TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Arquitectura Orientada a Servicios para el Problema de la Selección de Requisitos

Curso 2015/2016

Alumno/a:

José Luis García Sánchez

Director/es:

José del Sagrado Martínez

INDICE

1. El problema de la selección de Requisitos

Introducción

El problema de la selección de requisitos se presenta dentro de la ingeniería de requisitos, la cual, según Thayer y Dorfman [ISEP02], "facilita el mecanismo apropiado para comprender lo que quiere el cliente, analizando necesidades, confirmando su viabilidad, negociando una solución razonable, especificando la solución sin ambigüedad, validando la especificación y gestionando los requisitos para que se transformen en un sistema operacional." Podemos identificar varias fases dentro de la misma:

- Identificación de requisitos: Se pregunta a los clientes y usuarios el objetivo del sistema, sin embargo, pueden aparecer problemas debido a problemas de comunicación y a la volatilidad de los requisitos.

- Análisis de requisitos: Una vez identificados, los requisitos se agrupan, se estudia cada uno en relación con los demás requisitos y se clasifican. Es corriente que los clientes y usuarios soliciten más requisitos de los que pueden realizarse en un tiempo y con unos recursos limitados, esto se soluciona negociando con el cliente. También se realiza una estimación del esfuerzo que conlleva cada requisito.

- Especificación de requisitos: Una especificación puede consistir en varias cosas, como un documento escrito, un modelo gráfico o un modelo matemático. Se Deben presentar los requisitos de una forma más consistente y comprensible.

- Validación de requisitos: Se evalúa la calidad de la especificación y se comprueba que todos los requisitos han sido establecidos correctamente. Para ello se realiza una revisión técnica formal.

Durante la fase de análisis o de validación podemos darnos cuenta de que no podemos satisfacer todos los requisitos que han propuesto los clientes. Para resolver esto se debe realizar una selección de requisitos, permitiéndonos contentar lo máximo posible a nuestros clientes con el esfuerzo limitado del que dispongamos. Por eso es fundamental haber ponderado los requisitos correctamente en las fases anteriores.

Formulación

Una vez explicado el contexto en el que encontramos el problema de la selección de requisitos podemos definirlo formalmente [MSSR12]:

Sea $R = \{r_1, r_2, ..., r_n\}$ el conjunto de requisitos propuestos por un conjunto de clientes $C = \{c_1, c_2, ..., c_n\}$. Los clientes tienen asignado un peso w_i que mide su importancia dentro del proyecto. Cada requisito $r_j \in R$ tienen asociado un esfuerzo de desarrollo que representa los recursos necesarios para incluir ese requisito en el producto software final, $E = \{e_1, e_2, ..., e_n\}$.

Cada $r_j \in R$ puede ser sugerido por más de un cliente y cada cliente c_i asigna un valor v_{ij} al requisito que representa la importancia que para él tiene la inclusión de ese requisito. Es decir, el valor representa la importancia del requisito r_j para el cliente c_i. Estos valores forman una matriz $m_x n$. La satisfacción global, s_j, o el valor añadido de la inclusión del requisito r_j en el producto software, se puede calcular como la suma ponderada de los valores de importancia para cada cliente, $s_j = \sum_i^m w_i \, v_{ij}$. Mayores valores de v_{ij}, implican una mayor prioridad de ese requisito para el cliente c_i. Un valor cero representa que ese cliente no toma en consideración ese requisito.

Por ejemplo, podemos definir la siguiente matriz de satisfacción en la que contrariamos con 5 clientes y 10 requisitos. Los números entre paréntesis de los requisitos reflejan el esfuerzo de los mismos y el de los clientes su peso en el proyecto, siendo 1 el valor más bajo. Cada valor de la matriz refleja el valor que el cliente x le asigna al requisito y.

	Req0 (3)	Req1 (3)	Req2 (5)	Req3 (2)	Req4 (1)	Req5 (5)	Req6 (4)	Req7 (2)	Req8 (4)	Req9 (1)
Cli0 (5)	1	3	3	5	4	3	4	5	5	2
Cli1 (4)	2	2	2	4	1	1	2	3	1	3
Cli2 (1)	1	4	2	3	4	5	1	1	5	1
Cli3 (2)	2	5	1	4	2	4	1	3	2	1
Cli4 (2)	5	3	4	3	5	5	3	1	4	4
Total:	28	43	35	58	42	42	37	46	46	33

En este caso la satisfacción total del requisito 0 sería 28 y lo obtendríamos multiplicando el peso de cada uno de los 5 clientes (5, 4, 1, 2 y 2) por el valor que le asignan cada uno de ellos al requisito 0 (1, 2, 1, 2 y 5): 5x1 + 4x2 + 1x1 + 2x2 + 2x5 = 28.

Estos requisitos pueden estar interrelacionados o presentan dependencias, que hacen que tengan que abordase unos antes que otros o que sean excluyentes, las dependencias se pueden clasificar como dependencias funcionales que representan relaciones semánticas entre varios requisitos y dependencias por recursos consumidos, donde la presencia o no de varios requisitos puede modificar su esfuerzo o satisfacción. Las dependencias funcionales pueden ser definidas de la siguiente forma:

- Implicación (r_i implica r_j). El requisito r_i no puede ser seleccionado si el requisito r_j no ha sido implementado previamente.
- Combinación (r_i acoplado con r_j). El requisito r_i no puede ser incluido separado de r_j.
- Exclusión (r_i excluye a r_j). El requisito r_i no puede incluirse junto al requisito r_j.

De esta forma el objetivo del problema será encontrar el subconjunto de requisitos Ŕ de entre todos los subconjuntos de R, $\check{R} \in \wp(R)$, que sea la mejor solución. La calidad de la solución se mide mediante una función de evaluación, intentando maximizar la satisfacción (función mono-objetivo) sin sobrepasar las limitaciones de esfuerzo máximo B:

$$sat(\check{R}) = \sum_{j \in \check{R}} (Sj) \qquad eff(\check{R}) = \sum_{j \in \check{R}} (ej)$$

Maximizar sat(Ŕ)

eff(Ŕ) ≤ B

Ejemplo de un problema

Una vez definido el problema, podemos verlo más en detalle con un pequeño ejemplo, utilizando la matriz mostrada anteriormente y añadiéndole algunas dependencias:

	Req0 (3)	Req1 (3)	Req2 (5)	Req3 (2)	Req4 (1)	Req5 (5)	Req6 (4)	Req7 (2)	Req8 (4)	Req9 (1)
Cli0 (5)	1	3	3	5	4	3	4	5	5	2
Cli1 (4)	2	2	2	4	1	1	2	3	1	3
Cli2 (1)	1	4	2	3	4	5	1	1	5	1
Cli3 (2)	2	5	1	4	2	4	1	3	2	1
Cli4 (2)	5	3	4	3	5	5	3	1	4	4
Total:	28	43	35	58	42	42	37	46	46	33

6 → 2, 7 → 3, 7 → 4, 8 → 6, 9 → 0, 1 X 5, 7 ↔ 0

*Representando → la implicación, X la exclusión y ↔ la combinación.

Para encontrar una solución para este problema primero debemos indicar cuál es el esfuerzo máximo que disponemos, es decir, cuántos recursos vamos a poder emplear. La suma de los esfuerzos de los requisitos que incluyamos en nuestra solución no podrá nunca superar este esfuerzo máximo. Si, por ejemplo dispusiéramos de un esfuerzo máximo de 12:

- La solución Req2 + Req6. Con un esfuerzo de 9 y una satisfacción de 72 sería una solución válida pero incompleta ya que aún podríamos añadir más requisitos a la solución.

- La solución Req1 + Req2 + Req6. Con un esfuerzo de 12 y una satisfacción de 115 sería una solución válida y completa ya que no podemos añadir ningún requisito más a la solución porque ya hemos alcanzado el esfuerzo máximo. Sin embargo, para que una solución sea completa no es necesario que se alcance el esfuerzo máximo sino, que no se pueda añadir ningún requisito más. La solución Req2 + Req4 + Req6. Con un esfuerzo de 11 y una satisfacción de 114 es también completa. Ya que ningún requisito puede formar parte de la solución a pesar de que aún queda esfuerzo disponible.

- La solución Req1 + Req3 + Req6. Con un esfuerzo de 9 y una satisfacción de 138 no es una solución válida ya que no se cumplen las dependencias, al aparecer el req6 sin estar el req2.

Es indicativo que incluso en un problema tan pequeño como este, es difícil decidir a simple vista cuál es la solución óptima debido al elevado número de combinaciones posibles.

Características de los problemas generados

Debido a la gran cantidad de conjuntos de prueba que pueden llegar a generarse para comprobar el correcto funcionamiento de los algoritmos y cómo se comportan cuando aumenta el tamaño del problema o las dependencias, se ha visto conveniente el desarrollo de una pequeña aplicación que facilite la creación de los mismos.

Todos los conjuntos de pruebas generados por esta aplicación cumplen las siguientes características:

- Empiezan por el requisito 0 y acaban por el requisito n-1, siendo n el número de requisitos.

- Empiezan por el cliente 0 y acaban por el cliente n-1, siendo n el número de clientes.

- El esfuerzo de los requisitos, el peso de los clientes y el valor que los clientes le asignan a cada requisito, es un número aleatorio comprendido entre un máximo y un mínimo.

