolver debe proporcionar una función de evaluación o adaptación que devuelva, para cada cromosoma, un valor numérico proporcional a la utilidad o adaptación de la solución que representa. En resumen, en cada generación se llevan a cabo los tres pasos siguientes:

1. Evaluación de los individuos de la población.

2. Formación de una población intermedia a través del mecanismo de selección, en función de la adaptación de cada cromosoma.

3. Formación de una nueva población a partir de los operadores genéticos de cruce y mutación.

Estos tres pasos se repiten hasta que el sistema deja de mejorar o hasta que se alcanza un número máximo de generaciones especificado por el usuario.

La estructura de un algoritmo genético es tal y como se presenta en la siguiente figura:

Procedimiento AG básico

```
(1) t←0;
(2) Inicializar la población P(t);
(3) Evaluar P(t);
(4) t ← t + 1;
(5) Seleccionar P(t) desde P(t-1);
(6) Aplicar cruce y mutación sobre P(t);
(7) Evaluar P(t);
(8) Si se cumple la condición de parada, terminar. Si
    no, ir al paso 4.
```

Figura 12.: Estructura de un algoritmo genético básico

Representación de cromosomas

Debido a que los *Algoritmos Genéticos* utilizan una representación codificada del problema, la elección de una representación adecuada al problema que se maneja se convierte en un elemento clave del funcionamiento del algoritmo. Con frecuencia se ha utilizado la codificación binaria, donde los cromosomas son cadenas de bits (cadenas de ceros y unos). Éstos son los llamados *Algoritmos Genéticos* con codificación binaria. Pero además de la codificación binaria son múltiples las representaciones empleadas, entre ellas vectores de números reales, vectores de números enteros, listas ordenadas o expresiones representadas como árboles.

Mecanismo de selección

Si consideramos una población *P* con los cromosomas $c_1,\ldots,c_n$, al aplicar el mecanismo de selección sobre *P* obtendremos una población intermedia, P'. Esta población contendrá copias de cromosomas de *P*, donde el número de copias de cada cromosoma dependerá de su valor de adaptación. Aquellos cromosomas que tengan un valor de evaluación alto tienen mayor probabilidad de contribuir con copias para P'. Sobre esta población intermedia se aplicarán posteriormente los operadores genéticos. El procedimiento consta de dos pasos:

1. Para cada cromosoma c_i de la población *P* calcular su probabilidad asociada $p(c_i)$. El método más utilizado para calcular esta probabilidad es el modelo proporcional, donde $p(c_i)$, i =1,…,N se calcula como:

$$p(c_i) = \frac{f(c_i)}{\sum_{j=1}^{N} f(c_j)} \qquad (10)$$

   donde $f(c_i)$ es el valor de la función de evaluación para el cromosoma c_i. De esta manera, los cromosomas con función de evaluación por encima de la media reciben más copias que aquellos con función de evaluación por debajo de la media.

2. Una vez calculadas las probabilidades anteriores, se asigna aleatoriamente a cada elemento de *P* el número de copias suyas que aparecerán en *P*'. Este paso se lleva a cabo mediante un método de muestreo. El más simple, denominado muestreo aleatorio simple, consiste en simular el comportamiento de una ruleta en la que existe para cada individuo de la población una región proporcional al valor de su probabilidad de selección. Cada vez que se lanza, la ruleta determinará un cromosoma para la población intermedia, con lo que tras lanzar N veces se completa la población intermedia. Aquellos cromosomas con mayor probabilidad tendrán asignado en la ruleta una proporción de espacio mayor, lo que significa que la ruleta se detendrá con mayor probabilidad en los espacios correspondientes a tales individuos.

Como complemento al mecanismo de selección puede emplearse una estrategia denominada elitismo. Mediante tal técnica se asegura que el mejor individuo de la actual generación estará presente en la siguiente, ya que es muy posible que el mejor cromosoma de la generación en curso desaparezca en la siguiente al aplicar el mecanismo de selección o los operadores genéticos.

Operadores genéticos

Una vez obtenida la población intermedia tras llevar a cabo el mecanismo de selección se aplican sobre sus cromosomas los operadores de cruce y mutación.

Operador de cruce

El operador de cruce causa la recombinación del material genético de los cromosomas padres, es decir, el cruce combina las características (genes) de dos cromosomas padres para generar uno o dos cromosomas hijos. Es por esto por lo que al operador de cruce también se le llama operador de recombinación. Una propiedad importante del cruce es que explota el espacio de búsqueda asociado a los cromosomas padres. Las definiciones para los operadores de cruce (y como veremos posteriormente, para los operadores de mutación), dependen de la representación escogida para el problema. Por último, hay que tener en cuenta la denominada probabilidad de cruce, que establece la probabilidad de aplicar el operador de cruce sobre parejas de cromosomas de la población intermedia.

Operador de mutación

El operador de mutación altera arbitrariamente uno o más componentes (genes) de un cromosoma para aumentar la variabilidad estructural de la población. El papel de la mutación en los algoritmos genéticos es restaurar material genético perdido o no explorado en la población. De esta forma, se asegura que la probabilidad de alcanzar cualquier punto del espacio de búsqueda nunca es nula. Cada gen en cada cromosoma sufre una mutación de acuerdo con un parámetro de control, denominado probabilidad de mutación, p_m. En este caso ocurre también que la definición del operador de mutación dependerá de la representación de cromosomas seleccionada para la resolución del problema tratado.

Tipos de Algoritmos Genéticos

Podemos distinguir dos modelos dentro de los *Algoritmos Genéticos*:

- *Modelo generacional o clásico*: Durante cada generación se crea una población completa con nuevos individuos mediante la selección de padres de la población anterior y la aplicación de los operadores genéticos sobre ellos. La nueva población reemplaza directamente a la antigua.

- *Modelo estacionario:* Durante cada generación se escogen dos padres de la población (usando muestreo aleatorio simple o cualquier otro tipo de muestreo) y se le aplican los operadores genéticos. Los dos nuevos cromosomas (o el único cromosoma, dependiendo de si se obtiene uno o dos hijos) reemplazan a dos cromosomas (o uno) de la población, que suelen ser los dos (o uno) peores, es decir, con peor adaptación. Este esquema produce una presión selectiva alta cuando se reemplazan los peores, lo que hace que se converja muy rápidamente. Para evitarlo existen diferentes alternativas, como escoger de forma aleatoria los individuos o reemplazar los individuos más antiguos de la población.

Ventajas e inconvenientes de los Algoritmos Genéticos

Los *Algoritmos Genéticos* se han aplicado a un amplio rango de problemas pertenecientes a campos tan diversos como la robótica, ingeniería, inteligencia artificial o economía debido a que presentan las siguientes características:

- Pueden resolver problemas difíciles de forma rápida y fiable.

- Aprenden a mantener o eliminar posibles soluciones en función de su calidad.

- Son ciegos, en el sentido de que no manejan ningún tipo de información sobre el problema concreto, exceptuando la función de evaluación.

- Aunque no hay garantía de encontrar la solución óptima, generalmente encuentran soluciones aceptables.

- Se pueden hibridar fácilmente.

Sin embargo, existe un problema grave asociado a los *Algoritmos Genéticos*, el problema de la convergencia prematura hacia zonas del espacio de búsqueda que no contienen el óptimo global. Esto se debe a que la base del funcionamiento de los *Algoritmos Genéticos* reside en el mantenimiento del equilibrio entre explotar lo que actualmente es mejor (mediante el cruce) y explorar posibilidades que pueden convertirse en algo mucho mejor (haciendo uso de la mutación). Si este equilibrio se desproporciona, se llega a la pérdida de diversidad de la población y, consecuentemente se converge prematuramente. Una de las propuestas para solucionar este problema es la de la utilización de nichos.

Algoritmo de evolución CHC

Como sabemos, la diversidad está asociada a las diferencias entre los cromosomas en la población. La falta de diversidad genética supone que todos los individuos de la población son parecidos. Así mismo, el discurrir del algoritmo será conducido a una convergencia prematura a óptimos locales. En la práctica es algo irreversible, pero existen soluciones: inclusión de mecanismos de diversidad en la evolución y reinicialización cuando se produce convergencia prematura, entre otras.

A continuación, veremos uno de los algoritmos utilizados en nuestro estudio que tiene su base en los *Algoritmos Genéticos* pero que supone una mayor elaboración, intentando establecer un equilibrio entre diversidad y convergencia: el algoritmo de evolución *CHC*. Es importante considerar que *CHC* parte de una representación binaria.

El objetivo fundamental del *CHC* es el de combinar una selección elitista que preserva los mejores individuos que han aparecido hasta el momento con un operador de cruce que produce hijos muy diferentes a sus padres. Introduce cuatro componentes novedosas:

- *Selección Elitista:* selecciona los N mejores cromosomas entre padres e hijos. Los N mejores elementos encontrados hasta el momento permanecerán en la población actual.

- *Cruce Uniforme HUX.* Intercambia exactamente la mitad de los alelos que son distintos en los padres. Garantiza que los hijos tengan una distancia Hamming máxima a sus dos padres.

- *Prevención de Incesto.* Se forman N/2 parejas con los elementos de la población. Sólo se cruzan las parejas cuyos miembros difieren en un número determinado de bits (umbral de cruce). El umbral se inicializa a L/4 (L es la longitud del cromosoma). Si durante un ciclo no se produce ni un solo cruce, al umbral de cruce se le resta uno.

- *Reinicialización.* Cuando el umbral de cruce es menor que cero, la población se reinicializa: a) usando el mejor elemento como plantilla (35% de variación aleatoria) e incluyendo una copia suya, o b) manteniendo el mejor o parte de los mejores de la población y el resto aleatorio.

CHC no aplica el operador de mutación.

A continuación para finalizar mostramos en pseudocódigo el algoritmo básico asociado al CHC…

Procedimiento CHC

```
t←0;
d←L/4;
inicializar(P(t));
evaluar estructuras en P(t);
mientras (no se satisfaga la condición de terminación)
hacer
{
      t←t+1;
      selección individuos a cruzar en C(t) desde P(t-1);
      recombinar estructuras de C(t) para formar C'(t);
      evaluar estructuras de C'(t);
      selección nueva población P(t) entre C'(t) y P(t-1);
      si P(t) es igual que P(t-1) entonces
            d←d-1;
      si d<0 entonces
      {
            reinicializar(P(t));
            d←r*(1.0-r)*L;
      }
}
```

Figura 13.(a): Estructura del algoritmo CHC básico

Procedimiento Selección individuos a cruzar

```
      Copiar todos los miembros de P(t-1) en C(t)
aleatoriamente;
```

Procedimiento Selección nueva población a partir de padres e hijos

```
      Obtener P(t) a partir de P(t-1)
            al reemplazar los peores miembros de P(t-1)
            con los mejores miembros de C'(t)
      hasta que no queden miembros de C'(t)
      que sean mejores que ningún miembro restante de P(t-
1);
```

Procedimiento Recombinación

```
      para cada una M/2 parejas de estructuras de C(t)
hacer
      {
            determinar la distancia de Hamming;
            si(distancia_Hamming/2) > d entonces
                  intercambiar la mitad de los bits
                  diferentes aleatoriamente;
            sino
                  borrar la pareja de estructuras de C(t);
      }
```

Procedimiento Reinicialización

```
Reemplazar P(t) con M copias del mejor miembro de
P(t-1);
para todos los individuos menos uno de P(t) hacer
{
      cambiar r*L bits aleatoriamente;
      evaluar la estructura o individuo;
}
```

Figura 13.(b): Estructura del algoritmo CHC básico

Algoritmos Genéticos con nichos o multimodales

La necesidad de los Algoritmos Genéticos con nichos o multimodales

Los *Algoritmos Genéticos* son conocidos, como sabemos, por su capacidad para llevar a cabo procesos de búsqueda en espacios complejos. Aún así, las versiones más generales de este tipo de algoritmos pueden no trabajar de un modo adecuado cuando el espacio de búsqueda es multimodal y presenta muchos óptimos locales. En estos casos, los *Algoritmos Genéticos* simples se caracterizan por converger a la zona del espacio donde se encuentra la mejor solución o los mejores óptimos locales, abandonando la búsqueda en las zonas restantes. Este fenómeno se denomina deriva genética y se debe evitar por dos razones principales:

- En primer lugar, esta convergencia hacia el que se cree el óptimo más prometedor del espacio puede dar lugar a que éste no se encuentre en esa zona y el algoritmo genético no sea capaz de saltar a aquella en la que realmente está situado.

- Por otro lado, debido a esta forma de trabajo, la necesidad que existe en algunos problemas de conocer la localización de varios óptimos de la función no puede ser satisfecha por el algoritmo genético.

De los dos comportamientos descritos, nos centraremos en la práctica exclusivamente en el primero de ellos.

Para entender el porqué de la necesidad de los nichos en los *Algoritmos Genéticos*, veamos lo que ocurre cuando lanzamos un algoritmo genético sobre un problema multimodal en el que todos los óptimos pueden considerarse como globales.

Si ejecutamos el algoritmo con una población inicial elegida aleatoriamente, obtenemos una serie de puntos difuminados a lo largo del dominio de toda la función. Conforme se va produciendo la reproducción, el cruce y la mutación, la población empieza a escalar los picos y al final la mayoría de los individuos estarían dispuestos alrededor de la cima de uno de los picos. Esta convergencia a uno solo de los picos está producida por la deriva genética. De un modo u otro, nos gustaría que se formaran subpoblaciones estables a lo largo de cada posible solución.

Con un algoritmo genético estándar, lo único que habríamos conseguido hubiera sido que todos los individuos de la población se hubieran concentrado alrededor de una de las soluciones, como podemos ver en la figura siguiente…

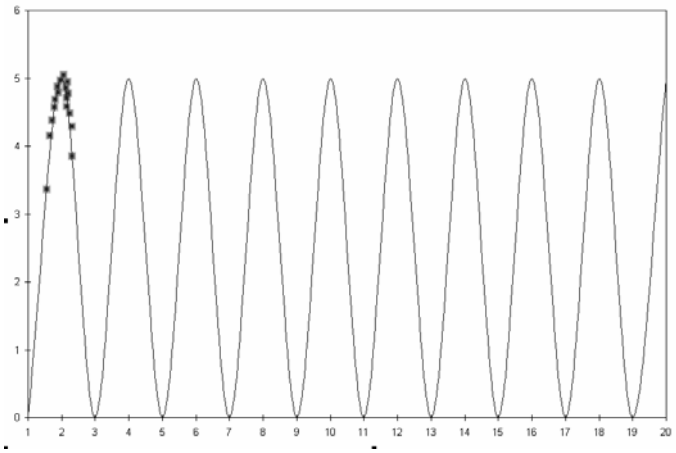

Figura 14.: Distribución de las soluciones en un espacio de búsqueda multimodal uniforme en una iteración avanzada de la ejecución del AG básico sobre un problema.

Efectivamente, obtenemos el valor máximo de la función pero esto se debe a que todos los picos toman ese valor máximo y además perdemos el resto de soluciones que nos lo proporcionan, en el caso de que estuviéramos interesados en ellas.

Podemos ver en la siguiente figura cómo utilizando la técnica de nichos con *Algoritmos Genéticos*, formamos subpoblaciones a lo largo de cada uno de los picos de la función y que finalmente van evolucionando hacia el máximo valor de cada uno de estos entornos. De esta forma lo que hacemos es obtener todas las soluciones del problema, que en este caso toman todas el mismo valor.

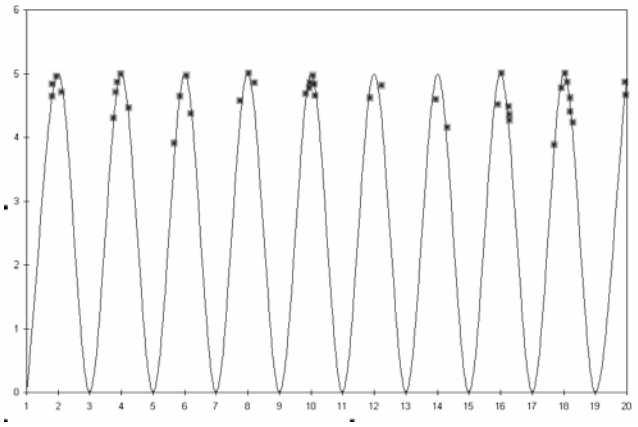

Figura 15.: Distribución de las soluciones en un espacio de búsqueda multimodal uniforme en una iteración avanzada de la ejecución de un AG multimodal sobre un problema.

De igual modo, nos gustaría obtener las diferentes soluciones o picos de un problema aún en el caso de que estos no sean de la misma magnitud. Ahora consideremos una función con varios óptimos y solo uno de ellos el global.

En este problema tenemos varios picos, pero estos decrementan en magnitud a la vez que el espacio de búsqueda avanza en su amplitud. El comportamiento de un algoritmo genético sería fácil de predecir.

Un algoritmo genético básico, ejecutadas las suficientes generaciones, distribuirá la mayoría de sus individuos a lo largo del mayor pico. Se podría producir la siguiente situación:

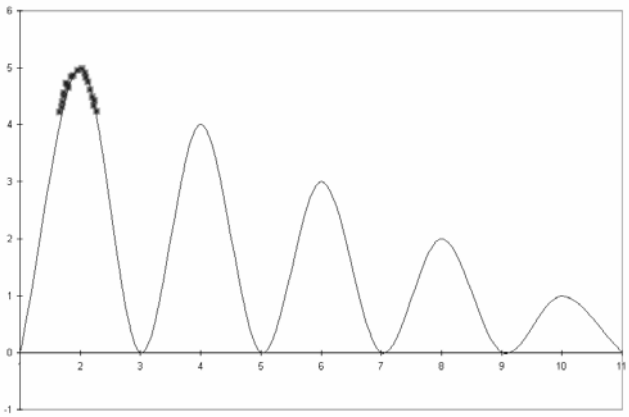

Figura 16.: Distribución de las soluciones en un espacio de búsqueda multimodal no uniforme en una iteración avanzada de la ejecución del AG básico sobre un problema.

En este caso, todos los individuos de la población se han concentrado alrededor de la mejor solución. Hemos perdido el resto de soluciones que quizás nos podrían interesar, pero además el hecho de que hayamos alcanzado el óptimo global ha podido ser hasta casual. Si la población inicial se hubiera distribuido de otra manera, nada habría impedido que el algoritmo genético hubiera evolucionado hacia otra posible solución que nos habría proporcionado un óptimo local.

El uso de nichos a la hora de afrontar el problema multimodal nos habría llevado a una situación parecida a la siguiente.

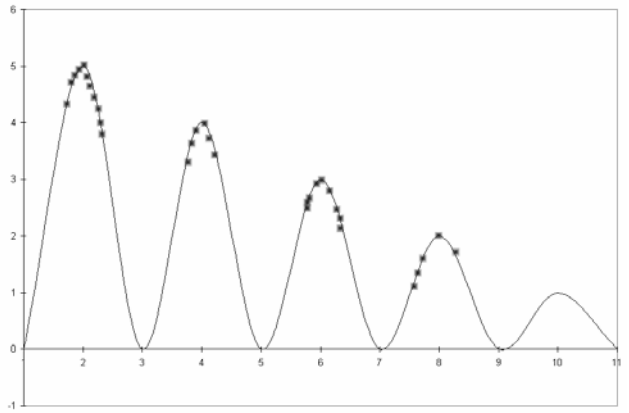

Figura 17.: Distribución de las soluciones en un espacio de búsqueda multimodal no uniforme en una iteración avanzada de la ejecución de un AG multimodal sobre un problema.

Ahora hemos conseguido distribuir todos los individuos de la población en torno a cada uno de los óptimos del problema (tanto locales como globales). Gracias a esto tenemos todas las soluciones del problema, incluida la mejor y al mantener subpoblaciones estables evitamos el problema de que el algoritmo pueda converger hacia un óptimo local y nos sea imposible evolucionar hacia la mejor solución.

Algoritmos Genéticos con nichos

Los métodos de nichos fueron desarrollados para reducir el efecto de la deriva genética resultado de la selección en el algoritmo genético estándar [KG89], [Gol89], [Hol75]. Los trabajos de investigación basados en la preservación de la diversidad mediante técnicas de nichos en *Algoritmos Genéticos* han dado resultados prometedores [SK98]. Los nichos mantienen la diversidad de la población y permiten al algoritmo genético investigar en diferentes espacios en paralelo. Por otro lado, previenen que el algoritmo genético quede atrapado en óptimos locales del espacio de búsqueda.

Los *Algoritmos Genéticos* con nichos están basados en mecanismos presentes en ecosistemas naturales. En la naturaleza, los animales compiten por su supervivencia cazando, alimentándose, reproduciéndose, etc., y diferentes especies evolucionan para ocupar diferentes roles. Un nicho puede verse como un subespacio en el entorno que soporta diferentes tipos de vida o como una tarea que lleva a cabo un individuo en el entorno. Una especie puede definirse como un grupo de individuos con similares características biológicas capaz de interrelacionarse entre ellos pero que no son capaces de relacionarse con individuos fuera de su grupo. Para cada nicho, los recursos son finitos y deben compartirse con toda la población de ese nicho.