- Cada cliente le asigna un valor a todos los requisitos.

- Para las dependencias se aplican una serie de restricciones para que los conjuntos de pruebas sean más realistas, están se aplican en el orden en el que vienen indicadas:
  - Un requisito no puede tener ninguna dependencia consigo mismo.
  - Dos requisitos acoplados no podrán tener ninguna otra dependencia entre ellos.
  - Un requisito no podrá ser prerrequisito de sus prerrequisitos directos o indirectos.
  - Un requisito y sus prerrequisitos, directos o indirectos, no podrán ser excluyentes.
  - Se evita toda información redundante, impidiendo que se repitan dependencias.

- Existe un número máximo de dependencias, para poder alcanzar el máximo en una dependencia, las otras deben ser 0, teniendo prioridad la combinación, después la implicación y finalmente la exclusión. Teniendo la primera un máximo de n-1 y las otras dos un máximo de n*(n-1) / 2 (el número de aristas en un grafo).

Todas estas restricciones son exclusivas del generador de conjuntos de prueba, por lo tanto, el único requerimiento para que un problema pueda ser leído correctamente es que el archivo de texto siga la estructura mencionada anteriormente. Pudiendo por tanto, trabajar con ficheros de entrada desordenados o incompletos.

Aunque las siguientes dependencias no se encontrarán en ningún caso real, ni tampoco en ningún caso generado de forma aleatoria por nuestra aplicación, es importante dejar claro cómo reaccionará el programa al encontrarse con cada una de ellas, para poder garantizar la robustez de la aplicación.

- Dependencias con requisitos inexistentes.
  Todas ellas serán ignoradas.

- Dependencias de un requisito consigo mismo.
  La implicación y la combinación serán ignoradas.
  La exclusión provocará que el requisito no pueda ser seleccionado nunca.

- Dependencias entre dos requisitos acoplados.
  Se tratarán como si fueran dependencias de un requisito consigo mismo.

- Ciclos dentro de la implicación.
  Ningún requisito dentro del ciclo podrá ser seleccionado.

- Implicación y exclusión.
  Si un requisito necesita y excluye a otro requisito, no podrá ser seleccionado.

- Repetición y redundancia.
  La información repetida de forma literal será ignorada.
  La información redundante será considerada, pudiendo agilizar el proceso.

2. Algoritmos propuestos

La ingeniería del software basada en búsqueda consiste en la aplicación de técnicas de optimización de problemas al ámbito de la ingeniería del software. Un problema de optimización considera una o más variables de decisión cuyos valores definen el espacio de búsqueda. Existen varios problemas de optimización en la ingeniería del software como, por ejemplo, el problema de la selección de requisitos, el problema de las pruebas de software o el problema de la planificación de un proyecto software. En el problema que nos ocupa, el de la selección de requisitos en su versión monoobjetivo, se han aplicado en el pasado varias técnicas metaheurísticas como GRASP (Greedy Randomized Adaptative Search Procedure), ascenso de colinas, recocido simulado, algoritmos genéticos o colonias de hormigas. También se ha tratado su versión multiobjetivo utilizando NSGA-II (Non-dominated Sorting Genetic Algorithm II) y MoCell (MultiObjective Cellular genetic algorithm). Para la realización de este trabajo se han implementado tres técnicas metaheurísticas: grasp, un algoritmo genético y un sistema de colonias de hormigas. Así como un algoritmo de búsqueda en profundidad para comprobar la precisión de las técnicas en aquellos problemas donde pueda ejecutarse en un tiempo razonable.

Complementariamente a estas técnicas, se realizará un preprocesamiento de la entrada, un postprocesamiento de la solución y una gestión de las dependencias.

Técnicas de búsqueda

Las técnicas de búsqueda consideran los sucesivos estados en los que puede encontrarse una instancia del problema. Mediante un estado inicial y una función sucesor, que sirve para movernos de un estado al siguiente podemos generar un árbol de búsqueda que represente al espacio de estados. En lugar de un árbol de búsqueda se puede hablar de un grafo cuando el mismo estado puede alcanzarse desde varios caminos. El proceso de búsqueda comienza en un nodo de búsqueda que corresponde al estado inicial, el cual se expande utilizando la función sucesor hasta alcanzar el nodo objetivo.

En nuestro problema de la selección de requisitos, se puede definir el espacio de estados como todas las combinaciones de requisitos válidas, es decir cuyo esfuerzo no supere el esfuerzo máximo y que cumplan todas las dependencias. La función sucesor consiste en añadir un nuevo requisito a una solución que aún no este complete y, al tratarse de un problema de optimización, desconocemos el nodo objetivo, por lo que deberemos continuar expandiendo el espacio de estados hasta haberlo explorado en su totalidad, o hasta que hayamos realizado un número máximo de iteraciones.

Para poder representar de forma visual el árbol de búsqueda de nuestro problema este debe ser muy pequeño pues:

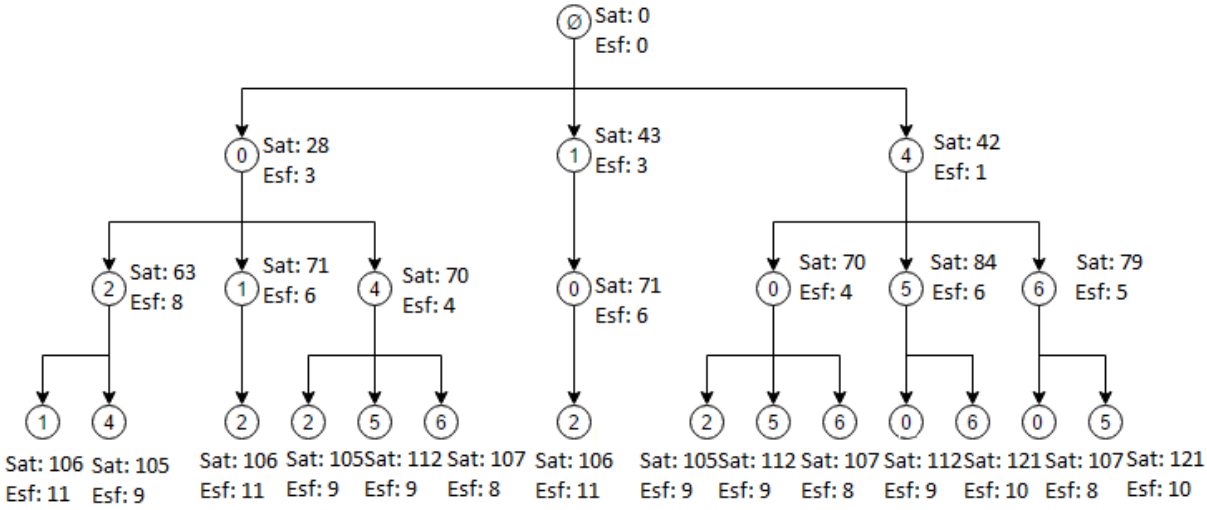
$$número\ de\ nodos\ hoja = \ número\ de\ requisitos^{profundidad\ del\ árbol}$$

Por lo tanto vamos a cortar nuestro problema y quedarnos sólo con 7 requisitos y vamos a generar nuevas dependencias:

	Req0 (3)	Req1 (3)	Req2 (5)	Req3 (2)	Req4 (1)	Req5 (5)	Req6 (4)
Cli0 (5)	1	3	3	5	4	3	4
Cli1 (4)	2	2	2	4	1	1	2
Cli2 (1)	1	4	2	3	4	5	1
Cli3 (2)	2	5	1	4	2	4	1
Cli4 (2)	5	3	4	3	5	5	3
Total:	28	43	35	58	42	42	37

1 X 4, 2 → 0, 3 → 1, 3 → 2, 5 → 4, 6 → 4

El árbol de búsqueda para este problema con un esfuerzo máximo de 12 es el siguiente:



Como podemos observas, el requisito 3 no puede llegar nunca a formar parte de la solución ya que se alcanza el esfuerzo máximo introduciendo sus prerrequisitos. También podemos observar que podemos llegar a la misma solución por varios caminos. La solución óptima para este problema es la formada por los requisitos 4, 5 y 6 ya que es la que aporta una mayor satisfacción (121) sin sobrepasar el límite de esfuerzo y cumpliendo con todas las dependencias.

A la hora de aplicar estas técnicas es importante comprobar si estás son completas, si existe una solución la encontrarán; y óptimas, si encuentran una solución podemos garantizar que es la mejor solución. Así como su complejidad en tiempo y espacio.

En nuestro problema, no tiene sentido plantearse si la búsqueda es completa, debido a que cualquier salida, incluso el conjunto vacío, puede considerarse una solución válida. Por tanto, debemos considerar si la solución es óptima y en caso de no serlo, como de cercana a la óptima es. También debido a la gran cantidad de memoria que poseen los ordenadores actuales, la complejidad en tiempo es más significativa que en espacio.