Por analogía, los métodos con nichos tienden a lograr una aparición natural de los nichos y las especies en el entorno (espacio de búsqueda). El concepto de nicho es normalmente referido a un óptimo del dominio, y el fitness representa los recursos de ese nicho. Las especies pueden ser definidas como individuos similares en términos de métricas similares.

A modo de resumen, los *Algoritmos Genéticos con nichos* se basan en provocar la formación de subpoblaciones de individuos estables (especies) que exploran zonas parciales del espacio de búsqueda (nichos), obligando a los individuos similares a compartir los recursos disponibles entre ellos.

Tipos de Algoritmos Genéticos con nichos

Se establecen distintos tipos de *Algoritmos Genéticos con nichos*. A continuación veremos los constatados normalmente como más efectivos y que hemos utilizado en este trabajo para nuestros experimentos…

Los **métodos de proporción** (Fitness sharing) [GR87], [OGC91] son probablemente las técnicas de nichos más ampliamente usadas. Con el método de proporción se pretende lograr la formación de elementos vecinos (nichos), asociando cada uno de ellos con un óptimo de la función multimodal. El número de elementos en cada nicho es proporcional al valor de su óptimo asociado. A partir del método básico se han desarrollado otros como el **método de proporción continuamente actualizado** (continuously updated sharing) [OGC91] y que pretende evitar algunas limitaciones de la técnica básica.

Los **métodos de aclarado** (Clearing) [Pet96], [Pet97] son quizás unos de los métodos más efectivos de nichos propuestos en la literatura de *Algoritmos Genéticos*, como ya fue indicado por [SK98]. Los métodos de aclarado son muy parecidos a los de proporción salvo que están basados en el concepto de recursos limitados del entorno. En lugar de compartir los recursos entre todos los individuos de una subpoblación como en los métodos de proporción, el aclarado lo hace tan solo con los mejores individuos de la subpoblación.

Los **métodos de multitud** (Crowding methods) [Mah92] pretenden mantener subpoblaciones estables dentro de la población reemplazando elementos de la población por individuos parecidos. Se aplican para localizar y preservar soluciones múltiples para una función multimodal, o preservar la diversidad para reducir la posibilidad de convergencia prematura. Una modificación sobre este mismo método es el de **multitud con multinichos**

(multiniche crowding) [CVS95] en el que cambia el modo de selección de los individuos y de reemplazo.

El último de los métodos estudiados es el **método de nichos elitista** (elitist niching system) [PHH01]. Se realiza después de la aplicación de los operadores genéticos y crea una nueva población con los mejores elementos que pertenecen a los nichos representados en el conjunto de padres y descendientes. Además, se realiza una selección de los elementos restantes para completar la formación de la nueva población. Esta selección se lleva a cabo mediante un modelo de ranking lineal [Bak85] con una presión selectiva débil. De esta manera, se introduce un alto nivel de explotación mediante la inclusión del mejor elemento de cada nicho y se favorece la exploración debido a la selección de los elementos restantes. Esta nueva población se someterá al proceso de selección (basado en el método proporcional) y a los operadores genéticos.

Método de proporción (fitness sharing)

El método de proporción modifica el espacio de búsqueda reduciendo el valor de fitness en las regiones densamente pobladas. El número de elementos en cada nicho es proporcional al valor de su óptimo asociado. La nueva función de fitness decrementa el fitness de los elementos de la población de acuerdo al número de individuos similares de la población. Típicamente, el fitness proporcional f_i' de un individuo i con fitness f_i es simplemente…

$$f_i' = \frac{f_i}{m_i} \qquad (11)$$

En la fórmula anterior m_i es el cálculo del nicho que mide el número aproximado de individuos con que el fitness f_i es proporcional.

El cálculo del nicho se obtiene sumando la función de proporción sobre todos los miembros de la población…

$$m_i = \sum_{j=1}^{N} sh(d_{i,j}) \qquad (12)$$

donde N denota el tamaño de la población y $d_{i,j}$ representa la distancia entre el individuo i y el individuo j. Por consiguiente, la función de proporción (sh) mide el nivel de similaridad entre dos elementos de la población. Devuelve uno si los elementos son idénticos, cero si su distancia $d_{i,j}$ es mayor que un umbral de diferencia, y un valor intermedio a un nivel intermedio de diferencia. La función de proporción más ampliamente usada es la siguiente…

$$sh(d_{i,j}) = \begin{cases} 1-(d_{i,j}/\sigma_s)^{\alpha} & \text{si } d_{i,j} < \sigma_s \\ 0 & \text{en otro caso} \end{cases} \qquad (13)$$

donde σ_s denota el umbral de diferencia (o también el radio del nicho) y α es una constante que regula la forma de la función de proporción. Normalmente α es fijada a uno.

La distancia $d_{i,j}$ entre dos elementos i y j está caracterizada por una métrica de similaridad que puede estar basada tanto en una distancia genotípica como fenotípica. La distancia genotípica está relacionada con la representación de la cadena y generalmente es una distancia de Hamming. En cambio, la distancia fenotípica se define usando conocimiento específico del fenotipo del problema. Podría ser por ejemplo la distancia euclídea.

Los métodos de proporción tienden a fomentar la búsqueda en regiones no exploradas del espacio y favorecen la formación de subpoblaciones estables.

Sin embargo, el método de proporción no está exento de limitaciones…

- La estimación del umbral de diferencia σ_s, requiere un conocimiento a priori de lo alejado que estamos del óptimo. Sin embargo, para la optimización de problemas reales, no suele haber información disponible sobre el espacio de búsqueda y la distancia entre los óptimos. Por otro lado, σ_s es el mismo para todos los individuos. Esto supone que todos los picos solución deben estar equidistantes en el dominio. Por estas razones, los métodos de proporción pueden fracasar al mantener todas las cimas deseadas si no están equidistantes o si la estimación de la distancia entre dos picos es incorrecta. Se han propuesto algunas fórmulas empíricas para fijar el umbral de diferencia [KG89], [Mah95].

- El esquema de proporción es muy costoso computacionalmente y nos supone un orden $O(n^2)$ por generación. Se han desarrollado algunos métodos de análisis de clusters y proporción dinámicos que reducen la complejidad computacional e incrementan la efectividad del método de proporción [YG93], [MS96].

Así pues, la función de fitness proporcional de un elemento decrece en razón al número de elementos pertenecientes a su nicho contenidos en la población en un momento dado, y sin embargo, crece en razón al valor de su función de evaluación. Si un nicho tiene muchos representantes en un momento dado y además éstos tienen alta adaptación, la función de fitness proporcional de cada uno de ellos decrece, con lo que se propicia su pérdida. De esta forma, se asegura que en cada nicho exista un número proporcional de elementos con respecto al valor del óptimo asociado con éstos.

Método de aclarado (clearing method)

El método de aclarado es muy similar al de proporción pero está basado en el concepto de recursos limitados del entorno [Pet96], [Pet97].

El procedimiento de aclarado se aplica después de evaluar el fitness de los individuos y antes de aplicar el operador de selección. Como en los métodos de proporción, el algoritmo de aclarado usa una medida de diferencia entre los individuos para determinar si pertenecen a la misma subpoblación o no. Este valor podría ser la distancia de Hamming para codificaciones binarias, la distancia euclídea para codificaciones reales o podría ser definida a nivel fenotípico.

Cada subpoblación contiene un individuo dominante: aquel que tiene el mejor fitness. Si un individuo pertenece a una subpoblación, entonces su diferencia con el dominante es menor de un umbral dado σ: el radio de aclarado. El algoritmo de aclarado básico preserva el fitness del individuo dominante mientras que fija el fitness del resto de individuos de la misma subpoblación a cero. Así, el procedimiento de aclarado atribuye todos los recursos del nicho a un único individuo: el ganador.

Con este mecanismo, el nicho de un individuo generalmente no es conocido. En efecto, puede ser dominado por varios ganadores. Por otro lado, para una población dada, el conjunto de ganadores es único. Esta proposición es probada por inducción: el individuo que tiene el fitness más fuerte en la población es necesariamente el ganador. El ganador y todos los individuos que domina son ficticiamente eliminados de la población. Procedemos de la misma

manera con la nueva población que obtenemos. Así, la lista de todos los ganadores se produce después de un cierto número de pasos.

Es posible generalizar el algoritmo de aclarado aceptando varios ganadores escogidos de entre los mejores individuos de cada nicho. La capacidad de un nicho es definida como el número máximo de ganadores que este nicho puede aceptar. Notar que si es elegida una capacidad mayor de uno, el conjunto de ganadores para una población dada no es generalmente único. Hay al menos una razón para querer capacidades mayores de uno: si las capacidades son iguales al tamaño de la población, el efecto de aclarado desaparece y el método de búsqueda se convierte en un algoritmo genético estándar. Así, eligiendo capacidades entre uno y el tamaño de la población se obtienen situaciones entre el máximo aclarado y el algoritmo genético estándar.

Una versión simplificada del procedimiento de aclarado es presentada a continuación en pseudocódigo. *P* y *n* son variables globales. *"Sigma"* es el radio de aclarado, *"Kappa"* es la capacidad de cada nicho y *"nbWinners"* indica el número de ganadores de la subpoblación asociada con el nicho actual. La población *P* puede ser considerada como un array de *n* individuos.

Procedimiento de aclarado (Clearing)

```
funcion Clearing(Sigma, Kappa)
{
SortFitness(P);
para i←0...n-1 hacer
{
      si Fitness(P[i])>0 entonces
      {
            nbWinners ← 1;
            para j←i+1...n-1 hacer
            {
              si
              ((Fitness(P[j])>0)Y(Distancia(P[i],P[j])<Sigma
              )) entonces {
                  si nbWinners < Kappa entonces
                        nbWinners ← nbWinners + 1;
                  sino
                        Fitness(P[j])← 0.0;
              }
            }
      }
}
```

Figura 18.: Función de aclarado.

El algoritmo simplificado usa tres funciones:

- `SortFitness(P)` ordena la población *P* de acuerdo al fitness de los individuos por orden descendente.

- `Fitness(P[i])` devuelve el fitness del i-ésimo individuo de la población *P*.

- `Distancia(P[i],P[j])` devuelve la distancia entre dos individuos *i* y *j* de la población *P*.

La complejidad del algoritmo se ha establecido como O(*cn*) donde *c* es el número de subpoblaciones y *n* es el tamaño de la población. Esta complejidad es igual a algunos de los métodos de proporción mejorados. Sin embargo, *c* podría ser del orden de *n* si el número de picos es mayor que el tamaño de la población. El mismo fenómeno ocurre si el radio de aclarado es elegido muy pequeño. En estos casos, la complejidad es idéntica a la complejidad del método de proporción básico, es decir, O(n^2). Una forma de reducir esto es construyendo subpoblaciones mediante un método de clustering jerárquico [Pet97]. En este caso la complejidad sería O(*nlog(n)*).

La estimación correcta del radio de aclarado σ usado en el método básico es un problema difícil. De cualquier modo, el radio de aclarado es solo un modo simple de definir subpoblaciones. Es algo similar al radio de proporción.

Método de multitud (crowding method)

El objetivo de los métodos de multitud [Mah93] es mantener subpoblaciones estables dentro de la población reemplazando elementos de la población por individuos parecidos. Se aplica para localizar y preservar soluciones múltiples para una función multimodal, o preservar la diversidad para reducir la posibilidad de convergencia prematura.

En estos métodos, el proceso de reemplazo es modificado para permitir la formación de nichos en la población. En los *Algoritmos Genéticos* tradicionales, los nuevos individuos creados tras el proceso de reproducción pasan a reemplazar la población entera en cada generación. Por otro lado, en los *Algoritmos Genéticos* estacionarios, cada nuevo individuo creado reemplaza solo uno de la población (generalmente el peor). Cuando este reemplazo se realiza teniendo en cuenta la distancia, estamos trabajando con los métodos de multitud.

En el modelo de multitud determinístico, Mahfoud [Mah92] empareja aleatoriamente todos los elementos de la población durante cada generación. A cada par se le aplica el operador de cruce en combinación con el operador de mutación u otros operadores genéticos para producir dos hijos. Estos hijos compiten contra sus padres para incluirse en la población.

Mediante el reemplazo de individuos similares, los métodos de multitud se esfuerzan en mantener la diversidad preexistente de la población. En cualquier caso, deben prevenirse los errores en los reemplazos para mantener a los individuos en la vecindad de las cimas deseadas. El método de multitud determinístico fue diseñado para minimizar el número de reemplazos erróneos.

El método funciona de la siguiente manera. En primer lugar se agrupan los elementos de la población en μ/2 parejas. Cada pareja se cruza y se mutan los descendientes. Cada descendiente compite contra uno de los padres que lo ha generado. Para cada par de descendientes, son posibles dos torneos entre padres e hijos. El pseudocódigo del algoritmo se establece a continuación: *g* es el número de generaciones, μ el tamaño de la población y *P(g)* la población final.

Procedimiento de multitud (Crowding)

```
funcion Crowding(g,μ)
{
      P(0)←inicializar();
      para t←1...g hacer
      {
            P(t)← mezcla(P(t-1));
            para i←0... (μ/2)-1 hacer
            {
```

```
                    p1←a2i+1(t);
                    p2←a2i+2(t);
                    {c1,c2}← recombinar(p1,p2);
                    c1'←mutar(c1);
                    c2'←mutar(c2);
                   si [d(p1,c1')+d(p2,c2')]≤ [d(p1,c2')+d(p2,c1')]
                    entonces{
                    si F(c1')>F(p1) entonces a2i+1(t)← c1' ;
                    si F(c2')>F(p2) entonces a2i+2(t)← c2' ;
                    }
                    sino{
                    si F(c2')>F(p1) entonces a2i+1(t)← c2' ;
                    si F(c1')>F(p2) entonces a2i+2(t)← c1' ;
                    }
            }
        }
}
```

Figura 19.: Función de multitud.

El método de multitud determinístico requiere que el usuario solo introduzca el tamaño de la población μ y el criterio de parada. El usuario puede parar la ejecución tras un número fijo de generaciones *g* o cuando la mejora de la población se aproxima a cero.

Todo el proceso de competición de los hijos con los padres y el posterior reemplazo puede resumirse fácilmente en el siguiente diagrama…

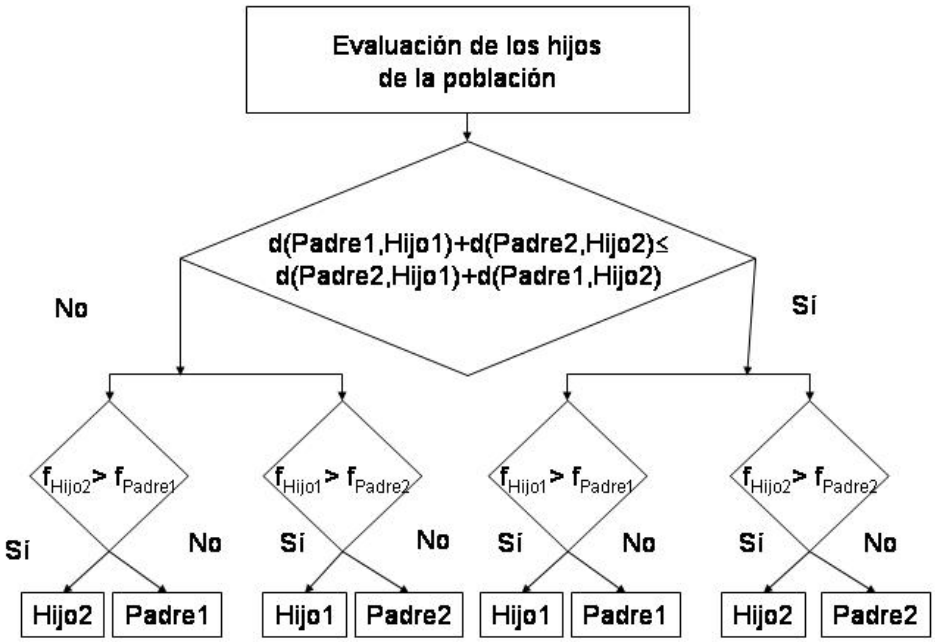

Figura 20.: Diagrama de flujo que refleja el proceso de competición de hijos con padres y posterior reemplazo en el algoritmo de multitud.

Método de proporción continuamente actualizado (Continuously updated sharing)

El método de proporción clásico se basa en la función de fitness de proporción, que decrementa el fitness de los individuos de acuerdo al número de individuos similares de la población.

Los estudios de [SK98] demostraron algunas limitaciones de esta técnica, y es por ello que surge como posible mejora el método de proporción continuamente actualizado [OGC91] que consiste en combinar el torneo de selección con un fitness proporcional.

En lugar de aplicar el torneo por selección a la población que ha recibido los cálculos del fitness proporcional, aplicamos el torneo por selección de acuerdo a los valores de fitness proporcional que ha sido continuamente actualizado usando solo los individuos escogidos para ser miembros de la próxima generación. De este modo, la retroalimentación continua se obtiene con respecto al estado actual de la población, y esta retroalimentación es usada inmediatamente en el cálculo del siguiente fitness proporcional que es usado para determinar si un individuo u otro resulta elegido para formar parte de la población objetivo. Es decir, intentamos crear la nueva generación mirando solo cuántos individuos existen en cada nicho en la población antigua. Lo que intentamos hacer ahora es crear la generación objetivo mirando cuántos individuos existen en la generación que está siendo creada. Esto es, cuando dos individuos compiten, miramos en la nueva generación y vemos cuál tiene proporcionalmente menos individuos que el objetivo, eligiendo aquel individuo que gane este particular torneo.

Hay varias maneras de implementar este método en una aplicación real. Una manera podría ser usando la idea de ajustar la función de fitness con la función de proporción [GR87] y calculando la función de proporción usando la población objetivo. Otro método podría ser introduciendo el parámetro tamaño del nicho n^*. Con este método, podríamos usar la función de proporción usual para determinar el número de individuos en un nicho, pero se podría determinar qué individuos ganan un torneo de la siguiente manera: si ambos nichos a los que los individuos pertenecen tienen menos de n^* miembros, entonces el individuo con mejor fitness gana, si no el individuo con más peligro a desaparecer gana. El método de proporción evoluciona a poblaciones que tienen más individuos en el nicho que tiene un mayor fitness, y el método de nicho con umbral evoluciona a poblaciones con aproximadamente n^* individuos en cada uno de los mejores N/n^* nichos.

El siguiente diagrama de flujo muestra el desarrollo del algoritmo, donde el valor de proporción del cromosoma *j* es actualizado cuando su distancia con el padre seleccionado es menor que σ_{share}.

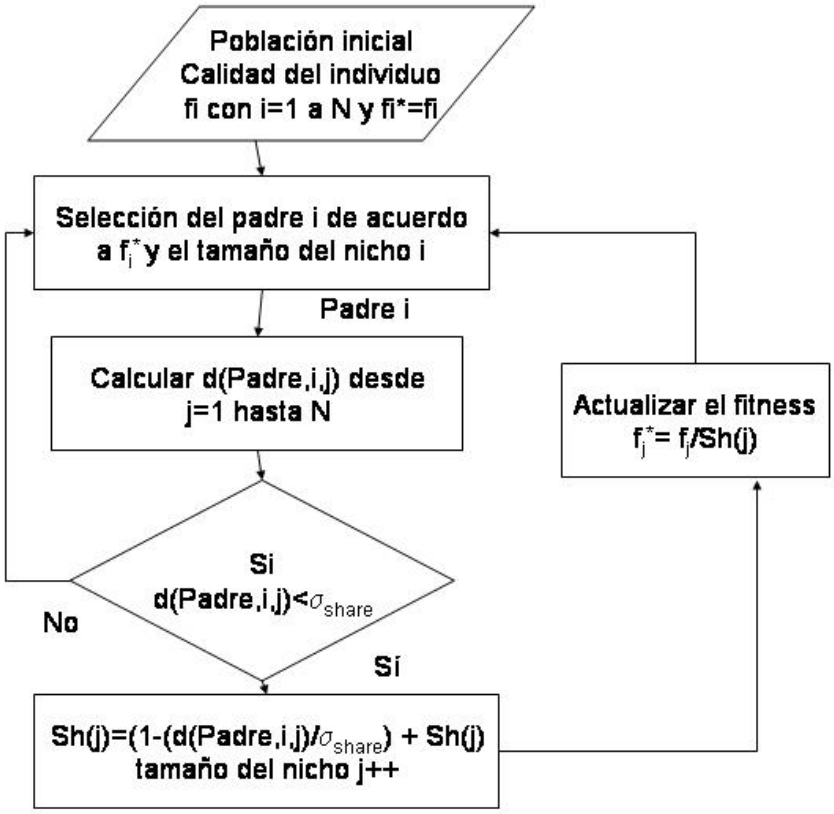

Figura 21.: Diagrama de flujo que refleja el desarrollo del algoritmo de proporción continuamente actualizado.