Búsqueda en Profundidad

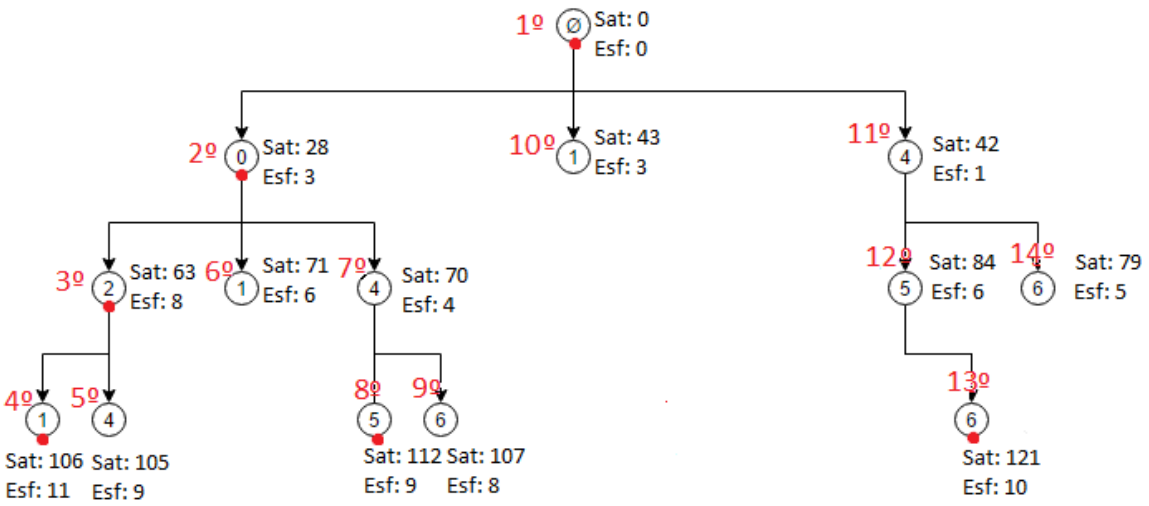
La búsqueda en profundidad, pertenece a las estrategias de búsqueda a ciegas. Las cuáles exploran el espacio de búsqueda, sin tener información adicional acerca de los estados.

Mediante ese proceso se genera un único sucesor para cada nodo hasta encontrar el nodo objetivo. Para evitar que el proceso de búsqueda continúe sin descanso se estable una profundidad límite. Una vez alcanzada esta profundidad límite sin alcanzar su objetivo se realiza un retorno a un nodo inmediatamente superior que aún admita nuevas expansiones. Esta búsqueda sólo necesita almacenar la parte del árbol de búsqueda que aún no haya sido totalmente expandido, creciendo el coste de memoria de forma lineal con la profundidad límite. Las dos grandes desventajas de este método son que no se puede garantizar que la solución encontrada sea la menos profunda y que probablemente se tendrá que explorar una gran parte del grafo de estados.

En este caso concreto, al no disponer de un nodo objetivo se explorará el árbol de búsqueda al completo. Al no admitir repetición de requisitos, la profundidad límite será el esfuerzo máximo y en el peor de los casos el número total de requisitos.

Realizándose de esta forma, podemos garantizar que esta búsqueda es completa y óptima, sin embargo, la complejidad temporal es demasiado elevada, lo que nos invita a explorar algunas soluciones heurísticas, que encuentren una solución potencialmente válida en tiempos más bajos.

Aquí se muestra un ejemplo de cómo se recorrería el árbol de búsqueda, están marcados con un punto rojo los estados que son el mejor en el momento en el que son explorados.



Como podemos apreciar, se exploran todas las combinaciones, sin embargo no se exploran las combinaciones repetidas, debido a que si ya hemos llegado a una combinación por otro camino este se omite. Esto se debe a que para cada nodo tenemos una pila con los requisitos que aún quedan por explorar.

Estrategias de búsqueda heurísticas

Las búsquedas no heurísticas como la búsqueda en profundidad que hemos visto antes, son también conocidas como búsquedas a ciegas debido a que el único conocimiento que tienen es si han alcanzado el estado objetivo o no. Sin embargo las búsquedas heurísticas tienen conocimiento específico del problema lo que les permite valorar como de buenos son varios estados e intentar elegir el mejor, esto provoca que puedan encontrar antes el estado objetivo. Sin embargo esta valoración puede ser inexacta por lo que es posible que se cometan errores.

En la realización de este trabajo se han explorado tres técnicas de búsqueda heurísticas:

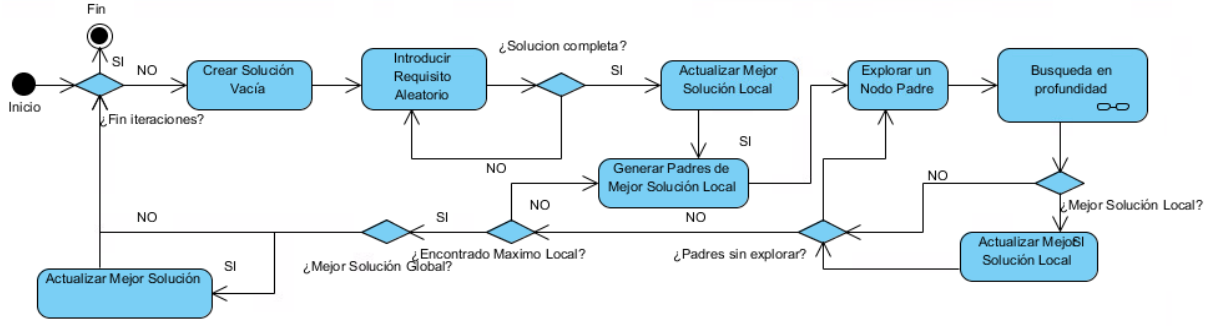
GRASP

El proceso de búsqueda voraz aleatorio adaptativo, GRASP por sus siglas en inglés (Greedy randomized adaptive search procedure), consiste en la aplicación iterativa de una búsqueda local para encontrar al mejor vecino partiendo de una solución aleatoria. Este proceso es aleatorio porque utiliza una técnica voraz de búsqueda local, llamada ascenso de colinas para encontrar al mejor vecino, aleatorio porque el estado inicial es aleatorio y adaptativo porque al comenzar desde diversos estados se pueden superar varios máximos locales para tratar de alcanzar el máximo global.

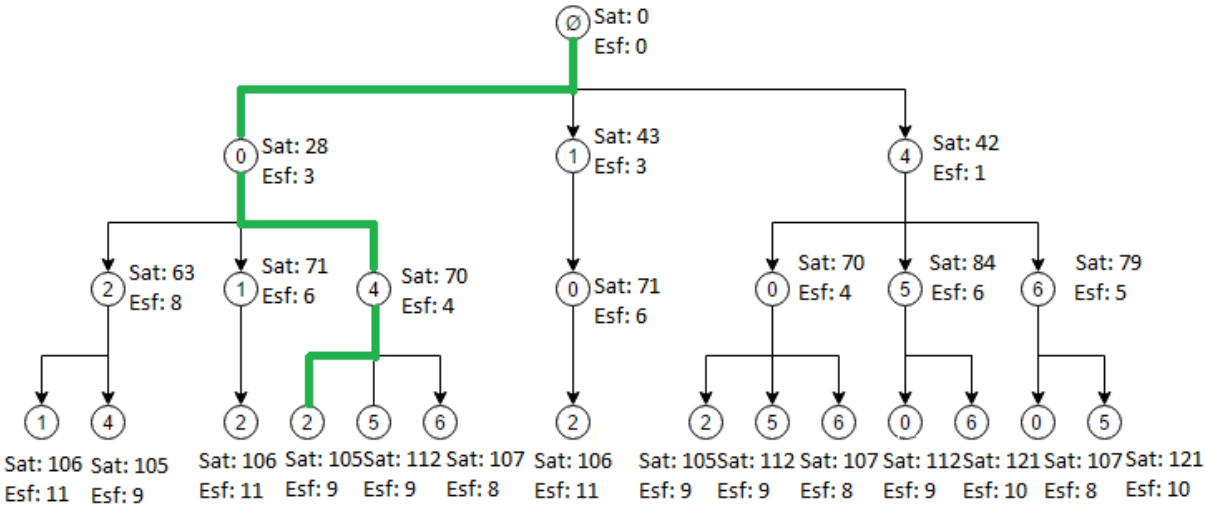
La búsqueda de ascensión de colinas consiste en moverse continuamente hacia un valor creciente, mirando únicamente a los vecinos inmediatos. Esta estrategia es voraz, ya que selecciona el mejor vecino basándose únicamente en su valor actual.

La única variable que podemos ajustar en este algoritmo es el número de iteraciones que queremos realizar. Aumentando de forma lineal la probabilidad que encontrar la solución óptima o al menos una solución cercana a la óptima y el tiempo de ejecución del programa. Mientras que el costo de memoria se mantendrá constante ya que las iteraciones anteriores no se mantendrán en memoria.

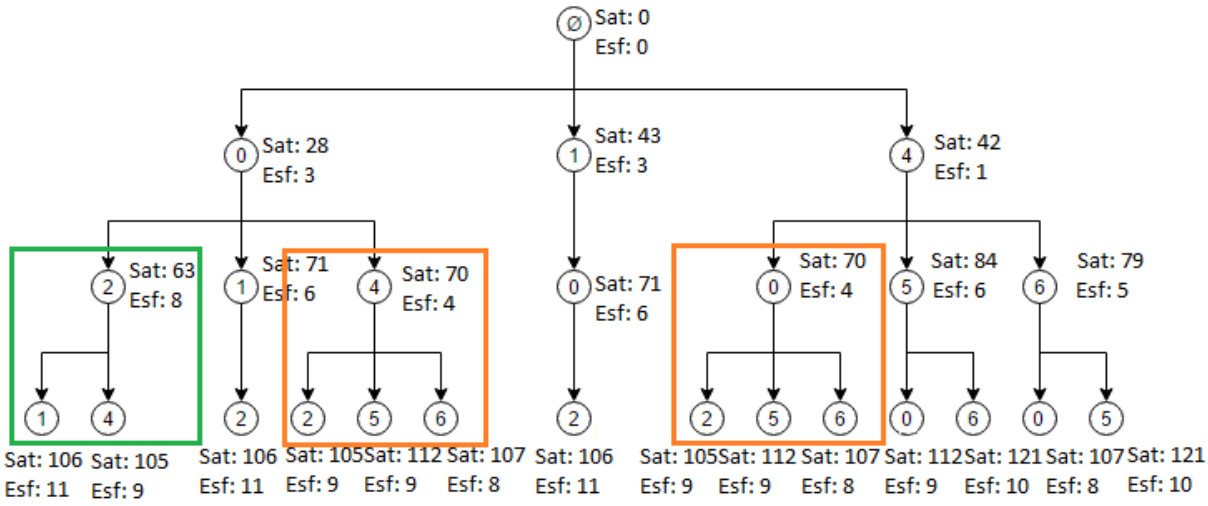
En nuestro caso es importante definir el concepto de vecindad que hemos elegido ya que este no es intrínseco al problema. Se ha decidido considerar vecino de una solución cualquier solución que sea descendiente de uno de los predecesores de la solución actual. Descendiente que será obtenido utilizando la búsqueda en profundidad partiendo de los predecesores en lugar de una solución vacía lo cual acota enormemente el espacio de búsqueda.



Volviendo a nuestro ejemplo anterior, comenzaríamos con una solución vacía y se seleccionaría aleatoriamente uno de los siguientes requisitos: Req0, Req1 y Req4. Supongamos se ha seleccionado el Req0, ahora los candidatos a formar parte de la solución son los requisitos: Req1, Req2 y Req4. Supongamos entonces que se selecciona el Req4 y posteriormente el Req2, nuestra solución ya sería completa por lo que detenemos la generación aleatoria. Esta solución es la mejor solución local que disponemos ya que es la única que hemos explorado.

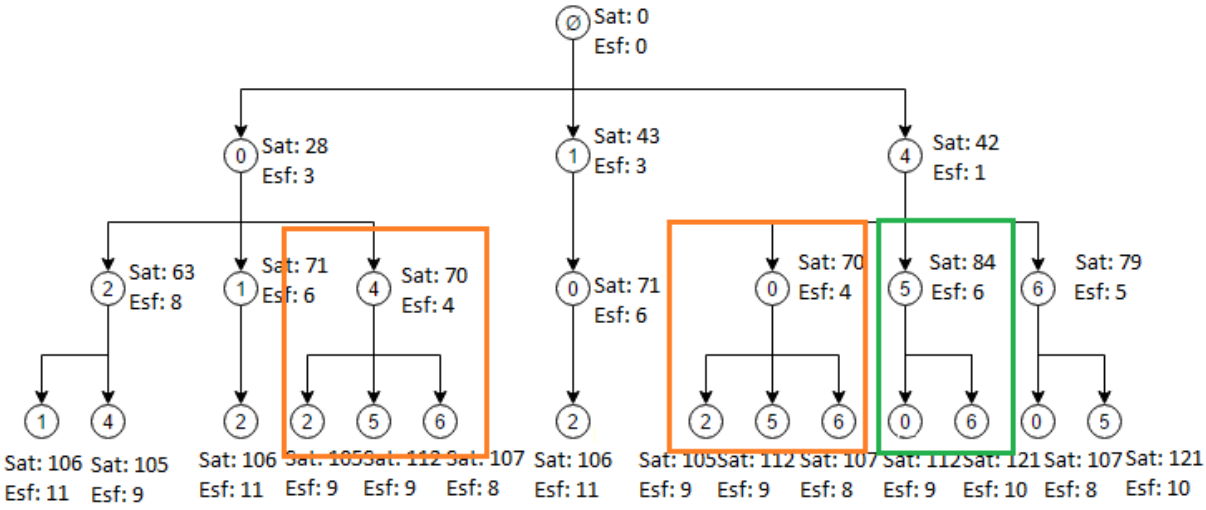


Ahora deberemos explorar todos los vecinos de nuestra solución. Como ya hemos definido antes, los vecinos son los descendientes de nuestra solución actual menos un requisito. Es decir los descendientes de Req0, Req2 de y de Req0, Req4 puesto que Req2, Req4 no es una solución válida puesto que Req2 no puede aparecer sin Req0.



Debemos explorar los descendientes de Req0, Req2 (Cuadro Verde) y Req0, Req4 (Cuadros naranjas). Aunque en la representación aparecen dos subárboles a explorar para el nodo Req0, Req4, al no importar el orden ambas ramas son la misma y la exploración de sus descendientes sólo se realiza una vez. Sin embargo si debemos explorar dos veces el estado Req0 Req2 Req4 ya que lo exploramos desde dos búsquedas locales diferentes y etas no se comunican entre sí.

Nos quedaremos con la mejor solución hasta el momento, la Req0, Req4 y Req5 y repetiremos el proceso.



Esta vez es el Req4 el que no podemos omitir por su dependencia con Req5.Tras realizar de nuevo la exploración de los vecinos obtenemos la solución formada por los requisitos Req4, Req5 y Req6. Como la solución ha cambiado habrá que volver a explorar los vecinos. Esta vez sin embargo sabemos que la solución no va a mejorar pues ya hemos visto anteriormente que la solución actual es la óptima. Al no mejorar podemos dar por finalizada la iteración y probar a partir desde un punto diferente.

Algoritmo de la colonia de hormigas

Este algoritmo intenta emular el comportamiento que tienen las hormigas cuando se mueven dentro de un hormiguero, las hormigas al ir caminando por los túneles van dejando tras de sí una feromona que se disipa con el tiempo y cuando llegan a un cruce son más propensas a ir por el camino que tenga más feromonas. De esta forma cuantas más hormigas pasen por un camino más probable será que futuras hormigas lo tomen también.

Este proceso es perfectamente extrapolable al ámbito de la búsqueda. Las hormigas comienzan en un estado inicial y se dirigen hacia un estado final. Por el camino se encuentran con cruces y deben decidir, teniendo cada camino una probabilidad de ser elegido en función de su deseabilidad, que viene determinada por el número de feromonas, el esfuerzo y la satisfacción:

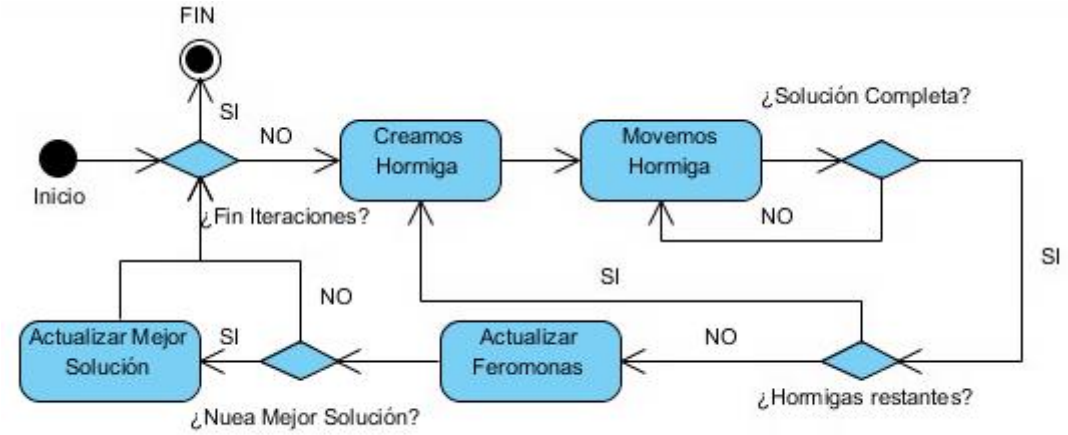
$$deseabilidad(i) = nFeroromas(i)^{\alpha} \cdot \frac{1}{eff(i)}^{\beta} \cdot (1 - \frac{1}{sat(n)})^{\gamma}$$

$$p(n) = deseabilidad(n)/deseabilidad(total)$$

‘α’, ‘β’ y ‘γ’ son constantes que se pueden asignar para darle más importancia o menos importancia a cada factor. Todos los requisitos parten con la misma cantidad de feromonas iniciales, cantidad que viene definida por una constante y varia siguiendo la siguiente fórmula:

$$nFeromonas(i) = nFeromonas(i) * (1 - coefEvaporacion) + nHormigas(i) * nFerAdicional$$