Método de nichos elitista

El método de nichos elitista, propuesto por Pérez, Herrera, Lozano y del Olmo [PHH01] consiste en un proceso de selección elitista aplicado tras la generación de:

a) la población padre, obtenida de la población actual, $P(t)$, por torneo de selección con el fitness proporcional, y

b) la población descendiente obtenida a través de los operadores genéticos aplicados sobre la población padre.

Se asegura que los mejores cromosomas pertenecientes a cada nicho, representados en el conjunto formado tanto por la población padre como por la descendiente, están presentes en la población para la siguiente generación, $P(t+1)$.

Esto se realiza de la siguiente manera:

1. Los individuos que tienen el mejor fitness en el conjunto de la población de padres y de descendientes son necesariamente los mejores elementos de un nicho individual. Son seleccionados para pertenecer a la nueva población, $P(t+1)$.

2. Estos individuos y todos los elementos que pertenezcan a su nicho son eliminados ficticiamente del conjunto.

3. Se procede de la misma manera con los elementos restantes para encontrar los siguientes mejores elementos de un nicho.

4. Si el número de los mejores elementos representativos obtenidos es mayor o igual que el tamaño de la población (*N*), seleccionamos los *N-1* mejores cromosomas para formar la nueva población. Por otro lado, si este número es menor que el tamaño de la población, elementos adicionales pertenecientes al conjunto de padres e hijos deben ser seleccionados hasta tener los *N-1* cromosomas. Esto se lleva a cabo mediante la aplicación del modelo de selección de ranking lineal [Bak85] sobre este conjunto.

5. Finalmente, usamos el método elitista clásico para añadir el mejor cromosoma de la población $P(t)$ a la población $P(t+1)$.

En cuanto al modelo de selección mediante ranking lineal, los cromosomas son ordenados de acuerdo al fitness tradicional, y entonces la probabilidad de seleccionar cada cromosoma, C_i, es calculada de acuerdo a su ranking, $rank(C_i)$ (con $rank(C_{mejor})=1$), usando la siguiente función de asignación:

$$p(C_i) = \frac{1}{M} \times \left(\eta_{max} - \frac{(\eta_{max} - \eta_{min}) \times (rank(C_i) - 1)}{M - 1} \right) \qquad (14)$$

donde M es el tamaño del conjunto y $\eta_{min} \in [0,1]$ especifica el número de copias esperadas del peor cromosoma (del mejor se esperan $\eta_{max}=2-\eta_{min}$ copias). La presión selectiva de la selección mediante ranking lineal está determinada por η_{min}. Si η_{min} es bajo, se consigue una alta presión, mientras que si es alto, la presión es menor.

El ranking lineal se lleva a cabo mediante un muestreo universal estocástico [Bak87]. Este procedimiento garantiza que el número de copias de cualquier cromosoma está limitado.

Los autores proponen introducir una baja presión selectiva (es decir, un valor de η_{min} alto, tal como $\eta_{min}=0.75$) para generar elevados niveles de diversidad. De este modo, la propuesta induce un alto grado de explotación a causa de la inclusión de los elementos más representativos de los nichos y una alta exploración (que es muy útil cuando tratamos con funciones multimodales) gracias a la selección de los elementos restantes por ranking lineal.

Método de multitud por multinichos

A partir del método de multitud determinístico se han propuesto varias mejoras y modificaciones. Una de ellas es el método de multitud por multinichos propuesta por Cedeño y Vemuri [CVS95].

En los métodos de multitud por multinichos (multiniche crowding o MNC), tanto la selección como el reemplazo son modificados con algún tipo de método de multitud. La idea está en eliminar la presión del proceso de selección causada por el método de selección proporcional a la vez que permite a la población mantener alguna diversidad. Este objetivo es conseguido, en parte, fomentando el emparejamiento y reemplazo con miembros del mismo nicho a la vez que permite la competición entre nichos. El resultado es un algoritmo que…

a) mantiene subpoblaciones estables dentro de diferentes nichos.

b) mantiene diversidad por toda la búsqueda.

c) converge a diferentes óptimos locales.

En los métodos de multitud por multinichos, la selección proporcional es reemplazada por lo que llamamos *selección por multitud*. En la selección por multitud, cada individuo de la población tiene la misma probabilidad de emparejarse en cada generación. La aplicación de esta regla de selección tiene lugar en dos pasos. Primero, un individuo A es seleccionado para emparejarse. Esta selección podría ser tanto secuencial como aleatoria. Segundo, su pareja M es seleccionada, no del total de la población sino de un grupo de individuos de tamaño C_s, elegido aleatoriamente (con reemplazo) de la población. El compañero M así escogido debe ser el más similar a A. La métrica de similaridad usada aquí no es una métrica genotípica como la distancia de Hamming, sino una distancia fenotípica definida convenientemente. La selección por multitud fomenta el emparejamiento entre individuos del mismo nicho aunque permite emparejamientos de individuos de diferentes nichos.

Durante el proceso de reemplazo, el método de multitud por multinichos usa una política de reemplazo llamada *el peor entre los más similares*. La meta de este paso es seleccionar un individuo de la población para ser reemplazado por un descendiente. La implementación sigue los siguientes pasos. Primero se crean C_f grupos seleccionando aleatoriamente s individuos (con reemplazo) por grupo de la población. Estos grupos son llamados *grupos de factor de multitud*. Segundo, se identifica un individuo de cada grupo que es el más similar fenotípicamente al descendiente. Esto nos da C_f individuos que son candidatos al reemplazo en virtud a su similaridad con el descendiente que los reemplazará. De este grupo de individuos más similares, elegimos aquel de peor fitness para morir, y rellenamos ese hueco con el descendiente. El descendiente podría tener peor fitness que el individuo al que ha reemplazado.

Tanto la selección como el reemplazo están primordialmente basados en una métrica de similaridad. El fitness es además considerado durante el reemplazo para promover la competición entre miembros del mismo nicho. También puede se puede dar la competición entre miembros de distintos nichos. El siguiente pseudocógio resume los aspectos más notables de esta técnica…

Algoritmo Genético Multimodal asociado al método de multitud por multinichos

```
Generar una población inicial de N individuos;
para gen=1...MAX_GEN  {
     para i=1...N    {
          Usar la selección por multitud para encontrar
          compañero para el padre i;
          Cruzar y mutar;
          Insertar el hijo en la población usando el
          reemplazo peor entre los más similares;
     }    }
```

Figura 22.: Algoritmo del método de multitud por multinichos.

La política de reemplazo seguida, puede observarse en el siguiente gráfico…

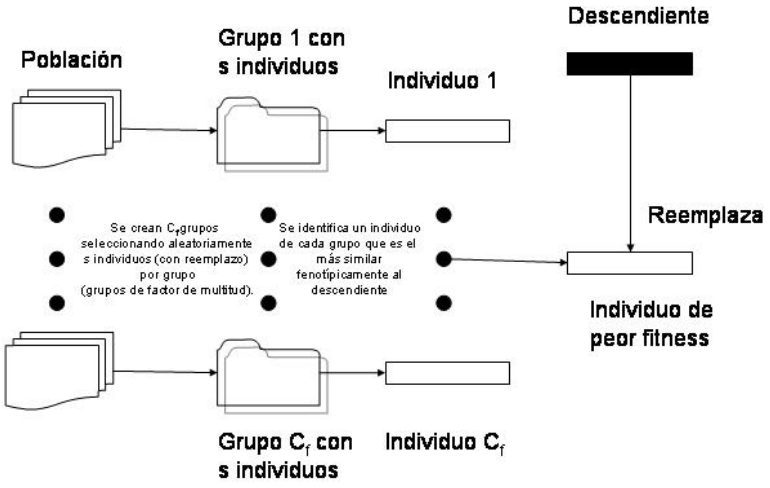

Figura 23.: Diagrama del método de multitud por multinichos.

En resumen, el método de multitud por multinichos difiere de otros métodos de multitud en las siguientes características:

1. Tanto la selección como el reemplazo usan algún método de multitud.

2. Se fomenta el emparejamiento entre miembros del mismo nicho.

3. Se promueve competición entre miembros del mismo nicho.

Estas características parecen ser las responsables de la aparente superioridad de este método sobre otros métodos basados en multitud en tanto como un mantenimiento de soluciones en múltiples cimas.

5.3. Algoritmos evolutivos basados en modelos probabilísticos. PBIL.

En la última década cada vez más investigadores trataron de vencer las desventajas de los operadores de recombinación usuales de los algoritmos de *Computación Evolutiva* que tienen probabilidades de romper los buenos bloques de construcción de soluciones. Así, se desarrollaron una serie de algoritmos, llamados en ocasiones *Algoritmos de Estimación de Distribución* (EDA). Estos algoritmos, que tienen su base teórica en la teoría de la probabilidad, también están basados en poblaciones que evolucionan a medida del progreso de búsqueda. Los *Algoritmos de Estimación de Distribución* usan modelado probabilístico de soluciones prometedoras para estimar una distribución sobre el espacio de búsqueda que es entonces usada para generar la siguiente generación al muestrear el espacio de búsqueda según la distribución estimada. Después de cada iteración es reestimada la distribución. Para obtener una visión general de los *Algoritmos de Estimación de Distribución* ver [PGL99]. A continuación se muestra en pseudocódigo el algoritmo básico asociado a todos los *Algoritmos de Estimación de Distribución*.

Algoritmo básico de Estimación de Distribución (EDA)

```
P ← InicializarPoblacion();
mientras no se alcance la condición de terminación hacer
{
      P_sel←Seleccionar(P);
      ρ(x)=ρ(x|P_sel)← EstimarDistribucionProbabilidad();
      P←MuestrearDistribucionProbabilidad();
}
```

Figura 24.: Estructura del algoritmo básico de Estimación de Distribución (EDA)

Uno de los primeros *Algoritmos de Estimación de Distribución* propuestos para la optimización combinatoria es el llamado *Aprendizaje Incremental basado en Población* (PBIL) [[Bal94],[BC95]].

Tratamos con ***Metaheurísticas*** de búsqueda basadas en la adaptación de probabilidades. Estas ***Metaheurísticas*** constituyen una familia de algoritmos para la resolución de problemas binarios fundamentalmente como aplicación más común. Vemos que, en lugar de adaptar los individuos, como se hace en otros *Algoritmos Evolutivos*, en este caso se adapta el vector de probabilidades.

El algoritmo general que sirve como base para la mayor parte de las ***Metaheurísticas*** que tratamos en esta sección se describe en pseudocódigo a continuación…

Algoritmo genérico básico de Metaheurísticas de búsqueda mediante adaptación de probabilidades

```
Inicializar V=(p_1,...,p_l); (en general (0.5,...,0.5))
Generar P=(s_1,...,s_n) utilizando V;
Evaluar f(s_1),...,f(s_n);
Actualizar V de acuerdo a P y f(s_1),...,f(s_n);
Ir a 2 o Parar.
```

Figura 25.: Estructura genérica más utilizada para los algoritmos asociados a la Metaheurísticas de búsqueda mediante adaptación de probabilidades.

La diferencia fundamental entre los distintos *Algoritmos asociados a las Metaheurísticas de Búsqueda Mediante Adaptación de Probabilidades* suele venir dada por el mecanismo de actualización de la distribución de probabilidad.

A continuación entraremos de forma concreta en la Metaheurística de las tratadas que hemos aplicado para la resolución del problema de la selección de la solución deseada: PBIL.

Evolución de poblaciones mediante aprendizaje incremental: PBIL.

El objetivo de este método es crear un vector de probabilidad real estimado (cada posición corresponde a una variable de decisión) que, cuando es muestreado, genera soluciones de alta calidad con alta probabilidad. Inicialmente, los valores del vector de probabilidad son inicializados a 0.5. El objetivo del desplazamiento de valores de este vector de probabilidad para generar soluciones de alta calidad se consuma como sigue: se genera un número de vectores

solución a partir del vector de probabilidad. Después el vector de probabilidad es dirigido hacia los vectores solución generados con la mayor calidad. La distancia que el vector de probabilidad es desplazado depende del parámetro ratio de aprendizaje. Después de eso, se repite el ciclo. El vector de probabilidad puede ser visto como un vector prototipo para generar vectores solución de alta calidad con respecto al conocimiento disponible sobre el espacio de búsqueda. La desventaja de este método es el hecho de que no proporciona una forma automática de tratar con problemas restringidos. En contraposición a PBIL que estima una distribución de soluciones prometedoras asumiendo que las variables de decisión son independientes, otras aproximaciones tratan de estimar las distribuciones teniendo en cuenta dependencias entre las variables de decisión. Un ejemplo de *Algoritmos de Estimación de Distribución* que tiene en cuenta la dependencia entre variables de decisión es MIMIC [BIV97] y un ejemplo para dependencias multivariadas es el Algoritmo de Optimización Bayesiana (BOA) [PGC99].

Usamos el concepto de población de cromosomas con distribuciones de probabilidad aplicadas a los genes. La distribución de probabilidad se representa como hemos visto como un vector con elementos en [0,1] que especifican la probabilidad de que cada posición contenga un 1 o un 0. El vector de probabilidades se usa para generar la siguiente población y se actualiza usando una regla que considera la mejor solución encontrada hasta ahora

```
V[i]← V[i]·(1.0-LR)+Mejor[i]·LR;                    (15)
```

donde LR es un parámetro que controla la velocidad de convergencia (ratio de aprendizaje).

Para evitar una convergencia prematura se aplica un operador de mutación sobre el vector de probabilidades.

El algoritmo PBIL básico se describe a continuación en pseudocódigo…

Algoritmo básico de Evolución de Poblaciones mediante Aprendizaje Incremental (PBIL)

```
InicializarVectorProbabilidades(P); (cada posición = 0.5)
para cada generación hacer {
      para i←0...numero_muestras hacer
      {
            sols[i]← generarSolucionSegunProbabilidades(P);
            evaluaciones[i]← evaluar(sols[i]);
      }
      mejor← EncontrarMejorSolucion(sols,evaluaciones);
      para i←0...longitud_vector_P hacer
            p[i]←p[i]*(1.0-LR)+mejor*LR;
      para i←0...longitud_vector_P hacer
         if (aleat([0,1])<PROB_MUT) entonces
            p[i]←p[i]*(1.0-DESP_MUT)
                  +aleat(0.0 o 1.0)*DESP_MUT;
}
```

Figura 26.: Estructura genérica del algoritmo PBIL básico.

En este algoritmo podemos comprobar cómo se utilizan una serie de constantes definidas por el usuario: número de generaciones de aprendizaje, tamaño de la población o número de muestras a producir en cada generación, longitud de cada solución, probabilidad de mutación de cada posición, cantidad de desplazamiento por mutación que afecta al vector de probabilidades y ratio de aprendizaje. Las variables utilizadas son las siguientes: vector de probabilidades, vector de soluciones, vector que contiene el fitness de cada solución del vector

de soluciones y vector solución correspondiente a la solución de mayor calidad de cada generación.

Se establecen una serie de extensiones básicas para el algoritmo PBIL. Una de ellas supone el aprender en cada iteración de los *M* mejores elementos en vez de solamente a partir del mejor elemento. Todo ello supone una modificación de la regla de actualización del vector de probabilidades. Además, existen varias formas posibles de aprender de los *M* mejores elementos, algunas de las cuales se indican a continuación: de forma equitativa, considerando sólo las posiciones en las cuales existe un consenso y de forma relativa a la función de evaluación.

Otra extensión básica supone el actualizar el vector de probabilidades teniendo en cuenta los peores elementos. Para ello con movernos hacia el complemento del peor no basta, pues dará problemas durante estados avanzados de la búsqueda, ya que el mejor y el peor se parecen. La idea fundamental la encontramos en utilizar sólo los bits del peor que difieran de los del mejor. En este caso la regla de actualización del vector de probabilidades supondría lo siguiente: actualización del vector de probabilidades cerca de la mejor solución y actualización del vector de probabilidades lejos de la peor solución en base solamente a los bits del peor que difieran de los del mejor. Habrá que utilizar aquí un nuevo parámetro: un ratio de aprendizaje negativo, para la parte de la regla de actualización que aleja de la peor solución.

Existen así mismo extensiones basadas en la comparación: de cromosomas (*Compact-GA*) y de poblaciones (*Compact-GA Extendido*).

A continuación mostramos los algoritmos en pseudocódigo para cada una de estas dos extensiones:

Algoritmo Compact-GA

```
(1){inicializar vector de probabilidades}
inicializarVectorProbabilidades(p);(p[i]←0.5)
(2){generar dos individuos a partir del vector de
probabilidades}
a←generarIndividuoConVectorProbabilidades(p);
b←generarIndividuoConVectorProbabilidades(p);
(3){realizar una competición entre los individuos}
ganador,perdedor←competicionFitness(a,b);
(4){actualizar el vector de probabilidad hacia el mejor
individuo}
para i←1...l hacer
     si ganador[i]≠perdedor[i] entonces
     {
          si ganador[i]=1 entonces
               p[i]←p[i]+1/n;
          sino
               p[i]←p[i]-1/n;  }
(5){comprobar si el vector de probabilidad ha convergido}
para i←1...l hacer
     si p[i]>0 Y p[i]<1 entonces
          volver a paso 2;
(6)p representa la solución final;
```

Figura 27.: Estructura genérica del algoritmo Compact-GA.

En el algoritmo anterior los parámetros utilizados han sido los siguientes: *p* (vector de probabilidades), *l* (longitud de cromosoma), *a* y *b* (cromosomas que intervienen), *ganador* y

perdedor (cromosomas ganador y perdedor de la competición entre *a* y *b*) y *n* (tamaño de población).

Veamos a continuación el pseudocódigo del algoritmo Compact-GA extendido (basado en la comparación de poblaciones)…

Algoritmo Compact-GA Extendido

```
(1){Inicializar el vector de probabilidades}
inicializarVectorProbabilidades(p);(p[i]←0.5)
(2){Generar s individuos a partir del vector de
probabilidades y guardarlos en S}
para i←1...s hacer
     S[i]←generarIndividuoConVectorProbabilidades(p);
(3){Ordena S para que S[1] sea el individuo con mayor
fitness}
ordenarPorFitness(S);
(4){Hacer competir a S[1] con el resto de individuos}
para j←2...s hacer
{
     ganador,perdedor←competicionFitness(S[1],S[j]);
ActualizarVectorProbabilidades(p,ganador,perdedor,l,n);
          (paso 4 del algoritmo Compact-GA)
}
(5)p representa la solución final;
```

Figura 28.: Estructura genérica del algoritmo Compact-GA Extendido.

En el algoritmo anterior los parámetros utilizados han sido los siguientes: *p* (vector de probabilidades), *l* (longitud de cromosoma), *S* (conjunto de cromosomas generados en cada iteración), *s* (número de cromosomas que alberga el conjunto *S*), *ganador* y *perdedor* (cromosomas ganador y perdedor de la competición entre cada par de cromosomas) y *n* (tamaño de población).

Por último, comentaremos que existe un algoritmo que integra características de PBIL y de los algoritmos de Optimización a partir de Colonias de Hormigas: el Sistema de la Mejor y la Peor Hormiga. Entraremos en los detalles de este algoritmo ya dentro de los algoritmos de Optimización a partir de Colonias de Hormigas.

5.4. Optimización a partir de Colonias de Hormigas (ACO)

Fundamentos de Optimización a partir de Colonias de Hormigas (ACO)

La *Optimización a partir de Colonias de Hormigas* (ACO) es un ***método metaheurístico*** reciente que está inspirado por el comportamiento de colonias de hormigas reales. A continuación, revisaremos las ideas subyacentes de esta aproximación que conducen desde la inspiración biológica a la ***Metaheurística*** de *Optimización a partir de Colonias de Hormigas*, la cual proporciona un conjunto de reglas de cómo aplicar los *Algoritmos de Optimización a partir de Colonias de Hormigas* para abordar los problemas combinatorios. Presentamos algunos de los algoritmos que han sido desarrollados bajo este marco, dando una visión global de las actuales aplicaciones, y analizamos la relación entre *Optimización a partir de Colonias de Hormigas* y algunas de las ***Metaheurísticas*** más conocidas.

La *Optimización a partir de Colonias de Hormigas* se inspira concretamente en el comportamiento asociado a la búsqueda de los caminos más cortos por parte de varias especies

de hormigas. Sin embargo, desde el trabajo inicial de Dorigo, Maniezzo y Colorni en el *Sistema de Hormigas* [DMC96], la *Optimización a partir de Colonias de Hormigas* se está convirtiendo ahora rápidamente en un campo de investigación maduro: un gran número de autores han desarrollado modelos más sofisticados que han sido usados para resolver satisfactoriamente un gran número de problemas complejos de optimización combinatoria y se están haciendo disponibles ahora perspicacias teóricas sobre los algoritmos.