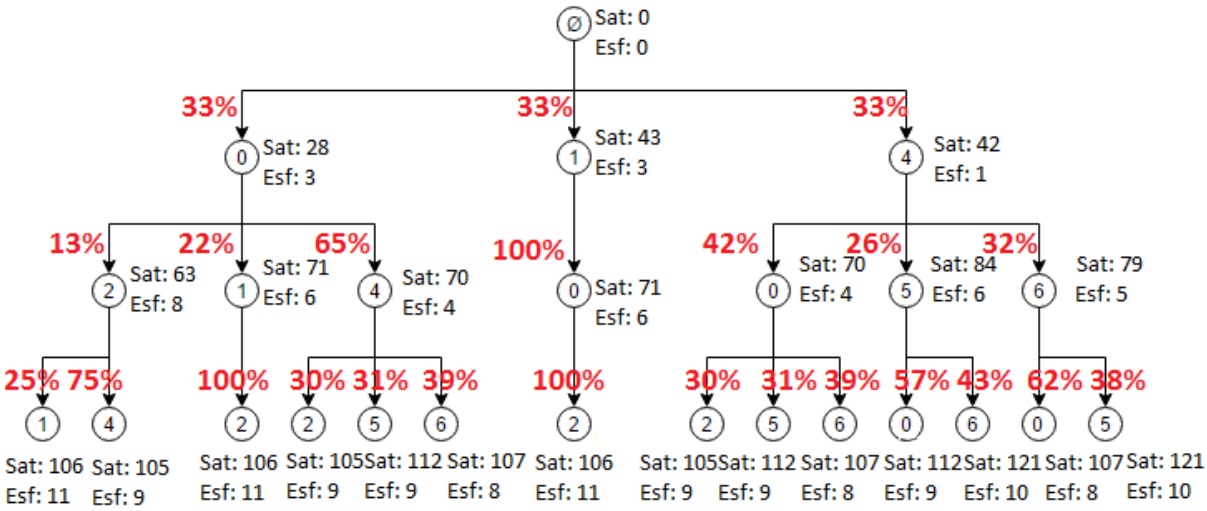
Siendo el coeficiente de evaporación una constante que debe tener unos valores entre 0 y 1 y que indica como de rápido pierden las feromonas los requisitos, el número de hormigas indica cuantas hormigas han pasado por un requisito en la iteración actual y número de feromonas adicionales indica cuanta feromona deposita cada hormiga.



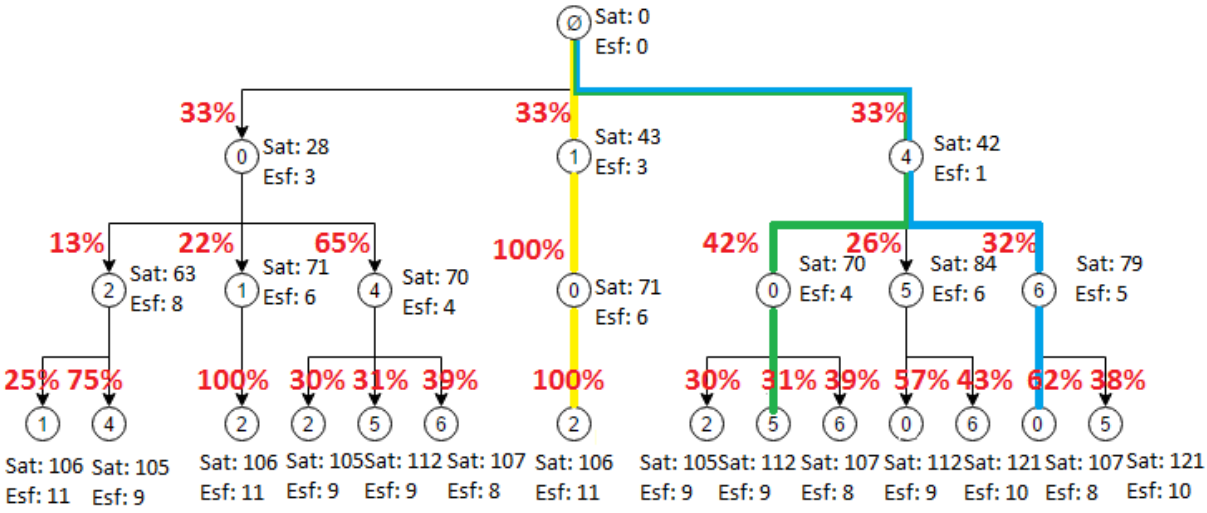
Vamos a intentar resolver nuestro problema utilizando una colonia de hormiga, para ello primero debemos seleccionar los valores que le vamos a dar a las constantes. Para simplificar el cálculo podemos fijar: de $\alpha = \beta = \gamma = 1$. Cada requisito va a partir con 5 unidades de feromona, cada hormiga va a dejar 1 unidad de feromona al pasar por un requisito y el coeficiente de evaporación va a ser del 25%. Además se van a generar 3 hormigas en cada iteración. Con estos datos podemos generar la tabla de deseabilidad inicial, es decir antes de la primera iteración:

Requisito	Feromonas	Esfuerzo	Satisfacción	Deseabilidad
Req0	5	3	28	1,60714286
Req1	5	3	43	1,62790698
Req2	5	5	35	0,97142857
Req3	5	2	58	2,45689655
Req4	5	1	42	4,88095238
Req5	5	5	42	0,97619048
Req6	5	4	37	1,21621622

Utilizando esta deseabilidad para calcular la probabilidad de que la hormiga tome cada túnel obtenemos el siguiente árbol de probabilidad:



Como podemos observar, el primer cruce no se corresponde con la deseabilidad de los requisitos ya que el punto de partida de la hormiga es completamente aleatorio y por tanto todos tienen la misma probabilidad. Procedemos a generar 3 hormigas y ver que camino toman:



Tras este paso, vemos que la mejor solución visitada hasta el momento es la formada por los requisitos: Req4, Req0 y Req5 con una satisfacción de 112. Y debemos contar cuantas hormigas han pasado por cada requisito para actualizar la tabla de feromonas.

Requisito	Nº Hormigas	Feromonas
Req0	3	6,75
Req1	1	4,75
Req2	1	4,75
Req3	0	3,75
Req4	2	5,75
Req5	1	4,75
Req6	1	4,75

Con estas nuevas tablas podemos calcular la deseabilidad para una nueva iteración:

Requisito	Feromonas	Esfuerzo	Satisfacción	Deseabilidad
Req0	6,75	3	28	2,16964286
Req1	4,75	3	43	1,54651163
Req2	4,75	5	35	0,92285714
Req3	3,75	2	58	1,84267241
Req4	5,75	1	42	5,61309524
Req5	4,75	5	42	0,92738095
Req6	4,75	4	37	1,15540541

Y vuelta a empezar durante el número de iteraciones que hayamos indicado.

Algoritmos genéticos

Los algoritmos genéticos intentan emular la reproducción sexual generando la descendencia a partir de la combinación de dos estados padres. Por regla general siguen el siguiente esquema: se genera una población inicial, se aplica una función de evaluación y se aplica de forma iterativa hasta alcanzar una condición de término las fases de selección, cruce, mutación y reemplazo.

En nuestro caso la población inicial es generada de forma aleatoria. La función de evaluación se corresponde con la satisfacción de cada solución y la condición de término es alcanzar un número máximo de iteraciones.

Generalmente la representación de las soluciones que se suele utilizar en los algoritmos genéticos es una lista de 0 y 1 en las que el 0 representa la ausencia de un elemento y el 1 su pertenencia, sin embargo para seguir utilizando la misma estructura que se ha utilizado en los demás métodos, en esta implementación se ha utilizado una lista con los elementos que forman parte de la solución. Por ejemplo la solución formada por el Req0, el Req4 y el Req6 de un total de 10 requisitos se mostraría de la siguiente forma:

-Representación tradicional: {1000101000}

-Representación utilizada:{0, 4, 6}

A continuación se definen los cuatro operadores básicos del algoritmo genético: la selección, el cruce, la mutación y el reemplazo:

La selección consiste en seleccionar los individuos que van a transmitir sus genes a la siguiente generación. Al igual que en la reproducción animal, los mejores tienen más probabilidad de poder reproducirse. Teniendo una solución i la siguiente probabilidad de ser elegido

$$p(i) = \frac{Sat(i)}{\sum_{j \in \check{R}} Sat(j)}$$

En cada iteración se seleccionan varios individuos para reproducirse por pares, pudiendo un individuo ser seleccionado varias veces y otro ninguna. Un individuo no puede reproducirse consigo mismo.

El cruce entre dos soluciones genera nuevas soluciones a partir de las dos que tenemos como progenitores, para nuestro problema al existir dependencias puede que no todos los genes puedan ser transmitidos a los descendientes. Sin embargo la representación especial que tenemos nos aporta una gran ventaja ya que mantenemos el orden de las dependencias dentro de la solución.

Por ejemplo supongamos que vamos a cruzar la solución formada por los requisitos Req0, el Req4 y el Req6 con otra formada por los requisitos Req1, Req4 y Req5 con un punto de cruce en el medio y que el Req0 tiene como prerrequisito al Req6:

{10001|01000} → {10001|10000}

{01001|10000} → {01001|01000}

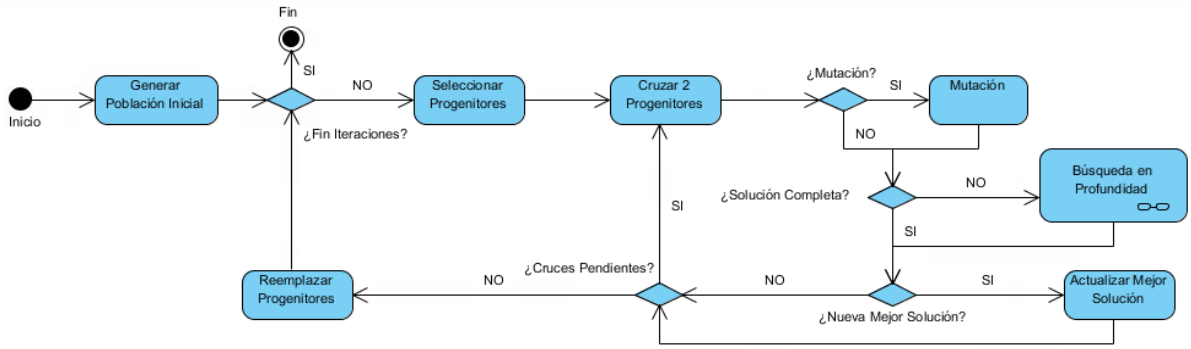
Como podemos apreciar, la solución Req0, Req4, Req5 no es válida ya que aparece el Req0 sin Req6. Sin embargo primero aparece el Req0 por lo que tendremos que hacer las comprobaciones posteriormente.

Sin embargo al usar nuestra notación mantenemos el orden en el que son introducidos los requisitos a la solución por tanto nunca tendríamos {0, 4, 6}. Podríamos tener {4, 6, 0}, {6, 0, 4} o {6, 4, 0} pero 6 siempre debe aparecer primero, por tanto cuando realicemos el cruce podremos comprobar cada dependencia con forme la añadimos a la descendencia y si no se cumple en ese momento no se cumplirá nunca, por lo que este elemento quedará fuera de la solución final. Es más, gracias a este método podemos garantizar que los requisitos que queden en el lado izquierdo siempre serán válidos ya que no pueden tener una dependencia sin cumplir por lo que sólo habrá que comprobar las dependencias en el lado derecho. Además de las dependencias también un requisito puede quedarse fuera si se excede el esfuerzo máximo, para intentar evitar esto dentro de lo posible, se ha decidido que las soluciones se partan por la mitad del esfuerzo y no por la mitad del número de elementos, aunque no siempre es posible crear dos mitades iguales, especialmente cuando las soluciones tienen pocos elementos.

La mutación o aparición aleatoria de nuevos genes se utiliza para explorar nuevos requisitos que no aparecieran en la población inicial. En la representación tradicional se limita a cambiar un 0 por un 1 o viceversa, sin embargo en nuestro caso habrá que cambiar un requisito por otro, y habrá que hacerlo por uno que cumpla todas las dependencias y no exceda el esfuerzo, por ello cuando se ejecute la mutación, se ejecuta aleatoriamente después del cruce habrá que escoger un requisito al azar y reemplazarlo por otro elegido nuevamente al azar de entre los disponibles. Si el requisito escogido al azar no puede ser eliminado ya que crearía un conflicto con las dependencias la mutación no tiene efecto, de hecho el requisito nunca llega a ser eliminado de forma que el orden dentro de la lista se mantiene. También puede darse el caso de que un requisito eliminado por la mutación vuelva a ser seleccionado en la propia mutación, es una posibilidad que no se debe eliminar ya que en algunos casos este puede ser el único requisito disponible.

Debido a los requisitos que son excluidos de una solución en el cruce por incompatibilidades o que al mutar un requisito de gran esfuerzo es sustituido por uno de menor, puede que nuestros descendientes sean soluciones incompletas, para solucionar esto, se emplea un operador de reparación que consiste en realizar una búsqueda en profundidad a partir de la solución parcial disponible.

El reemplazo consiste simplemente en sustituir a los progenitores por los descendientes antes de comenzar una nueva iteración.



Para este algoritmo, como no vamos a utilizar una representación gráfica del espacio de estados podemos utilizar un problema más grande, en este caso vamos a utilizar el siguiente problema de 20 requisitos:

	req0 (1)	req1 (4)	req2 (2)	req3 (3)	req4 (4)	req5 (2)	req6 (1)	req7 (4)	req8 (4)	req9 (4)
Cli0 (3)	3	1	1	3	3	2	3	3	1	2
Cli1 (3)	1	2	4	1	2	4	3	1	3	3
Cli2 (3)	4	3	2	4	4	4	4	3	4	2
Cli3 (1)	2	4	2	4	1	1	4	3	4	3
Cli4 (2)	1	3	1	2	2	4	1	2	3	1
Total:	28	28	25	32	32	39	36	28	34	26
	req10 (4)	req11 (4)	req12 (4)	req13 (3)	req14 (4)	req15 (1)	req16 (3)	req17 (1)	req18 (3)	req19 (4)
Cli0 (3)	3	1	4	4	4	4	4	4	4	2
Cli1 (3)	1	2	4	3	3	2	2	1	3	4
Cli2 (3)	1	2	2	1	1	1	3	4	3	3
Cli3 (1)	1	3	1	3	2	1	3	1	4	2
Cli4 (2)	4	2	1	2	1	3	1	1	1	3
Total:	24	22	33	31	28	28	32	30	36	35

2→4, 4→3, 4→11, 9→5, 9→11, 9→13, 11→5, 11→10, 12→8, 19→12 8x18 14x3

En primer lugar el algoritmo generará tantos individuos como se le haya indicado, en este caso 4:

Individuo 1: {6, 14, 13, 17, 1, 5} E: 15, S: 192

Individuo 2: {5, 6, 8, 16, 10, 15} E: 15, S: 193

Individuo 3: {0, 10, 16, 15, 17, 14, 6} E: 15, S: 206

Individuo 4: {7, 1,18, 6, 0, 17, 15} E: 15, S: 214

Ahora debemos seleccionar los cruces posibles, para ello cada individuo tendrá una probabilidad de ser elegido en función de su satisfacción. En este caso:

Individuo 1: 23.85%

Individuo 2: 23.97%

Individuo 3: 25.59%

Individuo 4: 26.58%

En este caso el primero en ser elegido es el individuo 2, ahora hay que seleccionar otro individuo para que se cruce con el individuo 2. Para ello hay nuevas probabilidades porque no podemos repetir individuo.

Individuo 1: 31.37%

Individuo 3: 33.66%

Individuo 4: 34.97%

Esta vez ha sido elegido el individuo 3. Se vuelve a repetir el proceso nuevamente con todos los individuos y son elegidos el 1 nuevamente con el 4 como pareja.

Ahora habrá que trocear cada individuo en 2 trozos intentando dejar la mitad del esfuerzo en cada lado, para ello incluiremos requisitos en el primer trozo hasta alcanzar la mitad del esfuerzo total:

Individuo 2: {5, 6, 8, | 16, 10, 15} E: 15, S: 193

Individuo 3: {0, 10, 16, | 15, 17, 14, 6} E: 15, S: 206

Individuo 4: {7, 1, | 18, 6, 0, 17, 15} E: 15, S: 214

Como podemos observar la división por esfuerzo deja menos elementos al lado izquierdo, esto se debe a como hemos generado la población inicial. Al principio todos los requisitos tenían la misma probabilidad de aparecer, pero con forme se iba llenando la solución solo los más pequeños podían formar parte de ella.

En primer lugar realizaremos el cruce entre los individuos 2 y 3:

Individuo 2: {5, 6, 8, | 16, 10, 15} → Hijo1: {5, 6, 8, | 15, 17, 14, 6}

Individuo 3: {0, 10, 16, | 15, 17, 14, 6} → Hijo2: {0, 10, 16, | 16, 10, 15}

Como podemos observar hay requisitos repetidos en nuestra descendencia, estos por tanto no son válidos y no se incluyen, dejando una solución incompleta.

Procedemos ahora a realizar la fase de mutación, esta vez en ambos casos ha sucedido:

Hijo1: {0, 10, 16, 15} → Hijo1: {0, 10, 15, 14}

Hijo2: {5, 6, 8, 15, 17, 14} → Hijo2: {5, 6, 8, 15, 14, 17}

Como podemos apreciar en el Hijo1 el requisito 16 ha sido sustituido por el requisito 14, y algo importante el requisito 14 aparece en la última posición de la lista. Esto es importante para mantener la integridad, ya que el requisito 16 podría haber mutado a un requisito que necesite la presencia del 15. Además nos ayuda a entender la curiosa mutación que ha sufrido el hijo2, el requisito 17 ha mutado a otro requisito 17, por lo que este ahora ocupa la última posición.

Para completar a nuestros dos descendientes, realizamos una búsqueda en anchura dejando como resultado final:

Hijo1: {0,10, 15, 14, 17, 6, 5} E: 14 S: 213

Hijo2: {5, 6, 8, 15, 14, 17, 0} E: 14 S: 223

Ahora realizamos el mismo proceso para el otro cruce:

Individuo 2: {5, 6, 8, | 16, 10, 15} → Hijo3: {5, 6, 8, | 18, 6, 0, 17, 15}

Individuo 4: {7, 1, | 18, 6, 0, 17, 15} → Hijo4: {7, 1, | 16, 10, 15}

Nuevamente nos encontramos con el requisito 6 repetido así que lo omitimos, además el requisito 18 no puede formar parte de la solución ya que en esta ya está presente el 8 y son excluyentes, finalmente el requisito 15 también se queda fuera ya que la solución superaría el esfuerzo límite si lo incluyera.