Colonias de Hormigas naturales

Las hormigas son insectos sociales que viven en colonias y, a causa de su interacción colaborativa, son capaces de mostrar comportamientos complejos y ejecutar tareas difíciles desde una perspectiva local de hormiga. Un aspecto muy interesante del comportamiento de varias especies de hormigas es su habilidad para encontrar los caminos más cortos entre el hormiguero y la comida. Este hecho es especialmente notable teniendo en mente que en muchas especies de hormigas, las hormigas son casi ciegas, lo que evita la explotación de los indicios visuales.

Mientras andan entre su hormiguero y la comida, algunas especies de hormigas depositan una sustancia química denominada feromona (una sustancia olorosa). Si no hay rastros de feromona disponibles, las hormigas se mueven esencialmente de manera aleatoria, pero en la presencia de feromona tienen una tendencia a seguir el rastro. De hecho, los experimentos de los biólogos han mostrado [[PDG87], [GADP89]] que las hormigas prefieren probabilísticamente caminos que están marcados por una alta concentración de feromona. En la práctica, escogen entre diferentes caminos que ocurren cuando varios caminos se intersectan. Entonces, las hormigas eligen el camino a seguir a partir de una decisión probabilística sesgada por la cantidad de feromona: cuanto más fuerte es el rastro de feromona, más alta es la deseabilidad. Debido a que las hormigas depositan en turno feromona en el camino que siguen, este comportamiento resulta en un proceso sobrerreforzado que conduce a la formación de caminos marcados por altas concentraciones de feromona. Este comportamiento permite identificar a las hormigas los caminos más cortos entre su hormiguero y un depósito de comida [GADP89]. Hay que notar que las hormigas sólo se comunican indirectamente, a través de las modificaciones del entorno físico que perciben. Esta forma de comunicación es llamada estimergia artificial en [DC99].

A continuación comentamos más explícitamente cómo este mecanismo permite a las hormigas alcanzar los caminos más cortos: inicialmente, no hay rastro de feromona en el entorno y, cuando las hormigas llegan a una intersección, eligen aleatoriamente una de las ramas. Sin embargo, a medida que las hormigas viajan, los caminos más prometedores reciben una cantidad más grande de feromona después de algún tiempo. Esto es debido al hecho de que, debido a que estos caminos son más cortos, las hormigas que los siguen son capaces de alcanzar la meta (la comida) más rápido y comenzar más pronto su camino de vuelta. Puesto que, en la rama más corta ya existe un rastro de feromona ligeramente más fuerte, la decisión de las hormigas es sesgada hacia la rama más corta, la cual, además, recibe una proporción más grande de la feromona de las hormigas que regresan que la que reciben las ramas más largas. Este proceso resulta finalmente en un sesgo incrementalmente más fuerte hacia la rama más corta y, al final, para converger a la más corta de las existentes.

Todo se complementa con el entorno natural por el hecho de que la feromona se evapora después de algún tiempo. De esta forma, los cominos menos prometedores pierden progresivamente feromona a causa de ser visitados por cada vez menos hormigas. Sin embargo, varios estudios biológicos muestran que los rastros de feromona son muy persistentes (la feromona puede permanecer de varias horas a varios meses dependiendo de aspectos tales como la especie de hormigas, el tipo de suelo, … [BDT99]), haciendo así menos significativa la influencia de la evaporación en el comportamiento de la búsqueda del camino más corto.

En [BDT99], varios experimentos han sido escritos mostrando que el reclutamiento de masas en la naturaleza es restrictivo ya que, como resultado de la larga persistencia de la feromona, es difícil que las hormigas olviden un camino con un alto nivel de feromona, aunque hayan encontrado uno más corto. Hay que darse cuenta de que, si este comportamiento es directamente trasladado a la computadora para diseñar un algoritmo de búsqueda, podemos conseguir un algoritmo que rápidamente se quede estancado en óptimos locales.

De las hormigas naturales a la Metaheurística de Optimización a partir de Colonias de Hormigas

Los *Algoritmos de Optimización a partir de Colonias de Hormigas* toman su inspiración del comportamiento de las colonias de hormigas reales para resolver problemas de optimización combinatoria. Están basados en una colonia de hormigas artificiales, esto es, agentes computacionales simples que trabajan cooperativamente y se comunican a través de rastros de feromona artificiales.

Los *Algoritmos de Optimización a partir de Colonias de Hormigas* son esencialmente algoritmos de construcción: en cada iteración del algoritmo, cada hormiga construye una solución al problema al viajar en un grafo de construcción. Cada arista del grafo, representando los posibles pasos que la hormiga puede hacer, tiene asociados dos tipos de información que guían el movimiento de la hormiga:

- *Información heurística:* que mide la preferencia heurística de moverse del nodo *r* al nodo *s*, o sea, de viajar por ejemplo por la arista a_{rs}. Es denotada por η_{rs}. Esta información no es modificada por las hormigas durante la ejecución de la hormiga.

- *Información memorística:* que mide la deseabilidad del movimiento e imita la feromona real que las hormigas naturales depositan. Esta información es modificada durante la ejecución del algoritmo dependiendo de las soluciones encontradas por las hormigas. Se denota por τ_{rs}.

Estamos introduciendo los pasos que conducen de las hormigas reales a la *Optimización a partir de Colonias de Hormigas.* Debería tenerse en cuenta para lo siguiente que los *Algoritmos de Optimización a partir de Colonias de Hormigas* presentan una doble perspectiva…

- Por una parte, son una abstracción de algunos patrones de comportamiento de las hormigas naturales relativos al comportamiento de búsqueda del camino más corto.

- Por otra parte, incluyen varios rasgos que no tienen un contrapunto natural, pero que permiten desarrollar algoritmos para obtener buenas soluciones al problema tratado (por ejemplo, el uso de información heurística para guiar el movimiento de la hormiga).

Tipos de problemas resueltos por *Optimización a partir de Colonias de Hormigas* (ACO)

El tipo de problemas que resuelven las hormigas artificiales pertenecen al grupo de problemas de caminos más cortos (restringidos).

Estos problemas pueden ser caracterizados por los siguientes aspectos (siguiendo principalmente la presentación de [DC99] y [DS02]):

- Hay un conjunto de restricciones Ω definido para el problema a resolver.
- Hay un conjunto finito de componentes $N=\{n_1, n_2,\ldots, n_l\}$.
- El problema presenta varios estados definidos sobre secuencias de componentes ordenados $\delta = < n_r, n_s,\ldots, n_u,\ldots>$ ($<r,s,\ldots,u\ldots>$ para simplificar) sobre los elementos de N. Si Δ es el conjunto de todas las posibles secuencias, denotamos por $\tilde{\Delta}$ el conjunto de las (sub)secuencias factibles con respecto a las restricciones Ω. Los elementos en $\tilde{\Delta}$ definen los estados factibles. $|\delta|$ es la longitud de una secuencia δ, o sea, el número de componentes en la secuencia.
- Hay una estructura de vecindario definida como sigue: δ_2 es un vecino de δ_1 si (i) tanto δ_1 como δ_2 pertenecen a Δ, (ii) el estado δ_2 puede ser alcanzado desde δ_1 en un movimiento lógico, o sea, si r es la última componente de la secuencia δ_1, debe existir una componente $s \in N$ tal que $\delta_2 = <\delta_1,s>$, es decir, existe una transición válida entre r y s. El vecindario factible de δ_1 es el conjunto que contiene todas las secuencias $\delta_2 \in \tilde{\Delta}$; si $\delta_2 \notin \tilde{\Delta}$, decimos que δ_2 está en el vecindario no factible de δ_1.
- Una solución S es un elemento de $\tilde{\Delta}$ verificando todos los requerimientos del problema.
- Hay un coste $C(S)$ asociado a cada solución S.
- En algunos casos, pueden ser asociados a los estados un coste o una estimación del coste.

Como se dijo, todas las características previas se encuadran en problemas de optimización combinatoria que pueden ser representados en la forma de un grafo con pesos $G=(N,A)$, donde A es el conjunto de las aristas que conectan el conjunto de componentes N. El grafo G es también llamado grafo de construcción G. Como se dice en [DS02], el conjunto de aristas puede conectar completamente los componentes. En este caso, la implementación de las restricciones está completamente integrada en la política de construcción de las hormigas. Por tanto, tenemos que…

- Los componentes n_r son los nodos del grafo.
- Los estados δ (y por tanto las soluciones S) corresponden a caminos en el grafo, es decir, secuencias de nodos o aristas.
- Las aristas del grafo, a_{rs}, son conexiones/transiciones que definen la estructura de vecindario. $\delta_2 = < \delta_1 , s >$ es un vecino de δ_1 si el nodo r es la última componente de δ_1 y la arista a_{rs} existe en el grafo.
- Puede haber costes de transición explícitos c_{rs} asociados a cada arista.
- Los componentes y conexiones pueden tener asociados rastros de feromona τ, que representan alguna forma de memoria de largo plazo indirecta del proceso de búsqueda, y valores heurísticos η, que representan alguna información heurística disponible en el problema a resolver.

La hormiga artificial

La hormiga artificial es un agente computacional simple que intenta construir soluciones factibles al problema tratado explotando los rastros de feromona disponibles y la información heurística. Sin embargo, si es necesario, puede construir también soluciones no

factibles que pueden ser penalizadas dependiendo de la cantidad de infactibilidad. Tiene las siguientes propiedades [[DC99], [DS02]]:

- Busca soluciones factibles de mínimo coste para el problema a resolver.
- Tiene una memoria L que guarda información acerca del camino seguido hasta el momento, o sea, L guarda la secuencia generada. Esta memoria puede ser usada para: (i) construir soluciones factibles, (ii) evaluar la solución generada, y (iii) volver a seguir la traza del camino seguido por la hormiga.
- Tiene un estado inicial $\delta_{inicial}$, que corresponde usualmente a una secuencia unitaria, y una o más condiciones de terminación t asociadas.
- Comienza en el estado inicial y se mueve hacia estados factibles, construyendo incrementalmente su solución asociada.
- Cuando está en un estado $\delta_r = < \delta_{r-1} , r >$ (es decir, está localizada en el nodo r y ha seguido previamente la secuencia δ_{r-1}), puede moverse a cualquier nodo s de su vecindario factible $N(r)$, definido como $N(r) = \{s|(a_{rs} \in A)$ y $(< \delta_r , s > \in \tilde{\Delta})\}$.
- El movimiento es hecho al aplicar una regla de transición, que está en función de los rastros de feromona y los valores heurísticos disponibles localmente, la memoria privada de las hormigas y las restricciones del problema.
- Cuando, durante el proceso de construcción, una hormiga se mueve del nodo r al s, puede actualizar el rastro de feromona τ_{rs} asociado a la arista a_{rs}. Este proceso es denominado *actualización de feromona en línea paso a paso*.
- El proceso de construcción acaba cuando se satisface una condición de terminación, usualmente cuando se alcanza un estado objetivo.
- Una vez que la solución ha sido construida, la hormiga puede volver a trazar el camino recorrido y actualizar los rastros de feromona de las aristas/componentes visitadas por medio de un proceso llamado *actualización de feromona en línea a posteriori*.

De esta forma, el único mecanismo de comunicación entre las hormigas es la estructura de datos que almacena los niveles de feromona de cada arista/componente (memoria compartida).

Semejanzas y diferencias entre las hormigas naturales y artificiales

Las colonias de hormigas reales y artificiales comparten una serie de características. Las más importantes se pueden resumir como sigue (ver [DCG99] para una descripción más detallada):

- Uso de una colonia de indivudos que interactúan y colaboran para resolver una tarea dada.
- Tanto hormigas naturales como artificiales modifican su "entorno" a través de la comunicación estimérgica basada en feromonas. En el caso de las hormigas artificiales, los rastros de feromona artificial son una información numérica sólo disponible localmente.
- Tanto hormigas naturales como artificiales comparten una tarea común: la búsqueda del camino más corto (construcción iterativa de una solución de mínimo coste, que puede ser trasladada a solución de máximo coste) desde un

origen, el hormiguero (decisión inicial), a algún estado objetivo, la comida (última decisión).

- Las hormigas artificiales construyen las soluciones iterativamente al aplicar una política de transición estocástica local para moverse entre estados adyacentes, como lo hacen las hormigas reales.

Sin embargo, sólo estas características no permiten desarrollar algoritmos eficientes para problemas combinatorios duros. Por lo tanto, las hormigas artificiales viven en un mundo discreto y tienen capacidades adicionales:

- Las hormigas artificiales pueden hacer uso de información heurística (y no solo información de rastros de feromona) en la política de transición estocástica que aplican.

- Tienen una memoria que guarda el camino seguido por la hormiga.

- La cantidad de feromona depositada por las hormigas artificiales está en función de la calidad de la solución encontrada por cada una de ellas. Sin embargo, esta diferencia es relativa ya que algunas especies naturales de hormigas depositan una cantidad más alta de feromona cuando encuentran una fuente de comida más rica [BDT99]. Una diferencia mayor también concierne al tiempo del depósito de feromona. Las hormigas artificiales usualmente solo depositan feromona después de generar una solución completa. Pero sin embargo, como veremos posteriormente, algunos *Algoritmos de Optimización a partir de Colonias de Hormigas* también modifican los rastros de feromona mientras se construye una solución.

- Como se dijo anteriormente, la evaporación de feromona en los *Algoritmos de Optimización a partir de Colonias de Hormigas* es diferente de la de la naturaleza, ya que la inclusión de un mecanismo de evaporación es una cuestión clave para evitar que el algoritmo se estanque en óptimos locales. La evaporación de feromona permite a la colonia de hormigas artificiales olvidar suavemente su historia pasada y dirigir su búsqueda hacia nuevas regiones del espacio. Esto evita una convergencia prematura del algoritmo hacia óptimos locales.

- Para mejorar la eficiencia y eficacia del sistema, los *Algoritmos de Optimización a partir de Colonias de Hormigas* pueden ser enriquecidos con capacidades adicionales. Ejemplos los tenemos en la capacidad para mirar más lejos de la siguiente transición [MM98], optimización local [[DG97], [SH97]] y backtracking (cuyo uso no está muy extendido), o la llamada lista de candidatos que contiene un conjunto de los estados vecinos más prometedores [[DG97], [DC99]] para mejorar la eficiencia del algoritmo.

Modo de operación y estructura genérica de un algoritmo de Optimización a partir de Colonias de Hormigas (ACO)

Como se ha visto anteriormente, el modo de operación básico de un *Algoritmo de Optimización a partir de Colonias de Hormigas* es como sigue: las *m* hormigas (artificiales) de la colonia se mueven, concurrentemente y asíncronamente, a través de estados adyacentes de un problema (que pueden ser representados en la forma de un grafo con pesos). Este movimiento es hecho según una regla de transición que está basada en información local disponible en los componentes (nodos). Esta información local comprende información heurística y memorística (rastros de feromona) para guiar la búsqueda. Al moverse en el grafo de construcción, las

hormigas construyen incrementalmente las soluciones. Opcionalmente, las hormigas pueden lanzar feromona cada vez que cruzan una arista (conexión) mientras construyen soluciones (*actualización de feromona en línea paso a paso*). Una vez que cada hormiga ha generado una solución, ésta es evaluada y se puede depositar una cantidad de feromona que es función de la calidad de la solución de la hormiga (*actualización de feromona en línea a posteriori*). Esta información guiará la búsqueda de las otras hormigas de la colonia en el futuro.

Además, el modo de operación genérico del *Algoritmo de Optimización a partir de Colonias de Hormigas* también incluye dos procedimientos adicionales, *la evaporación de rastros de feromona* y *las acciones del demonio*. La evaporación de feromona es desencadenada por el entorno y es usada como un mecanismo para evitar el estancamiento en la búsqueda y para permitir a las hormigas explorar nuevas regiones del espacio. Las acciones del demonio son acciones opcionales, sin contrapunto natural, para implementar tareas desde una perspectiva global que le falta a la perspectiva local de las hormigas. Las capacidades adicionales mencionadas en el apartado anterior están incluidas en estas acciones. Ejemplos son observar la calidad de todas las soluciones generadas y depositar una cantidad de feromona adicional en los componentes/transiciones asociados a algunas de las soluciones, o aplicar un procedimiento de búsqueda local a las soluciones generadas por las hormigas antes de actualizar los rastros de feromona. En ambos casos, el demonio reemplaza la *actualización de feromona en línea a posteriori* y el proceso es llamado *actualización de feromona fuera de línea*.

La estructura genérica de un *Algoritmo de Optimización a partir de Colonias de Hormigas* se establece en la página siguiente [[DC99], [DCG99]]…

Algoritmo de Optimización a partir de Colonias de Hormigas

```
Procedimiento Metaheuristica_ACO{
      inicializacion_parametros();
      mientras (no se satisfaga el criterio de terminacion)
hacer
      {
            {tareas_planificacion}
                  generacion_y_actividad_de_hormigas();
                  evaporacion_feromona();
                  acciones_demonio();{opcional}
      }
}
Procedimiento generacion_y_actividad_de_hormigas()
{
      para k←1...m hacer {m es el número de hormigas}
            nueva_hormiga(k);
}
Procedimiento nueva_hormiga(id_hormiga)
{
      inicializar_hormiga(id_hormiga);
      L←actualizar_memoria_hormiga();
      mientras(estado_actual≠estado_objetivo) hacer
      {
       P←calcular_probabilidades_transicion(A,L,Ω);
       sig_estado←aplicar_politica_decision_hormiga(P,Ω);
       mover_al_siguiente_estado(sig_estado);
       si (actualiz_feromona_en_linea_paso_a_paso) entonces
                  depositar_feromona_en_arista_visitada();
                  L←actualizar_estado_interno();
      }
```

```
        si(actualizacion_feromona_linea_posteriori)entonces
              para cada lado visitado hacer
                    depositar_feromona_en_lado_visitado();
        liberar_recursos_hormiga(id_hormiga);
}
```

Figura 29.: Estructura genérica de Algoritmo de Optimización a partir de Colonias de Hormigas (ACO).

El primer paso supone la inicialización de los valores de los parámetros considerados por el algoritmo. Entre otros son: el valor de feromona inicial asociado a cada transición, τ_0, que es un valor positivo pequeño que es típicamente el mismo para todas las conexiones/componentes, el número de hormigas en la colonia, *m*, y los pesos que definen el equilibrio entre la información heurística y la memorística en la regla probabilística de transición.

El procedimiento principal de la `Metaheuristica_ACO` gestiona, por medio de la construcción `tareas_planificacion`, la planificación de los tres componentes mencionados en esta sección: (i) la generación y operación de las hormigas artificiales, (ii) la evaporación de feromona, y (iii) las acciones del demonio. La implementación de esta construcción definirá el sincronismo existente entre los tres componentes. Mientras la aplicación a los problemas clásicos NP-duros (no distribuidos) usa típicamente una planificación secuencial, en los problemas distribuidos como paralelismo de enrutamiento en red puede ser fácil y eficientemente explotada.

Como se dijo, varios componentes son o bien opcionales, tales como las acciones del demonio, o bien estrictamente dependientes del *Algoritmo de Optimización a partir de Colonias de Hormigas* específico, como por ejemplo, cuándo y dónde la feromona es depositada. Generalmente, la *actualización de feromona en línea paso a paso* y la *actualización de feromona en línea a posteriori* son mutuamente exclusivas y no se presentan o faltan al mismo tiempo (si ambas faltan, típicamente el demonio actualiza los rastros de feromona).

Por otra parte, hay que darse cuenta de que el procedimiento `actualizar_memoria_hormiga` involucra especificar el estado inicial desde el que la hormiga comienza su camino y guardar el correspondiente componente en la memoria de la hormiga *L*. La decisión sobre cuál será ese nodo (puede ser una elección aleatoria o una fija para la colonia entera, una elección aleatoria o una fija para cada hormiga, etc.,) depende de la aplicación específica.

Finalmente, hay que notar que los procedimientos `calcular_probabilidades_transicion` y `aplicar_politica_decision_hormigas` consideran el estado actual de la hormiga, los valores actuales de los rastros de feromona visibles en ese nodo y las restricciones del problema Ω para establecer el proceso de transición probabilística a otros estados factibles.

Pasos para resolver un problema a partir de ACO

Desde las actualmente conocidas aplicaciones *ACO*, podemos identificar algunas líneas de guía de cómo abordar problemas a partir de *ACO*. Estas líneas de guía pueden ser resumidas a partir de las siguientes tareas de diseño:

1. Representar el problema en forma de conjuntos de componentes y transiciones o por medio de un grafo de pesos, que será recorrido por las hormigas para construir soluciones.

2. Definir apropiadamente el significado de los rastros de feromona τ_{rs}, es decir, el tipo de decisión que sesgan. Este es un paso crucial en la implementación de un algoritmo *ACO* y a menudo, una buena definición de los rastros de feromona no es una tarea trivial y típicamente requiere penetración dentro del problema a resolver.

3. Definir apropiadamente la preferencia heurística para cada decisión que tiene que tomar una hormiga mientras construye una solución, o sea, definir al información heurística η_{rs} asociada a cada componente o transición. Hay que darse cuenta de que la información heurística es crucial para un buen rendimiento si los algoritmos de búsqueda local no están disponibles o no pueden ser aplicados.

4. Si es posible, implementar un algoritmo de búsqueda local eficiente para el problema a resolver, debido a que los resultados de muchas aplicaciones de *ACO* a problemas de optimización combinatoria NP-duros muestran que el mejor rendimiento es logrado cuando se empareja *ACO* con optimizadores locales [[DC99], [DS02]].

5. Elegir un algoritmo *ACO* específico (a continuación describiremos algunos de los disponibles) y aplicarlo al problema a resolver, teniendo en cuenta los aspectos previos.

6. Afinar los parámetros del algoritmo *ACO*. Un buen punto de inicio para el ajuste de parámetros es usar ajustes de parámetros que fueron encontrados buenos al aplicar el algoritmo *ACO* a problemas similares o a una variedad de otros problemas. Otra posibilidad es la de usar procedimientos automáticos de ajuste de parámetros [BSPV02].

Hay que tener claro que los pasos dados arriba solo pueden ser una guía para la implementación de algoritmos *ACO*. Además, la implementación es a menudo un proceso iterativo. Queremos insistir finalmente en que, probablemente los pasos más importantes son los primeros cuatro, donde las elecciones son cruciales.

Modelos de Optimización de Colonias de Hormigas

Varios algoritmos han sido propuestos en la literatura siguiendo la ***Metaheurística*** *ACO*. Entre los algoritmos *ACO* disponibles para problemas de optimización combinatoria NP-duros se encuentran los siguientes: *Sistema de Hormigas* [DMC96], *Sistema de Colonia de Hormigas* [DG97], *Sistema de Hormigas MAX-MIN* [SH00], *Sistema de Hormigas Basado en Rangos* [BHS99], y *Sistema de Hormigas de la Mejor y la Peor Hormiga* [CFHM00]. A continuación, daremos una breve descripción de estos algoritmos. Para una descripción más detallada de los mismos, incluyendo algunas comparaciones de su rendimiento podremos consultar [[DC99], [DCG99], [CFH02], [DS03], [SD99]]. Mientras que el *Sistema de Hormigas* es principalmente de interés histórico porque fue el primer algoritmo *ACO*, los otros cuatro logran típicamente muchos mejores resultados computacionalmente hablando.

Notemos que en lo siguiente consideraremos el caso, donde la información heurística y los rastros de feromona corresponden únicamente a las conexiones, que es el caso para muchas aplicaciones de *ACO* a problemas de secuenciación o asignación. Es directa la extensión de la descripción al caso en el que los rastros de feromona están asociados a componentes.

Sistema de Hormigas

El *Sistema de Hormigas* (AS) [DMC96], desarrollado por Dorigo, Maniezzo y Colorni en 1991, fue el primer algoritmo *ACO*. Inicialmente, fueron propuestas tres variantes diferentes, *Sistema de Hormigas* de densidad, *Sistema de Hormigas* de cantidad y *Sistema de Hormigas* de ciclo, diferenciándose en la forma en la que los rastros de feromona eran actualizados. En los primeros dos, las hormigas depositan la feromona mientras construyen sus soluciones (es decir, aplican una actualización de feromona en línea paso a paso), con la diferencia de que la cantidad depositada en el *Sistema de Hormigas* de densidad es constante mientras que la depositada en *Sistema de Hormigas* de cantidad depende directamente de la deseabilidad heurística de la transición η_{ij}. Finalmente, en *Sistema de Hormigas* de ciclo, el depósito de feromona es hecho una vez que la solución se completa (actualización de feromona en línea a posteriori). Esta última variante fue la que dio mayor rendimiento y actualmente es la que es denotada como *Sistema de Hormigas* en la literatura (en lo sucesivo hablaremos de la misma con este nombre).

El *Sistema de Hormigas* está caracterizado por el hecho de que la actualización de feromona se dispara una vez que todas las hormigas han completado sus soluciones y ello es hecho como sigue. Primero, todos los rastros de feromona son reducidos por un factor constante, implementándose de esta forma la evaporación de feromona. Segundo, cada hormiga de la colonia deposita una cantidad de feromona que está en función de la calidad de su solución. Inicialmente, el *Sistema de Hormigas* no usó ninguna de las acciones del demonio, pero es muy directo hacerlo, por ejemplo, añadiendo un procedimiento de búsqueda local para refinar las soluciones generadas por las hormigas.

Las soluciones en el *Sistema de Hormigas* son construidas como sigue. En cada paso de construcción, una hormiga *k* del *Sistema de Hormigas* elige ir a un nodo siguiente con una probabilidad que es calculada como…

$$p_{rs}^{k} = \begin{cases} \dfrac{[\tau_{rs}]^{\alpha} \cdot [\eta_{rs}]^{\beta}}{\sum_{u \in N_r^k} [\tau_{ru}]^{\alpha} \cdot [\eta_{ru}]^{\beta}} & \text{si } s \in N_k(r) \\ 0 & \text{en otro caso} \end{cases} \qquad (16)$$

donde $N_k(r)$ es el vecindario factible de la hormiga *k* cuando se encuentra en el nodo *r*, y $\alpha,\beta \in \Re$ son dos parámetros que pesan la importancia relativa del rastro de feromona y de la información heurística. Cada hormiga *k* guarda la secuencia que ha seguido y esta memoria L_k es utilizada, como se explico antes, para determinar $N_k(r)$ en cada paso de construcción.

En cuanto a los parámetros α y β, su papel es como sigue: si α=0, aquellos nodos con mejor preferencia heurística tienen una probabilidad más alta de ser seleccionados, acercando así al algoritmo a un algoritmo greedy probabilístico clásico (con múltiples puntos de inicio en caso de que las hormigas estén localizadas en nodos diferentes al principio de cada iteración). Sin embargo, si β=0, solo los rastros de feromona son considerados para guiar el proceso constructivo, lo que puede causar un estancamiento rápido, por ej., una situación donde los rastros de feromona asociados a algunas transiciones son significativamente más altos que las restantes, haciendo así a las hormigas construir siempre las mismas soluciones, normalmente óptimos locales. Por tanto, hay una necesidad de establecer un equilibrio adecuado entre la importancia de las informaciones heurística y memorística.

Como ya se indicó, el depósito de feromona es hecho una vez que todas las hormigas han acabado de construir sus soluciones. Primero, el rastro de feromona asociado a cada arco es evaporado al reducir todos los valores de feromona por un factor constante…

$$\tau_{rs} \leftarrow (1-\rho)\cdot\tau_{rs} \qquad (17)$$

donde $\rho \in (0,1]$ es el ratio de evaporación. Posteriormente, cada hormiga vuelve a trazar el camino que ha seguido (este camino está guardado en su memoria local L_k) y deposita una cantidad de feromona $\Delta\tau_{rs}^{k}$ en cada conexión atravesada…

$$\tau_{rs} \leftarrow \tau_{rs} + \Delta\tau_{rs}^{k}, \forall a_{rs} \in S_k \qquad (18)$$

donde $\Delta\tau_{rs}^{k} = f(C(S_k))$, o sea, la cantidad de feromona depositada depende de la calidad $C(S_k)$ de la solución S_k de la hormiga k.

Para resumir la descripción del *Sistema de Hormigas*, mostraremos la composición del procedimiento `nueva_hormiga` para este algoritmo *ACO* particular en la siguiente página…

Procedimiento nueva_hormiga asociado al Sistema de Hormigas

```
Procedimiento nueva_hormiga(id_hormiga)
{
      k←id_hormiga;r←genera_estado_inicial();Sk←r;
      Lk←r;
      mientras (estado_actual≠estado_objetivo) hacer
      {
         para cada s∈Nk(r)hacer
                        p_rs^k = [τ_rs]^α·[η_rs]^β / Σ_{u∈N_r^k} [τ_ru]^α·[η_ru]^β
         sig_estado←aplicar_politica_decision_hormiga
                        (P,Nk(r));
         r←sig_estado; {Sk=<Sk,r>}
         ----
         Lk←Lk ∪ r;
      }
      para cada arista ars hacer
         τrs←(1-ρ)*τrs;
      para cada arista ars ∈ Sk hacer
         τrs←τrs+f(C(Sk));
      liberar_recursos_hormiga(id_hormiga);          }
```

Figura 30.: Estructura del núcleo del algoritmo del Sistema de Hormigas

Notemos que las indicaciones (---) en la figura anterior tienen como objetivo el remarcar que no se hace actualización de feromona en línea paso a paso. Así mismo, hemos de darnos cuenta que la evaporación se hace antes de realizar la deposición de feromona en línea a posteriori.

Antes de concluir esta sección, es importante hacer constar que los creadores del *Sistema de Hormigas* también propusieron una versión extendida de este algoritmo llamada *Sistema de Hormigas* elitista, de mejor rendimiento [DMC96]. En el *Sistema de Hormigas* elitista, una vez que las hormigas han depositado feromona en las conexiones asociadas a sus soluciones generadas, el demonio ejecuta un depósito de feromona adicional en las aristas que pertenecen a la mejor solución encontrada hasta el momento en el proceso de búsqueda (esta solución es también llamada mejor solución global de la búsqueda). La cantidad de feromona

depositada, que depende de la calidad de la mejor solución global, es pesada por el número de hormigas elitistas consideradas, *e*, como sigue…

$$\tau_{rs} \leftarrow \tau_{rs} + e \cdot f(C(S_{mejor-global})), \forall a_{rs} \in S_{mejor-global} \qquad (19)$$

Sistema de Colonia de Hormigas

El *Sistema de Colonia de Hormigas* (ACS) [DC99] es uno de los primeros sucesores del Sistema de Hormigas. Introduce tres modificaciones principales sobre el Sistema de Hormigas:

1. El *Sistema de Colonia de Hormigas* usa una regla de transición diferente, que es llamada una regla proporcional pseudoaleatoria: Sea *k* una hormiga localizada en un nodo *r*, sea $q_0 \in [0,1]$ un parámetro, y *q* un valor aleatorio del intervalo [0,1]. El siguiente nodo *s* es aleatoriamente elegido según la siguiente distribución de probabilidad…

   Si $q \leq q_0$:

   $$p_{rs}^k = \begin{cases} 1 & \text{si } s = \arg\max_{u \in N_k(r)} \{\tau_{ru}^{\alpha} \cdot \eta_{ru}^{\beta}\} \\ 0 & \text{en otro caso} \end{cases}$$

   sino ($q > q_0$)

   $$p_{rs}^k = \begin{cases} \dfrac{[\tau_{rs}]^{\alpha} \cdot [\eta_{rs}]^{\beta}}{\sum_{u \in N_r^k} [\tau_{ru}]^{\alpha} \cdot [\eta_{ru}]^{\beta}} & \text{si } s \in N_k(r) \\ 0 & \text{en otro caso} \end{cases} \qquad (20)$$

   Como puede verse, la regla tiene una doble vertiente: cuando $q \leq q_0$, utiliza el conocimiento disponible, escogiendo la mejor opción con respecto a la información heurística y memorística. Sin embargo si $q > q_0$, aplica una exploración controlada, como ocurre en el Sistema de Hormigas. En resumen, la regla establece un equilibrio dinámico entre la exploración de nuevas conexiones y la explotación de la información disponible en el momento.

2. Solo el demonio (y no las hormigas individuales) dispara la actualización de feromona, es decir, se realiza una actualización de feromona fuera de línea. Para hacer esto, el *Sistema de Colonia de Hormigas* sólo considera una sola hormiga, aquella que genera la mejor solución global, $S_{\text{mejor-global}}$ (aunque inicialmente fue considerada también una actualización basada en la mejor hormiga de la iteración [DG97], el *Sistema de Colonia de Hormigas* casi siempre aplica una actualización basada en la mejor hormiga global).

   La actualización de feromona es hecha primero evaporando los rastros de feromona en todas las conexiones usadas por la mejor hormiga global (es importante notar que en el *Sistema de Colonia de Hormigas*, la evaporación de feromona es sólo aplicada a las conexiones de la solución que es también usada para depositar feromona) como sigue…

   $$\tau_{rs} \leftarrow (1-\rho) \cdot \tau_{rs}, \forall a_{rs} \in S_{mejor-global} \qquad (21)$$

   A continuación, el demonio deposita feromona a través de la regla…

$$\tau_{rs} \leftarrow \tau_{rs} + \rho \cdot f(C(S_{mejor-global})), \forall a_{rs} \in S_{mejor-global} \qquad (22)$$

Adicionalmente, el demonio puede aplicar un algoritmo de búsqueda local para mejorar las soluciones de las hormigas antes de actualizar los rastros de feromona.

3. Las hormigas aplican una actualización de feromona en línea paso a paso que estimula la generación de diferentes soluciones a aquellas ya encontradas.

   Cada vez que una hormiga pasa por una arista a_{rs}, aplica la regla…

$$\tau_{rs} \leftarrow (1-\varphi) \cdot \tau_{rs} + \varphi \cdot \tau_0 \qquad (23)$$

   donde $\varphi \in (0,1]$ es un segundo parámetro de descenso de feromona. Como puede verse, la regla de actualización de feromona paso a paso incluye tanto la evaporación de feromona como el depósito. Debido a que la cantidad de feromona depositada es muy pequeña (de hecho, τ_0 es el valor del rastro de feromona inicial que es escogido de tal forma que, en la práctica, corresponda a un límite inferior de feromona, o sea, para la elección de las reglas de actualización de feromona del *Sistema de Colonia de Hormigas*, ningún valor de feromona puede caer por debajo de τ_0), la aplicación de esta regla hace descender al rastro de feromona de las conexiones atravesadas por una hormiga. Por tanto, esto resulta en una técnica de exploración adicional del *Sistema de Colonia de Hormigas* al hacer a las conexiones atravesadas por una hormiga menos atractivas para las siguientes hormigas y ayuda a evitar que todas las hormigas sigan el mismo camino.

El procedimiento `nueva_hormiga` y `acciones_demonio` (que en este caso interactúa con el procedimiento `evaporacion_feromona`) para el *Sistema de Colonia de Hormigas* son como sigue…

Procedimiento acciones_demonio asociado al Sistema de Colonia de Hormigas

```
Procedimiento acciones_demonio()
{     para cada Sk hacer
            busqueda_local(Sk); {opcional}
      Smejor-actual←mejor_solucion(Sk);
      si(mejor(Smejor-actual, Smejor-global)) entonces
            Smejor-global←Smejor-actual;
      para cada arista ars∈Smejor-global hacer
      {
            {evaporacion_feromona()=>τrs←(1-ρ)·τrs}
            τrs←τrs+ρ·f(C(Smejor-global));} }
```

Figura 31.a): Estructura del núcleo del algoritmo del Sistema de Colonia de Hormigas

Procedimiento nueva_hormiga asociado al Sistema de Colonia de Hormigas

```
Procedimiento nueva_hormiga(id_hormiga)
{
      k←id_hormiga;
      r←generar_estado_inicial();
      Sk←r;
      Lk←r;
```

```
    mientras (estado_actual≠estado_objetivo) hacer
    {
      para cada s ∈ Nk(r) hacer
        calcular  b_rs ← τ_rs^α · η_rs^β ;
      q←generar_valor_aleatorio_en_[0,1];
      si (q<q0) entonces
        sig_estado←max(brs,Nk(r));
      sino
      {
        para cada s ∈ Nk(r) hacer
                    p_rs^k ← b_rs / Σ_{u∈N_k(r)} b_ru
          sig_estado←aplicar_politica_decision_hormiga
                      (P,Nk(r));
      }
       r←sig_estado; {Sk=<Sk,r>}
       τrs←(1-φ)·τrs+φ·τ0;
       Lk←Lk ∪ r;
    }
    ---
    ---
    ---
    liberar_recursos_hormiga(id_hormiga);
}
```

Figura 31.b): Estructura del núcleo del algoritmo del Sistema de Colonia de Hormigas

Sistema de Hormigas MAX-MIN

El *Sistema de Hormigas MAX-MIN* (*MM*AS) [[SH96],[Stü99a],[SH00]], desarrollado por Stützle y Hoos en 1996, es una de las extensiones de mayor rendimiento del Sistema de Hormigas. Extiende al Sistema de Hormigas básico en los siguientes aspectos…

1. Se aplica una regla de actualización de feromona fuera de línea, similar a la del Sistema de Colonia de Hormigas. Después de que todas las hormigas han construido una solución, se evapora primero cada rastro de feromona…

$$\tau_{rs} \leftarrow (1-\rho)\cdot\tau_{rs} \qquad (24)$$

y posteriormente se deposita feromona según la siguiente expresión…

$$\tau_{rs} \leftarrow \tau_{rs} + f(C(S_{mejor})), \forall a_{rs} \in S_{mejor} \qquad (25)$$

La mejor hormiga a la que se le permite añadir feromona puede ser la solución mejor de la iteración o la mejor solución global. Resultados experimentales han mostrado que se obtiene el mejor rendimiento incrementando gradualmente la frecuencia de la elección de la mejor solución global para la actualización de rastros de feromona [[Stü99a], [SH00]].

Además, en el *Sistema de Hormigas MAX-MIN* típicamente las soluciones de las hormigas son mejoradas usando optimizadores locales antes de la actualización de feromona.

2. Los valores posibles para los rastros de feromona están limitados al rango [τ_{min}, τ_{max}]. La posibilidad del estancamiento del algoritmo es decrementada así al proporcionar a cada conexión alguna probabilidad, aunque muy pequeña, de ser escogida. En la práctica, existen heurísticas para establecer τ_{min} y τ_{max}. Primero, puede mostrarse que, debido a la evaporación de feromona, el nivel máximo posible del rastro de feromona está limitado a $\tau^*_{max}=1/(\rho \cdot C(S^*))$, donde S^* es la solución óptima. Basándonos en este resultado, la mejor solución global puede ser usada para estimar τ_{max} al reemplazar S^* por $S_{mejor\text{-}global}$ en la ecuación para τ^*_{max}. Para τ_{min} es a menudo suficiente elegirlo como algún factor constante inferior a τ_{max} (ver también [SH00] para detalles acerca de posibles formas de definir τ_{min}).

   Como medida para el futuro incremento de la exploración de soluciones, el *Sistema de Hormigas MAX-MIN* también usa la reinicialización ocasional de los rastros de feromona [[SH97], [Stü99a], [SH00]].

3. En vez de inicializar los rastros de feromona a una pequeña cantidad, en el *Sistema de Hormigas MAX-MIN* los rastros de feromona son inicializados a una estimación del máximo valor de rastro de feromona permitido (la estimación puede ser obtenida al generar primero alguna solución S' por una heurística de construcción greedy y posteriormente reemplazar S' en la ecuación para τ^*_{max}). Esto conduce a un componente adicional de diversificación en el algoritmo, porque al principio las diferencias relativas de los rastros de feromona no serán muy marcadas, lo que es diferente de cuando inicializamos los rastros de feromona a algún valor muy pequeño.

La estructura del procedimiento `acciones_demonio` en el *Sistema de Hormigas MAX-MIN* es mostrada debajo…

Procedimiento acciones_demonio asociado al Sistema de Hormigas MAX-MIN

```
Procedimiento acciones_demonio()
{
      para cada Sk hacer
            busqueda_local(Sk);
      Smejor-actual←mejor_solucion(Sk);
      si (mejor(Smejor-actual, Smejor-global)) entonces
            Smejor-global← Smejor-actual;
      Smejor←decision(Smejor-global,Smejor-actual);
      para cada arista ars ∈ Sbest hacer
      {
            τrs←τrs+f(C(Smejor));
            si τrs<τmin entonces
                  τrs←τmin;     }
      si condicion_estancamiento entonces
            para cada arista ars hacer
                  τrs←τmax;     }
```

Figura 32.: Estructura del núcleo del algoritmo del Sistema de Hormigas MAX-MIN

Sistema de Hormigas basado en Rangos

El *Sistema de Hormigas basado en Rangos* (AS_{rank}) [BHS99] es otra extensión del Sistema de Hormigas propuesto por Bullnheimer, Hartl y Strauss en 1997. Incorpora la idea de ranking dentro de la actualización de feromona, que es nuevamente desarrollada fuera de línea por el demonio como sigue…

1. Las *m* hormigas son ordenadas descendentemente según la calidad de sus soluciones: (S_1',…, S_m'), siendo S_1' la mejor solución construida en la generación actual.

2. El demonio deposita feromona en las conexiones atravesadas por las σ-1 mejores hormigas (hormigas elitistas). La cantidad de feromona depositada directamente depende del rango de la hormiga y de la calidad de su solución.