En esta ocasión sólo el hijo3 ha sufrido una mutación:

Hijo3: {5, 6, 8, 0, 17, 15} → Hijo3: {5, 6, 8, 17, 15, 10}

Además como el hijo 4 ya es una solución completa no se puede expandir más por lo que sólo se expande el hijo 3:

Hijo3: {5, 6, 8, 17, 15, 10, 0} E: 15 S: 191

Hijo4: {7, 1, 16, 10} E: 15 S: 112

Una vez obtenida la descendencia, esta sustituye a los progenitores y se realiza una nueva iteración hasta alcanzar el máximo de iteraciones.

Técnicas complementarias

Como se ha visto previamente, las estrategias de búsqueda exploran el conjunto de los requisitos para encontrar la mejor combinación con mayor satisfacción. Pero para poder realizar eso es necesario previamente haberle asignado el valor que cada cliente asigna a cada requisito.

De forma adicional, se puede simplificar el problema sustituyendo cada grupo de requisitos acoplados por un único requisito que los contenga. De esta forma se reduce la gestión de dependencias que hay que realizar. Esta sustitución se deshace una vez terminada la ejecución del algoritmo quedando oculta para el consumidor. Por ejemplo:

Dependencias:

0 ↔ 3
0 → 4
3 → 6
8 → 0
9 X 3

10 → 4
10 → 6
8 → 10
9 X 10

Solución:

{4, 6, 10} → {0, 3, 4, 6}

En este ejemplo los requisitos 0 y 3 estaban acoplados, por lo que han sido sustituidos por el requisito 10. El requisito 10 pasa a tener las dependencias que tuvieran los requisitos 0 y 3, así como la suma de su valor y de su esfuerzo. Los requisitos que tuvieran alguna dependencia con 10 la verán actualizada. Una vez resuelto el problema, a la hora de presentar la solución, 10 se separa nuevamente en 0 y 3.

3. Arquitectura Orientada a Servicios

La arquitectura orientada a servicios (SOA por sus siglas en inglés *Service Oriented Architecture*) es un patrón de diseño que define servicios para solucionar las necesidades de los usuarios. Siendo un servicio una representación lógica de una actividad de negocio repetible con una salida específica. Los servicios pueden hacer uso de otros servicios y encapsulan la información haciendo que actúen como una caja negra para el consumidor del servicio.

Esta arquitectura define tres roles: el proveedor del servicio, el consumidor del servicio y el bróker del servicio.

- El proveedor se encarga de crear los servicios y publicarlos en el bróker.
- El consumidor busca los servicios en el bróker y los utiliza.
- El bróker clasifica los servicios que el proveedor registra y ofrece al consumidor los registros que busca.

Podemos definir un servicio como cualquier cosa útil podamos ofrecerle a un consumidor. En este caso un servicio va a ser una función con una interfaz bien definida que nos ayude a resolver parte o la totalidad de nuestro problema. Cada servicio contiene tres elementos: un contrato, una interfaz y una implementación:

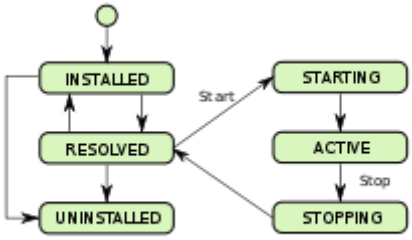
- El contrato especifica que ofrece un proveedor del servicio para satisfacer las necesidades de un consumidor del servicio.
- La interfaz define que ofrece un servicio y como usarlo.

- La implementación es el desarrollo en software de la especificación.

La aplicación de estas técnicas nos proporciona una alta interoperabilidad y es ideal para sistemas distribuidos ya que permite una fácil comunicación entre sistemas heterogéneos. Además también facilita la perdida de acoplamiento ya que minimiza las dependencias al encapsular la información. Esto provoca una mayor flexibilidad y escalabilidad.

OSGi: Java y SOA

Para la realización de este trabajo se ha utilizado OSGi (Open Services Gateway initiative) es un estándar para el desarrollo de SOA en java, concretamente el framework de Equinox. Cada servicio está implementado dentro de un bundle. El ciclo de vida de los bundles es el siguiente:



Primero se instalan, una vez instalados se comprueba si todas las clases que el bundle necesita están disponibles. Una vez resuelto el bundle puede activarse o desactivarse accediendo a los métodos start y stop de la clase Activator. Una vez acabe el ciclo de vida el bundle se desistala.

Además de la clase Activator, cada bundle contiene un fichero MANIFEST.FM que contiene la información del bundle y permite gestionar las dependencias.

Un bundle proveedor debe contener una clase con las operaciones que queramos implementar y que implemente una interfaz que conozca el consumidor. Una vez implementado el servicio este puede ser ofrecido haciendo uso del método registerService de la clase BundleContext. Dicho método deberá ser llamado utilizando la instancia de un atributo de tipo BundleContext que tiene la clase Activator y deberá llevar por parámetros la instancia del servicio que acabamos de generar, el nombre del servicio y un diccionario.

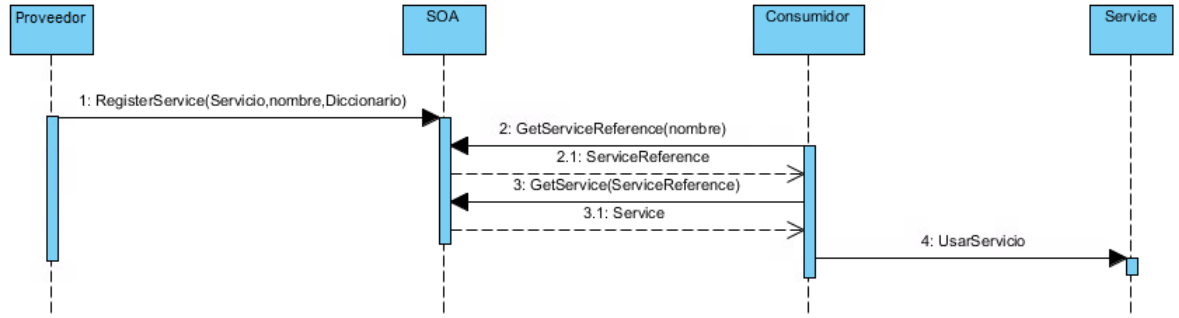
Un bundle consumidor debe acceder a los servicios que se le ofrece, para eso deberá hacer uso de dos métodos de la clase BundleContext, nuevamente haciendo uso de la instancia de la clase Activator:

- En primer lugar se hará uso del método getServiceReference que utiliza como parámetros un String con el nombre del servicio y que devuelve un objeto de tipo ServiceReference.
- En segundo lugar se utilizará el método getService que utiliza como parámetro el ServiceReference que acabamos de obtener y que devuelve la instnacia de la clase que el proveedor haya ofrecido.

Accediendo a la configuración de la ejecución, podemos definir qué bundles tienen mayor prioridad, siendo 0 la prioridad máxima, y también que bundles se activan automáticamente asignándole la propiedad true a Auto-Start.

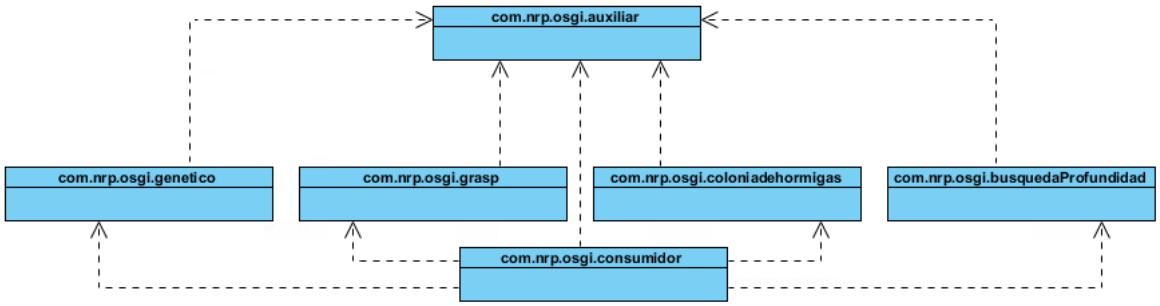
Bundles	Start Level	Auto-Start
com.nrp.osgi.auxiliar (1.0.0.qualifier)	default	false
com.nrp.osgi.busquedaProfundidad (1.0.0.qualifier)	2	default
com.nrp.osgi.coloniadehormigas (1.0.0.qualifier)	2	true
com.nrp.osgi.coloniadehormigaslter (1.0.0.qualifier)	1	default
com.nrp.osgi.consumidor (1.0.0.qualifier)	3	default
com.nrp.osgi.genetico (1.0.0.qualifier)	2	default
com.nrp.osgi.grasp (1.0.0.qualifier)	2	default
com.nrp.osgi.grasplter (1.0.0.qualifier)	1	default
Target Platform		

En el siguiente esquema podemos ver como el proveedor registra el servicio y el consumidor acceder a él. Una vez tiene una instancia del servicio deseado puede utilizarlo comunicándose con la clase del servicio.