3. Las conexiones atravesadas por la mejor solución global reciben un aporte adicional de feromona que depende de la calidad de esa solución. Este depósito de feromona es considerado el más importante, y por tanto, recibe un peso de σ.

Este modo de operación es puesto en marcha por medio de la siguiente regla de actualización de feromona, que es aplicada a cada arista una vez que los rastros de feromona han sido evaporados…

$$\tau_{rs} \leftarrow \tau_{rs} + \sigma \cdot \Delta\tau_{rs}^{gb} + \Delta\tau_{rs}^{rank}$$

donde $$\Delta\tau_{rs}^{gb} = \begin{cases} f(C(S_{mejor-global})), & \text{si } a_{rs} \in S_{mejor-global} \\ 0, & \text{en otro caso} \end{cases},$$

$$\Delta\tau_{rs}^{rank} = \begin{cases} \sum_{\mu=1}^{\sigma-1} (\sigma-\mu) \cdot f(C(S'_{\mu})), & \text{si } a_{rs} \in S'_{\mu} \\ 0, & \text{en otro caso} \end{cases} \qquad (26)$$

Por tanto, el procedimiento del demonio del *Sistema de Hormigas basado en Rangos* presenta la siguiente estructura…

Procedimiento acciones_demonio asociado al Sistema de Hormigas basado en Rangos

```
Procedimiento acciones_demonio() {
     para cada Sk hacer
          busqueda_local(Sk); {opcional}
     ordenar {S1,...,Sm} en orden decreciente según la
     calidad de la solución para obtener {S1',...,Sm'};
     si (mejor(S1',Smejor-global)) entonces
          Smejor-global← S1';
     para μ←1...(σ-1) hacer
          para cada arista ars ∈ Sμ' hacer
               τrs←τrs+(σ-μ)·f(C(Sμ'));
     para cada arista ars ∈ Smejor-global hacer
               τrs←τrs+σ·f(C(Smejor-global));   }
```

Figura 33.: Estructura del núcleo del algoritmo del Sistema de Hormigas basado en Rangos

Sistema de Hormigas de la Mejor y la Peor Hormiga

El *Sistema de Hormigas de la Mejor y la Peor Hormiga* (BWAS) [CFHM00], propuesto por Cordón et al. en 1999, es un algoritmo de Optimización a partir de Colonias de Hormigas que incorpora conceptos de computación evolutiva. Constituye otra extensión del Sistema de Hormigas, pues usa su regla de transición y mecanismo de evaporación de feromona (la evaporación es aplicada a cada transición como en el Sistema de Hormigas, el Sistema de Hormigas basado en Rangos y el Sistema de Hormigas MAX-MIN). Por otra parte, como ocurre en el Sistema de Hormigas MAX-MIN, el *Sistema de Hormigas de la Mejor y la Peor Hormiga* siempre considera la explotación sistemática de los optimizadores locales para mejorar las soluciones de las hormigas.

En el núcleo del *Sistema de Hormigas de la Mejor y la Peor Hormiga*, se encuentran las tres siguientes acciones del demonio…

1. La regla de actualización de feromona de la mejor y la peor hormiga, que refuerza las aristas contenidas en la mejor solución global. Además, la regla de actualización penaliza cada conexión de la peor solución generada en la iteración actual, $S_{peor\text{-}actual}$, que no está presente en la mejor solución global a través de una evaporación adicional de los rastros de feromona. Por tanto, la regla de actualización del *Sistema de Hormigas de la Mejor y la Peor Hormiga* hace lo siguiente…

$$\tau_{rs} \leftarrow \tau_{rs} + \rho \cdot f(C(S_{mejor-global})), \forall a_{rs} \in S_{mejor-global},$$

$$\tau_{rs} \leftarrow (1-\rho)\cdot\tau_{rs}, \forall a_{rs} \in S_{peor-actual} \text{ y } a_{rs} \notin S_{mejor-global} \qquad (27)$$

2. Se ejecuta una mutación de los rastros de feromona para introducir diversidad en el proceso de búsqueda. Para hacer esto, el rastro de feromona asociado a cada una de las transiciones empezando desde cada nodo (es decir, cada fila de la matriz de rastros de feromona) es mutada con probabilidad P_m al considerar cualquier operador de mutación de codificación real.

   El propuesta original para el *Sistema de Hormigas de la Mejor y la Peor Hormiga* aplicaba un operador que alteraba el rastro de feromona de cada transición mutada añadiendo o restando la misma cantidad en cada iteración. El rango de mutación $mut(it,\tau_{umbral})$, que depende del promedio de rastros de feromona en las transiciones de la mejor solución global, τ_{umbral}, es menos fuerte en las etapas iniciales del algoritmo – cuando no hay riesgo de estancamiento – y más fuerte en las etapas finales, cuando el peligro de estancamiento es más fuerte:

$$\tau'_{rs} \leftarrow \begin{cases} \tau_{rs} + mut(it,\tau_{umbral}), & \text{si } a = 0 \\ \tau_{rs} - mut(it,\tau_{umbral}), & \text{si } a = 1 \end{cases} \qquad (28)$$

   siendo a un valor aleatorio de entre {0, 1} e it la iteración actual.

3. Como otros modelos de Optimización a partir de Colonias de Hormigas, el *Sistema de Hormigas de la Mejor y la Peor Hormiga* considera la reinicialización de los rastros de feromona cuando se estanca, lo que se hace poniendo cada rastro de feromona a un valor τ_0.

El procedimiento `acciones_demonio` se establece en la página siguiente…

Procedimiento acciones_demonio asociado al Sistema de Hormigas de la Mejor y la Peor Hormiga

```
Procedimiento acciones_demonio()
{     para cada Sk hacer
            busqueda_local(Sk);
      Smejor-actual←mejor_solucion(Sk);
      si (mejor(Smejor-actual,Smejor-global)) entonces
            Smejor-global←Smejor-actual;
      para cada arista ars∈Smejor-global hacer
      {
            τrs←τrs+ρ·f(C(Smejor-global));
            suma←suma+τrs;
      }
      τumbral←suma/(|Smejor-global|);
      Speor-actual←peor_solucion(Sk);
      para cada arista ars∈Speor-actual Y ars∉Smejor-global hacer
            τrs←(1-ρ)·τrs;
      mut←mut(it, τumbral);
      para cada nodo/componente r∈{1,...,l} hacer
      {
            z←generar_valor_aleatorio_en_[0,1];
            si z<=Pm entonces
            {
                  s←generar_valor_aleatorio_en_[1,...,l];
                  a←generar_valor_aleatorio_en_[0,1];
                  si (a=0) entonces
                        τrs←τrs+mut;
                  sino
                        τrs←τrs-mut;
            }
      }
      si (condicion_estancamiento) entonces
            para cada arista ars hacer
                  τrs←τ0;}
```

Figura 34.: Estructura del núcleo del algoritmo del Sistema de Hormigas de la Mejor y la Peor Hormiga

Los estudios sobre ciertos problemas han demostrado que la versión del *Sistema de Hormigas de la Mejor y la Peor Hormiga* que incluye las tres componentes ya comentadas produce un mejor rendimiento en la mayoría de los casos. La utilización de sólo una componente o una combinación de dos supone peores resultados. Más aún, se ha propuesto un nuevo modelo en [RT00] de buen rendimiento llamado Sistema de Colonia de Hormigas de la Mejor y la Peor Hormiga, que está basado en la introducción de las tres componentes del *Sistema de Hormigas de la Mejor y la Peor Hormiga* en el Sistema de Colonia de Hormigas.

Estrategias de Intensificación y Diversificación

La intensificación y la diversificación del proceso de búsqueda son conceptos bastante inexplorados en la investigación sobre *Algoritmos de Optimización a partir de Colonias de Hormigas*. Los mecanismos ya existentes pueden ser divididos en dos categorías diferentes. La primera consiste en mecanismos que cambian de alguna forma los valores de feromona, o bien

en línea (por ej., [[DG97], [RT00]]), o al reinicializar los valores de feromona (por ej., [SH00]). La segunda categoría consiste en algoritmos que aplican múltiples colonias e intercambian información entre ellas de alguna forma (por ej., [[MRS00], [KYSO00]]).

En contraposición a esto, existen métodos de intensificación y diversificación, que subrayaremos también a continuación, que están basados en un conjunto de soluciones élite encontradas por el algoritmo en la historia del proceso de búsqueda.

Comenzaremos a comentar las diferentes estrategias…

Reinicialización e inicialización del vector o matriz de feromona

Como primera estrategia para introducir más intensificación y diversificación dentro del proceso de búsqueda se pueden cambiar las funciones de inicialización y reinicialización de los valores de feromona. Se puede usar un valor de inicialización y reinicialización para la feromona no fijo, sino generado uniformemente de manera aleatoria en un intervalo (por ej. [0.25,0.75] en caso de valores de feromona en el intervalo [0,1]). Esto introduce más intensificación, debido a la razón de que en la fase de inicio o reinicialización, el algoritmo se encuentra más enfocado en un área del espacio de búsqueda. Por otra parte, se introduce más diversificación, ya que con cada reinicialización el algoritmo queda enfocado probablemente en un área diferente del espacio de búsqueda.

Listas de soluciones élite

Existen *Algoritmos de Optimización de Colonias de Hormigas* (ACO), que tienen su máxima representación en el Sistema de Hormigas MAX-MIN, que, invierten mucho tiempo para siempre, al final de la fase de reinicialización, moverse hacia la mejor solución encontrada desde el inicio del algoritmo. Después de algún tiempo no se encuentran mejoras alrededor de esta solución. La estrategia de siempre moverse hacia la mejor solución podría trabajar bien para ciertos problemas, como el problema del Viajante de Comercio que presenta una relación fitness-distancia, pero para problemas que carecen de dicha relación se requieren otras estrategias. Para cambiar este esquema mantenemos una lista L_{elite} de soluciones élite encontradas durante la búsqueda para ser manejadas y usadas como se describirá a continuación. L_{elite} trabaja como una lista FIFO (First In First Out) y tiene una longitud máxima l. Hay varias acciones que se pueden realizar sobre la lista…

- *Adición de soluciones a L_{elite}:* Cada solución s_{rb} (mejor solución encontrada desde la última reinicialización) al final de una fase de reinicialización (cuando el algoritmo se estanca) se añade a la lista. Al principio del algoritmo la lista está vacía. En una situación donde la longitud de L_{elite} es más pequeña que l, la nueva solución se añade simplemente. En caso de que la longitud de L_{elite} sea igual a l debemos también borrar el primer elemento de L_{elite}.

- *Uso de soluciones de L_{elite}:* Reemplazamos la convergencia a la mejor solución encontrada por el siguiente esquema. Al final de una fase de reinicialización (cuando el algoritmo se estanca) seleccionamos entre las soluciones de L_{elite}, la que denotaremos por $s_{\text{mas-cercana}}$, que será la más cercana a s_{rb}. Se debe utilizar alguna medida de distancia. Si la codificación es binaria, la medida más común suele ser la distancia de Hamming. En el caso de que $s_{\text{mas-cercana}}$ sea diferente a s_{rb}, s_{rb} es reemplazada por $s_{\text{mas-cercana}}$ y el algoritmo continúa hasta que nuevamente queda estancado. Si durante esta fase se mejora $s_{\text{mas-cercana}}$, ésta es borrada de la lista y la solución mejorada se añade a L_{elite}, como último elemento. Esta fase del algoritmo es llamada en base a este

modo de operación: *fase de convergencia local*. Cuando el algoritmo se estanca en la fase de convergencia local escogemos la mejor solución s_{mejor} entre las soluciones de la lista L_{elite}. Si es diferente de s_{rb}, la mejor solución encontrada durante la fase de convergencia local, reemplazamos s_{rb} con s_{mejor} y proseguimos con el algoritmo. Si durante esta fase s_{mejor} es mejorada, será borrada de la lista L_{elite} y la solución mejorada es añadida a L_{elite}, como último elemento añadido. Esta fase del algoritmo es así llamada *fase de convergencia al mejor*.

El mecanismo comentado implica que una vez que una solución es añadida a L_{elite} tiene $|L_{\text{elite}}|$ (la longitud de L_{elite}) fases de reincialización del algoritmo para ser mejorada. Si es mejorada tiene otra vez el mismo tiempo para ser mejorada nuevamente. Si una solución de L_{elite} no es mejorada durante $|L_{\text{elite}}|$ fases de reinicialización del algoritmo, se elimina de la lista L_{elite} y su vecindario queda establecido como una región explorada del espacio de búsqueda.

La intención detrás del uso de una lista de soluciones élite como se describe arriba será la que a continuación detallaremos. En vez de trabajar sobre un área de búsqueda, nuestro algoritmo trabaja sobre varias áreas en el espacio de búsqueda tratando de mejorar las mejores soluciones encontradas en tales regiones. Si no puede mejorarlas en una cierta cantidad de tiempo las descarta incluso si contienen la mejor solución encontrada desde el inicio del algoritmo. Esto previene al algoritmo de perder tiempo y tiene así un componente de diversificación. Este mecanismo también incorpora un fuerte componente de intensificación al aplicar la fase de convergencia local y la fase de convergencia al mejor.

Además, basándonos en el uso de una lista de soluciones élite, podemos desarrollar dos formas más de reinicializar los valores de feromona. La primera de ellas consiste en dirigirnos hacia una diversificación de la búsqueda en base a las soluciones élite de L_{elite}. En este esquema los valores de feromona se reinicializan como sigue (suponiendo que la feromona se ubica en un vector de tamaño n, siendo n el tamaño del problema, y que los valores se encuentran en el intervalo [0,1])…

$$\tau_i \leftarrow f\left(\frac{\sum_{s \in L_{elite}} s_i}{|L_{elite}|}\right) \text{ donde } f(x) = \begin{cases} 0.5 + x & \text{si } x < 0.25, \\ 1.0 - x & \text{si } 0.25 \le x \le 0.75, \\ x - 0.5 & \text{si } x > 0.75 \end{cases} \qquad (29)$$

La idea aquí es la de concentrar la búsqueda en áreas del espacio de búsqueda diferentes del conjunto actual de soluciones élite.

El otro esquema para reinicializar los valores de feromona permite una intensificación del proceso de búsqueda. Primero se elige la mejor solución entre las soluciones élite s_{mejor}, y después se selecciona otra solución s_{segunda} de manera uniformemente aleatoria de L_{elite} distinta de s_{mejor}. Usando estas dos soluciones la reinicialización de los valores de feromona se realiza como sigue (suponiendo que la feromona se ubica en un vector de tamaño n, siendo n el tamaño del problema, y que los valores se encuentran en el intervalo [0,1])…

$$\tau_i \leftarrow \left(\frac{s_{mejor}(i) + s_{segunda}(i)}{2.0}\right) \qquad (30)$$

Lo ideal para un algoritmo es que en cada fase de reinicialización del vector de feromona selecciona aleatoriamente entre las dos posibilidades que hemos proporcionado: intensificación o diversificación.

Frecuencia y deseabilidad globales

A continuación definiremos los conceptos de deseabilidad global y frecuencia global para los componentes solución. Nos servirán para establecer una posible reinicialización de la búsqueda tras el estancamiento. Suponemos que trabajamos con codificación binaria.

Denotaremos al conjunto de todas las soluciones generadas por las hormigas desde el inicio del algoritmo por S_{hormigas}. La deseabilidad global de una componente solución o_j (j=1…n) es la siguiente (para maximización)…

$$v_j^{des} \leftarrow \max\{f(s) : s \in S_{hormigas}, s_j = 1\} \qquad (31)$$

Si el problema es de minimización f(s) cambiaría por 1/f(s) en la ecuación *2.31*.

La frecuencia global se define como sigue…

$$v_j^{fr} \leftarrow \sum_{s \in S_{hormigas}} s_j \qquad (32)$$

Como podemos comprobar, tenemos dos vectores de tamaño n, siendo n el tamaño del problema, uno para almacenar la deseabilidad global y otro para almacenar la frecuencia global. El objetivo es usar estos dos vectores para reposicionar los valores del vector de feromona. Para reinicializar el vector de feromona se usarán b_1 hormigas para construir soluciones de acuerdo a los valores de deseabilidad global y b_2 hormigas para construir soluciones de acuerdo a las frecuencias globales. La regla de transición que queda para la reinicialización es la usual del Sistema de Hormigas ya conocida pero donde en vez del vector de feromona se utilizan respectivamente los vectores de deseabilidad (para las b_1 hormigas) y frecuencia (para las b_2 hormigas) normalizados.

Obtenidas las $b_1 + b_2$ soluciones obtendremos, tras reinicializar el vector de feromona a sus valores iniciales, aplicando la regla de actualización de feromona correspondiente el vector de feromona reinicializado.

El uso de la deseabilidad global asegura un valor bastante alto de feromona para los elementos solución que han sido encontrados como pertenecientes a buenas soluciones en el pasado (explotación), mientras que el uso de frecuencia global guía a la búsqueda hacia áreas en el espacio de búsqueda que no han sido exploradas tanto (exploración).

6. Implementación

Propuesta la forma de resolver el problema, queda dar el paso hasta la disposición práctica de tal forma de abordarlo. Así, se han seleccionado un conjunto de algoritmos correspondientes a las distintas filosofías de Metaheurísticas, y se han implementado. El objetivo es el de contrastar la teoría con la práctica en una posterior experimentación y análisis de resultados.

Veamos cómo se ha realizado dicha implementación de los distintos algoritmos estableciendo las adaptaciones necesarias a realizar sobre la forma genérica, descrita en secciones anteriores. Se describirán las adaptaciones dependientes del problema y las que han supuesto realizar una elección ante la gran variedad de opciones proporcionadas por el algoritmo genérico.

Así, sobre el conjunto de algoritmos decididos a implementar hemos establecido cuál es la correspondiente traslación a la resolución de nuestro problema con vistas a la posterior

comprobación de los resultados proporcionados por las Metaheurísticas ante el problema a resolver: el problema de la selección de la solución deseada.

7. Algoritmos elaborados para la resolución del problema

La aplicación de las distintas *Metaheurísticas* al problema supone la particularización para cada una de las mismas de los distintos componentes genéricos anteriormente descritos en la estructura de los algoritmos asociados y así mismo la adaptación de cada una de ellas al problema específico que van a resolver. Damos ahora el paso correspondiente a la implementación relacionando problema y *Metaheurísticas*. En primer lugar, relacionaremos la lista de algoritmos implementados para cada una de las *Metaheurísticas*.

Los algoritmos implementados quedan resaltados en negrita. Son un total de 27 algoritmos, cada uno de los cuales con una determinada filosofía y encuadrándose dentro de un tipo de *Metaheurística*.

Una vez enumerados, concretaremos cómo se ha resuelto la estructura y componentes genéricos de cada uno para la implementación. Primero comenzaremos indicando los aspectos aplicables a cualquier problema para cada algoritmo dentro de cada *Metaheurística* y posteriormente veremos aquellos que han necesitado ser adaptados al problema particular que resolvemos: la selección de la solución deseada dentro del problema de resolución de restricciones geométricas.

Los algoritmos se muestran a continuación:

- ***Métodos de trayectoria***
  - Búsqueda Local (LS).
    - **Básica.**
    - **Múltiple (MLS).**
  - Enfriamiento Simulado (SA).
    - **Enfriamiento Simulado Simple.**
    - Enfriamiento Simulado Paralelo (SAP).
      - **Sólo múltiples ejecuciones.**
      - **Estrategia de división.**
  - **Búsqueda Tabú (TS).**
  - Búsqueda Multiarranque.
    - Búsqueda Local Iterativa (ILS).
      - **Simple.**
      - **Basada en Poblaciones.**
    - **Búsqueda basada en Vecindarios Cambiantes (VNS).**
- ***Métodos basados en Poblaciones***
  - Algoritmos Genéticos (GA).
    - **Básico.**
    - **CHC.**
    - Multimodales.

                - **Sharing.**
                - **Continuously updated sharing.**
                - **Clearing.**
                - **Nichos elitista con sharing.**
                - **Multitud con multinichos.**
        - Metaheurísticas basadas en Modelos Probabilísticos.
            - Evolución de Poblaciones mediante Aprendizaje Incremental (PBIL).
                - Básico
                    - **Actualización basada en el mejor.**
                    - **Actualización basada en el mejor y el peor.**
                    - Actualización basada en las M mejores soluciones.
                        - **Actualización con las M.**
                        - **Actualización a partir de solución de consenso.**
                - Compact-GA.
                    - **Básico.**
                    - **Extendido.**
        - Optimización a partir de Colonias de Hormigas (ACO).
            - Sistema de Hormigas.
                - **Básico.**
                - **Con Búsqueda Local.**
            - **Sistema de Hormigas basado en Rangos.**
            - **Sistema de Hormigas MAX-MIN.**
            - **Sistema de Hormigas de la Mejor y la Peor Hormiga.**

8. Aplicación de los aspectos de los algoritmos ajenos a la naturaleza del problema

A continuación veremos como se han particularizado las estructuras y componentes genéricos de las *Metaheurísticas* utilizadas para cada uno de los algoritmos de las mismas que hemos seleccionado para implementar. Veamos primero los elementos comunes a todos los algoritmos de todas las ***Metaheurísticas***...

8.1. Elementos comunes a todas las Metaheurísticas

A la hora de la implementación de los distintos algoritmos seleccionados, tenemos un conjunto de componentes comunes a todos ellos que no dependen del problema particular que abordamos y que a continuación detallamos...

- Selección de la solución inicial.

- Condición de finalización.
- Generación aleatoria.

Selección de la solución inicial

La selección de la solución inicial se llevará a cabo para todos los algoritmos de todas las *Metaheurísticas* de manera aleatoria.

Condición de finalización

La condición de finalización para todos los algoritmos de todas las *Metaheurísticas* va a tener dos vertientes distintas: en base al número de iteraciones y en base al tiempo. La primera de las condiciones de finalización no puede equiparar a los distintos algoritmos ya que el concepto de iteración es distinto de uno a otro. De esta forma, aunque se han puesto disponibles estos dos tipos, la que se utilizará en las ejecuciones para la comparación de los algoritmos será la condición de finalización en base al tiempo. El tiempo permite contrastar resultados, mientras que el número de iteraciones no puede obedecer a un mismo tiempo y puede poner en condiciones de superioridad, sin serlo, a un algoritmo sobre otro.

Generación aleatoria

Para todos los algoritmos tenemos un mismo motor de generación de números aleatorios partiendo de una semilla inicial.

8.2. Elementos específicos para cada algoritmo de cada Metaheurística

A continuación veremos para cada algoritmo de cada ***Metaheurística*** en qué se han particularizado a la hora de la implementación todos los aspectos, componentes y estructura genéricos que fueron establecidos en el capítulo anterior.

Métodos de trayectoria

Búsqueda Local

Además de los elementos comunes ya especificados, el resto de componentes de la *Búsqueda Local* tanto *simple* como *múltiple* corresponden a una adaptación al problema del algoritmo y serán comentados en la siguiente sección 3.2.3. donde se establecerá cómo se adaptan los distintos algoritmos de todas las ***Metaheurísticas*** al problema.

Sólo decir que la *Búsqueda Local simple* es una ascensión de colinas y que la *Búsqueda Local múltiple* queda implementada simplemente como tal: varias búsquedas locales (ascensiones de colinas) encadenadas sin ningún componente de refinamiento.

Enfriamiento Simulado

Dentro de los algoritmos de *Enfriamiento Simulado* se han realizado las siguientes elecciones para la implementación y particularización de la estructura genérica...

Valor inicial del Parámetro de Control

No se ha considerado un valor fijo independiente del problema. Así, se ha seleccionado la propuesta ya indicada en el capítulo anterior (ec. *4.*) para la temperatura inicial que parte del conocimiento de la solución inicial y de los parámetros que indican cómo evolucionará el algoritmo.

Función de enfriamiento de Energía

Se ha utilizado el esquema de Cauchy modificado (ec *8*).

Velocidad de enfriamiento

Se utiliza como valor del número de vecinos a generar un número fijo proporcionado como parámetro al algoritmo.

Temperatura final

Se ha seleccionado un valor muy próximo a cero que, junto con el proporcionamiento como parámetro del número de enfriamientos, permite aplicar el esquema de Cauchy modificado.

Los esquemas paralelos no han necesitado más particularización con respecto a lo establecido que, por parte del *Enfriamiento Simulado Paralelo*, solamente la elección del estado a pasar a cada procesador en la comunicación: en este caso, mejor solución global y mejor solución actual. Así mismo, no se tiene en cuenta la existencia de varios procesadores, sino una ejecución paralela de modo secuencial en un solo procesador.

Búsqueda Tabú

La *Búsqueda Tabú* tiene como sabemos una parte global que dirige la búsqueda y una parte local que inspecciona las distintas regiones a las que se dirige la búsqueda.

La parte global supone la reinicialización de la búsqueda cuando se produce estancamiento. En nuestro caso, dicha reinicialización se realizará en base a tres posibilidades diferentes entre las que elegir probabilísticamente: diversificación (partiendo de una solución aleatoria como solución actual para la siguiente búsqueda a corto plazo), intensificación (partiendo como solución actual de la mejor solución encontrada hasta el momento y cambiando el tamaño de la lista tabú) o una conjunción entre ambas: diversificación e intensificación (basada en la memoria de frecuencias tomando como base la mejor solución encontrada hasta el momento y dirigiendo algunos de sus componentes hacia valores no explorados).

La parte local supone una búsqueda por entornos limitada a partir de la lista tabú. Esa búsqueda por entornos tendrá lugar generando en cada iteración un número fijo de vecinos con los que se tratará en la forma original de la búsqueda a corto plazo: aceptándolos en cualquier caso si son mejores que la mejor solución encontrada y en otro caso aceptándolos como solución actual si no están en la lista tabú de atributos de soluciones generadas recientemente.

Búsqueda Multiarranque

La búsqueda multiarranque en base a los algoritmos implementados tiene dos vertientes: Búsqueda Local Iterativa y Búsqueda por Vecindarios Cambiantes. Veamos que ha ocurrido para cada una de ellas...

Búsqueda Local Iterativa (ILS)

Como ya hemos adelantado, hemos aplicado a nuestro problema la forma básica y la basada en poblaciones de la *Búsqueda Local Iterativa*.

Con respecto a la *Búsqueda Local Iterativa básica* hemos de decir que se ha seguido un esquema sencillo que supone las aplicaciones sucesivas y alternativas de Búsqueda Local y mutación sobre la solución actual. La Búsqueda Local nos permite escoger el mejor individuo

de la región en que nos encontramos y la mutación nos permite trasladarnos a otra región no explorada.

Por otra parte, en lo que se refiere a la *Búsqueda Local Iterativa basada en poblaciones*, tenemos la incorporación de una Búsqueda Local Iterativa básica, en la forma ya comentada, sobre cada individuo de la población en cada iteración, para luego concentrar a la población cada vez más hacia la mejor solución, reemplazando el peor, por el mejor encontrado hasta el momento, en cada iteración.

Búsqueda basada en Vecindarios Cambiantes (VNS)

Para la *Búsqueda basada en Vecindarios Cambiantes* hemos utilizado la forma básica, en la que los cambios corresponden a la adaptación del algoritmo al problema y que posteriormente comentaremos.

Métodos basados en poblaciones

Algoritmos Genéticos

Dentro de los *Algoritmos Genéticos* se han implementado la versión básica, el CHC y varios con características multimodales. Veamos cada uno de ellos a continuación...

Básico

Dentro del *Algoritmo Genético* cabe comentar para la particularización a la implementación del algoritmo genérico solamente la elección del mecanismo de selección. Existen otros componentes que cambian en dicha particularización, pero vienen dados por la adaptación del algoritmo al problema y serán posteriormente comentados.

En lo que se refiere al mecanismo de selección, hemos de decir que, hemos seleccionado el Muestreo Universal Estocástico propuesto por Baker en [Bak87] en el cual, el número de copias de cada individuo en la población intermedia está acotado inferior y superiormente por un número de copias esperado, calculado en función de su adaptación. Así mismo, completamos el mecanismo de selección por el modelo de selección elitista, basado en mantener el individuo mejor adaptado de la población anterior, en la nueva población, la obtenida después de llevar a cabo el proceso de selección y aplicar los operadores de cruce y mutación, [Gol89].

CHC

El algoritmo *CHC* ha sido implementado en base a la estructura proporcionada en el capítulo anterior con las únicas adaptaciones que corresponden a las características del problema que pretendemos resolver y que se comentarán, como ya se dijo, posteriormente.

Algoritmos Genéticos con nichos

Los modelos implementados de los *Algoritmos Genéticos con nichos* reciben la particularización correspondiente al Algoritmo Genético básico en base a las características propias de cada modelo y la que recibirán de su adaptación al problema. La estructura que siguen fue detallada en el capítulo anterior. Hay que añadir que se ha tendido a la simplicidad en la implementación sin procesos de adaptación de parámetros, sino estableciéndolos fijos, como es el caso del radio del nicho.

Metaheurísticas basadas en Modelos Probabilísticos

Las *Metaheurísticas basadas en Modelos Probabilísticos* seleccionadas para aplicar a nuestro problema están todas ellas basadas en la *Evolución de Poblaciones mediante Aprendizaje Incremental (PBIL)*. Han sido implementadas tal y como fueron descritas, con el pequeño inciso para los algoritmos que actualizan a partir de las M soluciones mejores de que, dicha actualización sigue la misma regla que la versión básica, pero en un caso actualizando para M mejores soluciones en vez de para una y en otro caso actualizando para una solución (solución de consenso) que recoge los componentes predominantes en las M mejores soluciones.

Hay que añadir que, se ha establecido para todos la posibilidad de reinicialización básica. Dicha reinicialización consiste en que, una vez comprobada la llegada al estancamiento: la estabilización del vector de probabilidades, el vector de probabilidades queda inicializado como al principio del algoritmo y proseguimos la ejecución.

Algoritmos de Optimización a partir de Colonias de Hormigas

Todos los algoritmos de Optimización a partir de Colonias de Hormigas siguen el esquema descrito para cada uno con la gran particularidad de que no existe factor de paralelismo. Sin embargo, dicho esquema sufre el gran cambio con la adaptación al problema, algo que veremos en la siguiente sección.

9. Adaptación de los aspectos de los algoritmos que dependen de la naturaleza del problema

Como sabemos, e indicamos a la hora de describir el problema a resolver, un sistema con restricciones normalmente no define a un único objeto, sino que independientemente del enfoque utilizado para resolverlo, vamos a encontrar un conjunto amplio de soluciones u objetos que lo cumplen. El problema que subyace aquí una vez resuelto el que parece fundamental es el de la selección de la solución deseada. La resolución de dicho problema fue enunciada en una doble vertiente: selección por similitud con el croquis y selección mediante restricciones adicionales. De la segunda de ellas emana la reformulación del problema de la selección de la solución deseada como un problema de optimización combinatoria.

En el capítulo anterior vimos distintas *Metaheurísticas* para resolver problemas de optimización combinatoria. Llega la hora de ver cómo aplicar las distintas *Metaheurísticas* descritas anteriormente a nuestro problema, en este caso en lo que se refiere a los aspectos que dependen de la naturaleza del problema, ya que aquellos que son independientes fueron analizados en la sección anterior.

En primer lugar, comentaremos los elementos comunes a todas las *Metaheurísticas* que del problema se extraen. Posteriormente, pasaremos a describir para cada *Metaheurística* en particular lo que supone la aplicación de la misma al problema en cuanto a su estructura, evolución y componentes. Tomaremos como referencia las descripciones básicas y genéricas dadas para cada ***Metaheurística*** en el capítulo anterior.

9.1. Elementos comunes en la aplicación de las distintas Metaheurísticas al problema

Los elementos comunes en la aplicación de las distintas ***Metaheurísticas*** al problema son fundamentalmente cinco:

- La codificación de las soluciones.

- La función de evaluación de soluciones.
- Comparación entre soluciones.
- Factibilidad de soluciones.
- El concepto de solución vecina (para aquellas ***Metaheurísticas*** que generan de una u otra forma vecinos de las soluciones para evolucionar en la búsqueda, que son muchas de las descritas).
- Medida de distancia (para aquellas ***Metaheurísticas*** que la utilizan).

La codificación de las soluciones

Sabemos que el hecho de que los planes de construcción, en la resolución de un problema de resolución de restricciones geométricas, estén compuestos por multifunciones implica que, en general, existe la posibilidad de poder seleccionar diferentes instancias solución. Así, una vez que se han asignado valores a los parámetros de las restricciones, sabemos que podemos considerar que las diferentes instancias posibles definen un árbol binario. Cada nodo de ese árbol de decisión representa un punto donde tomamos una decisión, donde escogemos un hijo de los dos posibles.

De esta forma, una solución a nuestro problema, la solución seleccionada por el usuario, vendrá dada por una secuencia de decisiones. Dichas decisiones suponen el haber escogido en cada momento entre dos opciones posibles. Así, considerando *n* el número de decisiones, podemos representar una solución como una secuencia (vector) de ceros y unos. El valor cero representará una rama en la decisión y el valor uno la otra rama. Tenemos así representada la secuencia ordenada de decisiones tomadas. Se escoge por tanto una codificación binaria para representar a las soluciones.

La función de evaluación de soluciones

Nuestra solución para resolver el problema de la selección de la solución deseada se basa en una idea aparentemente simple. Se trata de añadir al problema bien restringido un conjunto adicional de restricciones que la solución deseada debería cumplir. Nos quedaríamos como solución con aquella de las obtenidas de la resolución del problema que cumpla además el máximo número de predicados definidos por el usuario como deseables para su solución requerida.

Siendo esta la base para resolver nuestro problema visto como un problema de optimización combinatoria, se establece claramente que la función de evaluación de soluciones es la que se establece a continuación y que ya adelantamos en el primer capítulo en la descripción de la selección como un problema de optimización combinatoria...

donde

$$\delta(P_i(n)) = \begin{cases} 0 & \text{si no se cumple el predicado } P_i \\ 1 & \text{si se cumple el predicado } P_i \end{cases} \qquad (33)$$

Como vemos la función de evaluación calcula el número de predicados adicionales definidos por el usuario que cumple la solución que se le proporcione como entrada.

Comparación de soluciones

La comparación de soluciones deriva de la definición de la función de evaluación de soluciones. Como hemos indicado anteriormente la función de evaluación calcula el número de

predicados adicionales definidos por el usuario que cumple cada solución. Cuantos más predicados cumpla la solución mejor será y por tanto nos encontramos ante un problema de maximización. De ahí que en la comparación de soluciones a la hora de evolucionar en los distintos algoritmos debamos de tener en cuenta este objetivo fundamental de llegar a la solución de mayor coste de función de evaluación.

Factibilidad de soluciones

Es posible que una serie de decisiones a la hora de construir una solución nos conduzcan a una solución no factible. Esto debe ser tenido en cuenta en la función de evaluación asociando una cláusula correspondiente a este caso. Varias son las posibilidades. De ellas hemos seleccionado la más sencilla, aquella que asocia un valor de la función de evaluación inferior a la de todas las soluciones factibles posibles. Dicho valor debe ser obviamente inferior a 0, pues el mínimo número de predicados adicionales definidos por el usuario que una solución puede cumplir es ninguno. Así, se ha escogido el valor -1 para asociarlo a soluciones no factibles, valor que, dadas las características del problema (maximización), no interfiere en la evolución de los distintos algoritmos ni supone cambios adicionales.

Concepto de solución vecina

Dada una solución *s*, decimos que *s′* es solución vecina de *s* si podemos llegar a *s′* a partir de *s* mediante un operador de vecindario. Un operador de vecindario es una función que realiza un pequeño cambio sobre una solución y permite obtener otra cercana en el espacio de búsqueda a la que recibe.

Para las soluciones que presentan codificación binaria, se considera solución vecina a aquella que resulta de cambiar un pequeño número de componentes en la solución de partida por sus complementarios, o sea, aquella que proviene de cambiar en la solución de partida algunos componentes con valor cero por valor uno o viceversa. Normalmente, el número de componentes que se cambian suelen ser uno o dos.

Medida de distancia

Para medida de distancia se utilizará la correspondiente a la codificación binaria y normalmente más aplicable: la distancia de Hamming.

9.2. Elementos específicos en la aplicación de las distintas Metaheurísticas al problema

Métodos de trayectoria

Búsqueda Local básica y Búsqueda Local múltiple

Para la *Búsqueda Local básica* y *Búsqueda Local múltiple* sobre nuestro problema nos basta con la definición de los elementos comunes ya realizada para completar la aplicación de la misma a la selección de la solución deseada.

Enfriamiento Simulado

Para el *Enfriamiento Simulado* ocurre igual que con la Búsqueda Local en sus dos vertientes en lo que se refiere al cambio de la ***Metaheurística*** genérica a la ***Metaheurística*** adaptada al problema.

Búsqueda Tabú

Dentro de la *Búsqueda Tabú* se establecía la existencia de dos tipos de memoria para guiar la búsqueda: memoria a corto plazo y memoria a largo plazo. La primera de ellas permitía calificar a ciertas soluciones como tabú para permitir dirigir la búsqueda hacia otras regiones distintas a las ya exploradas. La segunda de ellas permitía guardar la historia de la búsqueda para la reinicialización de la búsqueda una vez que se produjera el estancamiento.

Dada la codificación seleccionada para el problema, dichas memorias adquieren forma propia que se traduce para la memoria a corto plazo en un vector de tamaño la tenencia tabú que indica las posiciones recientes que han cambiado de valor en la generación de vecinos (una solución es tabú cuando se obtiene a partir de otra utilizando un cambio de valor en una posición establecida en la lista tabú o memoria de corto plazo); en cambio, para la memoria a largo plazo esto se traduce en el almacenamiento en un vector de n posiciones, siendo n el tamaño del problema, del número de veces que ha sido cambiada cada posición en la evolución del algoritmo en la búsqueda tabú a corto plazo.

Así mismo, en lo que se refiere al procedimiento de reinicialización, hemos de tener en cuenta que, dado que en él puede intervenir la memoria de largo plazo, el proceso correspondiente a dicha intervención deberá cambiar en el sentido del significado que tiene tal memoria. De esta forma, en la reinicialización se pretende crear a partir de la memoria de frecuencias (o de largo plazo) una solución que se aleje de lo que frecuentemente ha sido generado a lo largo de la ejecución del algoritmo: se generará una solución partiendo de la mejor solución encontrada hasta el momento en la que habrá mayor probabilidad de cambiar las posiciones que hayan sido cambiadas menor número de veces durante la búsqueda.

Búsqueda Multiarranque

Dentro de la *Búsqueda Multiarranque*, además de la ya comentada Búsqueda Local múltiple, se establecen la Búsqueda Local Iterativa y la Búsqueda basada en Vecindarios Cambiantes. Veamos a continuación los aspectos de estas ***Metaheurísticas*** que se particularizan para la resolución de nuestro problema.

Búsqueda Local Iterativa (ILS)

Con respecto a la *Búsqueda Local Iterativa* (ILS), hay que decir con respecto a las modificaciones a realizar en la misma para su adaptación a la resolución de nuestro problema que, éstas se centran en el proceso de mutación de soluciones.

Nuevamente, la codificación de las soluciones condiciona un proceso: en este caso el de mutación. Existen múltiples posibilidades de realizar una mutación sobre codificación binaria. De ellas hemos escogido aquella que, en base a una probabilidad de mutación, permite cambiar o no cada posición a partir de la generación de un aleatorio.

La mutación es idéntica tanto para la forma básica como para la que parte de una población para la exploración del espacio de búsqueda.

Búsqueda basada en Vecindarios Cambiantes (VNS)

Con respecto a la *Búsqueda a partir de Vecindario Variable* (VNS), decir que encontramos el cambio, debido a la necesidad de resolución de nuestro problema, en la estrategia de caracterización de vecindario. Así, y muy influido por la codificación del problema, el vecindario viene dado por un parámetro de mutación de solución variable. La generación de un vecino se realiza por mutación en base al parámetro comentado tal y como

ocurre en ILS, parámetro que irá cambiando a lo largo del algoritmo en base a las características y estructura del algoritmo ya conocida.

Métodos basados en poblaciones

Algoritmos Genéticos

Algoritmo Genético básico

Dentro del *Algoritmo Genético básico*, la adaptación al problema supone la definición de los operadores de cruce y mutación. Teniendo en cuenta la codificación de las soluciones o cromosomas, en este caso de la población, se limitan las posibilidades de elección de operador de cruce y mutación.

Como operador de cruce hemos seleccionado el cruce simple en un punto, consistente en la selección de una posición del cromosoma e intercambio del resto de cromosoma a partir de esa posición con ese mismo resto en el cromosoma con que se cruza. Tenemos dos cromosomas de partida y generamos dos cromosomas que han intercambiado su contenido a partir de un punto seleccionado y que el resto lo mantienen como sus padres.

Como operador de mutación se ha escogido aquel que, basado en una probabilidad de mutación, cambia los valores de las posiciones del cromosoma a sus complementarios en base al lanzamiento de un aleatorio para cada posición.

Son los operadores binarios básicos para trabajo con cromosomas.

CHC

Dado que el algoritmo *CHC* es un algoritmo definido inicialmente en su estructura para codificación binaria y ésta misma es la que utilizamos para nuestro problema, no necesitamos adaptación alguna, más que seguir la estructura ya definida en el capítulo anterior.

Algoritmos Genéticos con nicho

Con la adaptación del Algoritmo Genético básico y con los elementos comunes ya adaptados para todos los algoritmos, teniendo en cuenta las características propias de cada algoritmo multimodal, tenemos dispuestos a todos los *Algoritmos Genéticos con nicho* para abordar nuestro problema.

Metaheurísticas basadas en Modelos Probabilísticos

Las *Metaheurísticas basadas en Modelos Probabilísticos* que se describieron en el capítulo anterior y que van a ser utilizadas para abordar la resolución de nuestro problema son PBIL (Evolución de poblaciones mediante aprendizaje incremental) y sus derivados. Se basan en la codificación binaria, de tal forma que planteada la resolución de nuestro problema de esta forma, ningún cambio sobre la filosofía general es necesario aplicar.

Algoritmos de Optimización a partir de Colonias de Hormigas

Dada la codificación usada: codificación binaria, a la hora de aplicar los distintos Algoritmos de Optimización a partir de Colonias de Hormigas a nuestro problema hemos de utilizar una implementación de los mismos asociada a los valores binarios. Esta implementación, que define explícitamente el hiperespacio para los valores de feromona como el cierre convexo del conjunto de soluciones binarias factibles del problema bajo consideración,

recibe el nombre de marco de trabajo de hipercubo para los Algoritmos de Optimización a partir de Colonias de Hormigas (HC-ACO).

HC-ACO establece el almacenamiento de los rastros de feromona en un vector con tantas componentes como tiene el vector solución. Dicho vector obedece a una distribución de probabilidad sobre las componentes de la solución y cada una de sus componentes tomará valores en el intervalo [0,1]. Cada componente expresa la probabilidad de que haya 1 como valor en la correspondiente componente del vector solución.

Teniendo en cuenta esto, hay que reescribir las reglas de actualización de feromona, que quedarán como sigue…

<u>*Sistema de Hormigas y Sistema de Hormigas con Búsqueda Local*</u>

$$\tau_j \leftarrow \rho \cdot \tau_j + (1-\rho) \cdot \sum_{h=1}^{Num\_hormigas} g(s_h, j)$$

donde

$$g(s_h, j) = \begin{cases} \dfrac{f(s_h)}{\sum_{k=1}^{Num\_hormigas} f(s_k)} & si\_s_h(j) = 1 \\ 0 & en\_otro\_caso \end{cases} \qquad (34)$$

Aquí tenemos que…

j: es el número de componente.

h: es el número de hormiga.

τ_j: es la componente *j* del vector de feromona.

ρ: es un parámetro del intervalo [0,1].

s_h: es el vector solución correspondiente a la hormiga *h*.

f: es la función de evaluación de soluciones.

<u>*Sistema de Hormigas basado en Rangos*</u>

$$\tau_j \leftarrow \rho \cdot \tau_j + (1-\rho) \cdot \left(\sum_{h=1}^{M-1} g(s_h, j, h, M, s_{mejor}) + g(s_{mejor}, j, 0, M, s_{mejor})\right)$$

donde

$$g(s_h, j, h, M, s_{mejor}) = \begin{cases} \dfrac{(M-h) \cdot f(s_h)}{\sum_{k=1}^{M-1} (f(s_k) \cdot (M-k)) + f(s_{mejor}) \cdot M} & si\_s_h(j) = 1 \\ 0 & en\_otro\_caso \end{cases} \qquad (35)$$

Aquí tenemos que…

j: es el número de componente.

h: es el número de orden de hormiga en base a su fitness.

τ_j: es la componente *j* del vector de feromona.

ρ: es un parámetro del intervalo [0,1].

s_h: es el vector solución correspondiente a la hormiga *h*.

s_{mejor}: es la mejor solución encontrada.

f: es la función de evaluación de soluciones.

M: es el número de mejores soluciones escogidas de entre las soluciones generadas.

Sistema de Hormigas MAX-MIN

$$\tau_j \leftarrow \rho \cdot \tau_j + (1-\rho) \cdot s_{mejor}(j) \qquad (36)$$

Aquí tenemos que…

j: es el número de componente.

τ_j: es la componente *j* del vector de feromona.

ρ: es un parámetro del intervalo [0,1].

s_{mejor}: es la mejor solución encontrada hasta el momento.

Sistema de Hormigas de la Mejor y la Peor Hormiga

$$\tau_j \leftarrow \rho \cdot \tau_j + (1-\rho) \cdot s_{mejor}(j)$$

si $s_{mejor}(j) \neq s_{peor}(j)$

$$\tau_j \leftarrow \rho \cdot \tau_j + (1-\rho) \cdot s_{mejor}(j) \qquad (37)$$

Aquí tenemos que…

j: es el número de componente.

τ_j: es la componente *j* del vector de feromona.

ρ: es un parámetro del intervalo [0,1].

s_{mejor}: es la mejor solución generada en la iteración actual.

s_{peor}: es la peor solución generada en la iteración actual.

Las reglas de actualización de feromona siguen la filosofía de las ya enunciadas en el capítulo anterior, pero dentro de la normalización para mantener los valores de feromona en el intervalo [0,1].

Como podemos comprobar, la evaporación del vector de feromona se fusiona ahora en la regla de actualización de feromona, pues no puede existir evaporación de feromona como tal en el vector de feromona que representa una distribución de probabilidad.

Así mismo, hemos de hablar de otro aspecto importante de este tipo de algoritmos y que hemos implementado para cada uno de los modelos: la reinicialización. Así, hemos utilizado la reinicialización ya comentada en el capítulo anterior correspondiente a la lista de soluciones élite y a la reinicialización e inicialización del vector de feromona para el *Sistema de Hormigas*

MAX-MIN. Para el resto de algoritmos se ha utilizado la reinicialización basada en la deseabilidad y frecuencia globales también comentada en dicho capítulo.

10. Conclusiones

Existen varias formas de abordar el problema de la selección de la solución deseada. En este trabajo nos introducimos en el problema desde una nueva alternativa: *abordando el problema como un problema de optimización combinatoria*. Para ello, nos introducimos en un conjunto de algoritmos nuevos surgidos en los últimos años con el objeto de resolver los problemas de optimización combinatoria que revisten gran complejidad: las *Metaheurísticas*. El conjunto de métodos y técnicas que agrupan, suponen abordar los problemas con garantías de soluciones de calidad en poco tiempo. En teoría es lo que necesitamos para una posible aplicación práctica en tiempo real a la resolución del problema planteado.

De esta forma, se escogió una clasificación amplia de las *Metaheurísticas* y se describieron gran parte de las filosofías y algoritmos dentro de cada filosofía que éstas aglutinan. Se consideraron las *Metaheurísticas* divididas en dos grupos *Métodos de Trayectoria* y *Métodos basados en Población*. En los primeros, se distinguieron algoritmos correspondientes a la filosofía de *Búsqueda Local, Enfriamiento Simulado*, *Búsqueda Tabú* y *Búsqueda Multiarranque*. Para los segundos se dispusieron algoritmos integrados dentro de tres grandes técnicas: *Algoritmos Genéticos*, *Algoritmos Evolutivos basados en Modelos Probabilísticos y Optimización a partir de Colonias de Hormigas*. Cada técnica y filosofía ha sido analizada convenientemente, así como cada uno de los algoritmos que la constituían, planteándose para éstos una posible implementación.

De cada uno de los métodos y de cada una de las técnicas y filosofías dentro de estos métodos se han seleccionado un conjunto bastante amplio y representativo de algoritmos que finalmente han sido implementados. Para dicha implementación se han hecho las correspondientes elecciones específicas del problema a resolver y genéricas asociadas a la filosofía del algoritmo. Se han seleccionado un total de 27 algoritmos: 9 correspondientes a *Métodos de Trayectoria* y 18 correspondientes a *Métodos basados en Población*.

En trabajos posteriores se realizarán una serie de experimentos que nos permitan determinar cuál o cuáles de estas técnicas se ajusta más al problema, en base a los resultados obtenidos.

Referencias

[AKL97] E.H.L. Aarts, J.H.M. Korst, and P.J.M. van Laarhoven. Simulated annealing. In Emile H.L. Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, 91-120.Wiley-Interscience, Chichester, England, 1997.

[AL97] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience, 1997.

[Bak85] J. E. Baker, *Adaptative selection methods for genetic algorithms*. Proceedings of the first International Conference on Genetic Algorithms and their applications. John J. Grefenstette (Ed.). Lawrence Erlbaum. New Jersey. 101-111, 1985.

[Bak87] J.E. Baker, *Reducing bias and inefficiency in the selection algorithm*. Proceeding of the Second International Conference on Genetic Algorithms and their applications. John J. Grefenstette (Ed.). Lawrence Erlbaum Associates, Hillsdale, MA, 14-21, 1987.

[Bal94] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report No. CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[Bar97] Y. Bar-Yam. *Dynamics of Complex Systems*. Studies in nonlinearity. Addison-Wesley, 1997.

[BC95] S. Baluja and R. Caruana. Removing the Genetics from the Standard Genetic Algorithm. In A. Prieditis and S. Russell, editors, *The International Conference on Machine Learning 1995*, 38-46, San Mateo, California, 1995. Morgan Kaufmann Publishers.

[BDT99] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, NY, 1999.

[BHS99] B. Bullnheimer, R.F. Hartl, and C. Strauss. A new rank-based version of de Ant System: A computational study. *Central European Journal for Operations Research and Economics*, 7(1):25-38, 1999.

[BIV97] J.S. de Bonet, C.L. Isbell Jr., and P. Viola. MIMIC: Finding optima by estimating probability densities. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, vol. 9, pag. 424. The MIT Press, 1997.

[BP97] R. Battiti and M. Protasi. Reactive Search, a history-base heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 1997.

[BSPV02] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. in W. B. Langdon et al., editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pags 11-18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[BT94] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2):126-140, 1994.

[Cer85] V. Cerny. A thermodinamycal approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41-51, 1985.

[CFH02] O. Cordón, I. Fernández de Viana, and F. Herrera. Analysis of the Best-Worst Ant System and its variants on the QAP. In M. Dorigo, G. Di Caro, and M. Sampels, editors, *Ant Algorithms*, vol. 2463 of *Lecture Notes in Computer Science*, pags 228-234. Springer Verlag, Berlin, Germany, 2002.

[CFHM00] O. Cordón, I. Fernández de Viana, F. Herrera, and L. Moreno. A new ACO model integrating evolutionary computation concepts: The best-worst Ant System. In M. Dorigo, M. Middendorg, and T. Stützle, editors, *Abstract proceedings of ANTS2000 - From Ant Colonies to Artificial Ants: A Series of International Workshops on Ant Algorithms*, pags 22-29. IRIDIA, Université Libre de Bruxelles, Belgium, 2000.

[CVS95] W. Cedeño, V.R. Vemuri and T. Slezak. *Multiniche crowding in genetic algorithms and its application to the assembly of DNA restriction-fragments*. Evolutionary computation, 2, 321-345.

[DC99] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pags 11-32. McGraw-Hill, 1999. También disponible como Technical Report IRIDIA/99-1, Université Libre de Bruxelles, Belgium.

[DCG99] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ants algorithms for discrete optimization. *Art. Life*, 5(2):137-172, 1999.

[Dev89] R.L. Devaney. *An introduction to chaotic dinamycal systems.* Addison-Wesley, Second Edition, 1989.

[DG97] M. Dorigo and L.M. Gambardella. Ant Colony System: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53-66, 1997.

[DLM99] M. Dell'Amico, A. Lodi, and F. Maffioli. Solution of the Cumulative Assignment Problem with a well-structured Tabu Search method. *Journal of Heuristics*, 5:123-143, 1999.

[DMC96] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 26(1):29-41, 1996.

[DS02] M. Dorigo and T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications and advances. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2002.

[DS03] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2003.

[ER97] A.E. Eiben and Z. Ruttkay. Constraint satisfaction problems. In T. Bäck, D. Fogel, y M. Michalewicz, editors, *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press, 1997.

[ERR94] A.E. Eiben, P.-E. Raué, and Z. Ruttkay. Genetic algorithms with multi-parent recombination. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, vol. 866 of *Lecture Notes in Computer Science*, pags 78-87, Berlin, 1994. Springer.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications*. John Whiley, 1968.

[Fle95] M. Fleischer. Simulated Annealing: past, present and future. In C. Alexopoulos, K. Kang, W.R. Lilegdon, and G. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pags 155-161, 1995.

[Fog62] L.J. Fogel. Toward inductive inference automata. In *Proceedings of the International Federation for Information Processing Congress*, pags 395-399, Munich, 1962.

[FOW66] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.

[GADP89] S. Goss, S. Aron, J.L. Deneubourg, and J.M. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76:579-581, 1989.

[GDK91] D.E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, La Jolla, CA, 1991. Morgan Kaufmann.

[GJ79] M.R. Garey and D.S. Johnsonn. *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman, 1979.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Comp. Oper. Res.*, 13:533-549, 1986.

[Gol89] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA, 1989.

[GR87] D.E. Goldberg and J. Richardson. *Genetic algorithms with sharing for multimodal function optimization*. En Proc. of the Second International Conference on Genetic Algorithms, 41-49, 1987.

[Har99] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report No. 99010, IlliGAL, University of Illinois, 1999.

[Han86] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[HK00] A. Hertz and D. Klober. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126:1-12, 2000.

[HLV98] F. Herrera, M. Lozano, y J.L. Verdegay. *Algoritmos Genéticos: Fundamentos, extensiones y aplicaciones.* Arbor CLII 597, 9-40, 1998.

[HM99] P. Hansen and N. Mladenovic. An introduction to variable neighborhood search. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Metaheuristics: advances and trends in local search paradigms for optimization*, chapter 30, pags 433-458. Kluwer Academic Publishers, 1999.

[HM01] P. Hansen and N. Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449-467, 2001.

[Hol75] J.H. Holland. *Adaption in natural and artificial systems*. The University of Michigan Press, Ann Harbor, MI, 1975.

[Ing96] L. Ingber. Adaptive Simulated Annealing (ASA): Lessons learned. *Control and Cybernetics - Special Issue on Simulated Annealing Applied to Combinatorial Optimization*, 25(1):33-54, 1996.

[Kem96] C.H.M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature - PPSN IV*, vol. 1141 of *Lecture Notes in Computer Science*, pags 494-503, Berlin, 1996. Springer.

[KG89] K. Keb and D.E. Goldberg. *An investigation of niche and species formation in genetic function optimization.* En Proc. Third International Conference on Genetic Algorithms (ICGA '89), 42-50, 1989, Hillsdale.

[KGV83] S. Kirkpartick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science, 13 May 1983,* 220(4598):671-680, 1983.

[KYSO00] H. Kawamura, M. Yamamoto, K. Suzuki, and A. Ohuchi. Multiple Ant Colonies Algorithm Based on Colony Level Interactions. *IEICE Transactions on Fundamentals*, E83-A(2):371-379, February 2000.

[LM86] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming,* 34(1):111-124, 1986.

[Mah92] S.W. Mahfoud. *Crowding and preselection revisited*. *Parallel solving from nature II*. R. Männer and B. Manderick, editors, Elsevier, 27-36, 1992.

[Mah93] S.W. Mahfoud. *Symple analytical models of genetic algorithms for multimodal function optimization*. IlliGAL Report 93001, Illinois Genetic Algorithms Laboratory, Universidad de Illinois, Illinois, EEUU, 1993.

[Mah95] S.W. Mahfoud. *Niching methods for genetic algorithms*. PhD dissertation, University of Illinois, Urbana-Champaign, 1995.

[Mit98] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, 1998.

[MM98] R. Michel and M. Middendorf. An island model based Ant System with lookahead for the shortest supersequence problem. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of PPSN-V, Fifth International Conference on Parallel Problem Solving from Nature*, vol 1498 of *Lecture Notes in Computer Science*, pags 692-701. Springer Verlag, Berlin, Germany, 1998.

[MOF91] O. Martin, S.W. Otto, and E.W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299-326, 1991.

[Mos89] P. Moscato. On evolution, search, optimization, genet algorithms and martial arts: Toward memetic algorithms. Tech. Rep. Caltech Concurrent Computation Program 826, California Institute of Technology, Pasadena, California, USA, 1989.

[Mos99] P. Moscato. Memetic algorithms: A short introduction. In F. Glover, D. Corne and M. Dorigo, editors, *New Ideas in Optimization*. McGraw-Hill, 1999.

[MRS00] M. Middendorf, F. Reischle, and H. Schmeck. Information Exchange in Multi Colony Ant Algorithms. In *Proceedings of the Workshop on Bio-Inspired*

Solutions to Parallel Processing Problems, LNCS 1800, pags 645-652. Springer Verlag, 2000.

[MS96] B.L. Miller and M.J. Shaw, *Genetics algorithms with dynamic niche sharing for multimodal function optimization*. En Proc. 1996 IEEE Int. Conf. Evolutionary Computation. Piscataway, NJ: IEEE Press, pags 786-791, 1996.

[Müh91] H. Mühlenbein. Evolution in time and space - the parallel genetic algorithm. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Mateo, USA, 1991.

[NW88] G.L. Nemhauser and A.L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[OGC91] C.K. Oei, D.E. Goldberg and S.J. Chang. *Tournament selection, niching and the preservation of diversity*. IlliGAL Report No. 91011. University of Illinois, 1991.

[OL96] I.H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:513-623, 1996.

[Osm93] I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421-451, 1993.

[Pet96] A. Pètrowski. *Clearing procedure as a niching method for genetic algorithms.* En Proc. IEEE International Conference on Evolutionary Computation, Nagoya, Japan, 798-803, 1996.

[Pet97] A. Pètrowski. *A new selection operator dedicated to speciation*. Proceedings of the Seventh International Conference on Genetic Algorithms. T. Back, editor, Kaufmann. San Mateo, 144-151, 1997.

[PDG87] J.M. Pasteels, J.-L.Deneubourg, and S. Goss. Self-organization mechanisms in ant societies (I): Trail recruitment to newly discovered food sources. *Experientia Supplementum*, 54:155-175, 1987.

[PGC99] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, vol I, pags 525-532, Orlando, FL, 13-17 1999. Morgan Kaufmann Publishers, San Francisco, CA.

[PGL99] M. Pelikan, D.E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. Technical Report No. 99018, IlliGAL, University of Illinois, 1999.

[PHH01] E. Pérez, F. Herrera, and C. Hernández. *Finding multiple solutions in job shop scheduling by niching genetic algorithms*. Technical Report #DECSAI-010110, Dept. of Computer Science and Artificial Intelligence, University of Granada, Spain, 2001.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications, Inc., New York, 1982.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.

[RMS02] H. Ramalhino Lourenco, O. Martin, and T. Stützle. Iterated Local Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, 2002.

[RT00] M. Randall and E. Tonkes. Intensification and Diversification Strategies in Ant Colony System. Technical Report TR00-02, School of Information Technology, Bond University, 2000.

[SD99] T. Stützle and M. Dorigo. ACO algorithms for the traveling salesman problem. In K. Miettinen, M.M. Mäkelä, P. Neittaanmäki, and J. Périaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pags 163-183. John Wiley & Sons, Chichester, UK, 1999.

[SH96] T. Stützle and H.H. Hoos. Improving the Ant System: A detailed report on the MAX-MIN Ant System. Technical Report AIDA-96-12, FG Intellektik, FB Informatik, TU Darmstadt, Germany, August 1996.

[SH97] T. Stützle and H.H. Hoos. The MAX-MIN Ant System and local search for the traveling salesman problem. In T. Bäck, Z. Michalewicz, and X. Yao, editors, *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pags 309-314. IEEE Press, Piscataway, NJ, 1997.

[SH00] T. Stützle and H.H. Hoos. MAX-MIN Ant System. *Future Generation Computer Systems*, 16(8):889-914, 2000.

[SK98] B. Sareni and L. Krahenbuhl. *Fitness sharing and niching methods revised*. IEEE Transactions on Evolutionary Computation, (2):97-106, 1998.

[Stü99a] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications*. DISKI - Dissertationen zur Künstliken Intelligenz. infix, 1999.

[Stü99b] T. Stützle. *Iterated Local Search for the quadratic assignment problem*. Technical Report aida-99-03, FG Intellektik, TU Darmstadt, 1999.

[Sys93] G. Syswerda. Simulated Crossover in Genetic Algorithms. In L.D. Whitley, editor, *Proc. of the second workshop on Foundations of Genetic Algorithms*, pags 239-255, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[Tai91] E. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443-455, 1991.

[VMOR99] S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors. *Meta-Heuristics - Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, 1999.

[WHP98] R.A. Watson, G.S. Hornby, and J.B. Pollack. Modeling building-block interdependency. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 1998. Stanford University Bookstore.

[YG93] X. Yin and N. Germany. *A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization*. En Artificial Neural Networks and Genetic Algorithms, Proceedings of the International

Conference in Innsbruck, R.F. Albrecht, C. Reeves, and N.C. Steele, editors, Berlin, Germany: Springer-Verlag, pags 450-457, 1993.