SOA para el problema de la selección de requisitos

Podemos considerar cada una de las cuatro técnicas de búsqueda que hemos implementado como un servicio independiente, por tanto necesitaremos definir un bundle proveedor para cada uno de ellos, además necesitaremos un bundle consumidor que acceda a los servicios que estamos definiendo. Finalmente necesitaremos un sexto bundle librería en el que aparezcan todas las clases adicionales que van a necesitar nuestros servicios. Esto nos da como resultado una estructura parecida a la siguiente:



Paralelización de experimentos

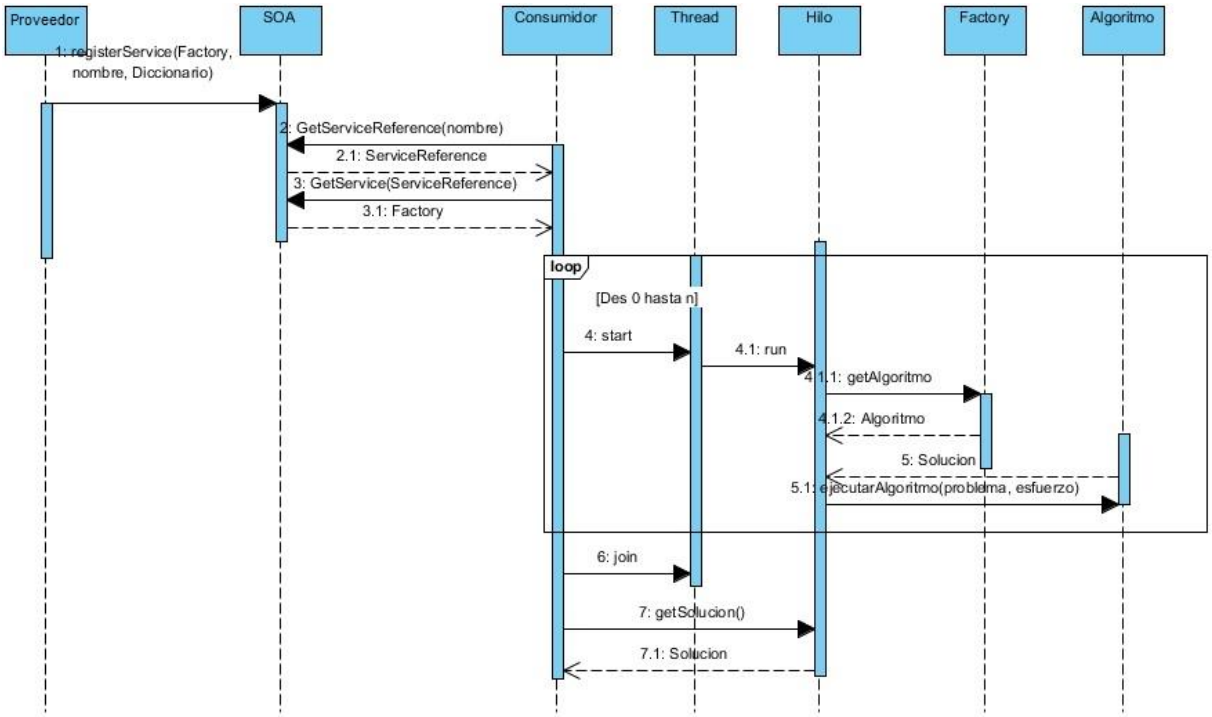
Esta estructura nos permitiría lanzar varias ejecuciones de los algoritmos en paralelo si tuviéramos un sistema distribuido realizando cada ejecución del algoritmo de forma independiente. Sin embargo por limitaciones de hardware tendremos que simularlo.

Para ello utilizaremos la clase Thread y la interfaz Runnable de java que nos permiten realizar programación concurrente. En primer lugar, crearemos una nueva clase Hilo que será la encargada de realizar la ejecución del servicio, lanzaremos cada Hilo con el comando start de la clase Thread y una vez lanzados todos esperaremos a que terminen utilizando el comando join de la misma clase. Al realizar esto hay que tener dos cosas en cuenta:

- La eficiencia de la ejecución no va a mejorar, puesto que los procesos no se ejecutan en paralelo, si no que van turnándose en la cpu. Además debido a las comunicaciones y técnicas adicionales es probable que el tiempo de ejecución aumente sensiblemente. De todas formas el tiempo que obtenemos no es de fiar, pues desde que empieza a medir la ejecución de un servicio hasta que este termina, la ejecución habrá cambiado de hilo varias veces.

- Es importante que cada uso de cada ejecución se corresponda a una instancia nueva de nuestro algoritmo para evitar conflictos en los atributos, para ello se emplea el patrón de diseño Factory, usando una clase constructora que genera una nueva instancia cada vez. También hay que vigilar que ningún atributo sea estático para evitar conflictos entre los diferentes hilos. En último lugar el propio problema a resolver también debe ser una instancia nueva cada vez, debido a que las dependencias se gestionan modificando atributos de la clase Requisito.

El esquema cambia con respecto al visto anteriormente, en primer lugar el proveedor no registra la instancia del algoritmo que implementa en sí, sino una factoría para generar nuevas instancias del mismo. Además el consumidor llama a Thread que a su vez llama a Hilo y es este el que actúa con el Factory y finalmente con el algoritmo implementado.



De momento hemos utilizado SOA para poder realizar simultáneamente varias iteraciones de diversos algoritmos, sin embargo podemos intentar buscar partes de nuestros algoritmos que podamos ejecutar en paralelo, estas partes las podemos identificar fácilmente en el grasp y en la colonia de hormigas.

Grasp realiza varias iteraciones independientes en las que como hemos visto anteriormente primero genera una solución aleatoria y posteriormente intenta buscar el máximo local. Por tanto podemos encapsular cada iteración como un servicio independiente y lanzarlos todos a la vez utilizando la clase Hilo que hemos creado anteriormente.

En la colonia de hormigas la situación es diferente, cada iteración se actualizan las feromonas por lo que hay que realizarlas de forma iterativa. Sin embargo, en cada iteración generamos un determinado número de hormigas, estas hormigas si podemos lanzarlas en paralelo. Es importante asegurarnos a la hora de actualizar las feromonas que las estamos actualizando en la instancia principal que utilizamos para generar nuevas hormigas y no en la instancia propia de cada hormiga.

4. Análisis del proyecto

Una vez explicados el problema a resolver y las técnicas a emplear es necesario realizar el análisis del proyecto software a desarrollar. Este análisis incluirá una especificación formal del proyecto, una especificación de los requisitos del proyecto y la planificación temporal del proyecto.

Especificación del proyecto

Será necesaria la implementación en java de las técnicas heurísticas de búsqueda expuestas anteriormente, grasp, algoritmo genético y colonia de hormigas. Estas técnicas se desarrollaran primero de forma tradicional y posteriormente se construirán como servicios utilizando OSGi.

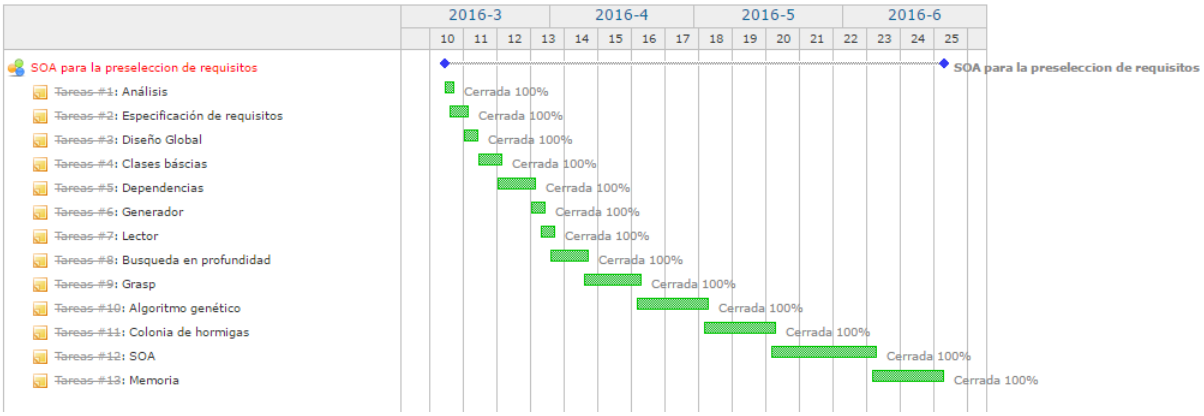
Paralelamente al desarrollo de los algoritmos se realizaran dos aplicaciones con interfaz gráfica para probar el funcionamiento de los algoritmos, una aplicación se utilizará para generar problemas y otra para resolverlos.

Desarrollo de proyectos ágiles

El desarrollo de proyectos ágiles está basado en el desarrollo iterativo e incremental, en cada iteración se realizan todas las etapas del ciclo de vida de la ingeniería del software de una parte del producto final. Este tipo de desarrollo es ideal para la realización de un proyecto de fin de grado ya que permite el desarrollo de partes completas que puedes mostrar a tu tutor.

Scrum es una metodología ágil en la que se van realizando “sprints” de 15 a 30 días. Al principio de cada sprint se decide que tareas se van a llevar a cabo. Al final se comprueba que se ha llevado a cabo y se mira con retrospectiva como ha ido al sprint.

Para la planificación del programa se ha utilizado una herramienta web llamada redmine cuyo cronograma fina les el siguiente:



Especificación de requisitos

Objetivos

OBJ-01	Solucionar el problema de la selección de requisitos
Versión	1.0
Autor	José Luis García Sánchez
Descripción	El sistema deberá poder resolver el problema de la selección de requisitos. Para ello hará uso de técnicas heurísticas de búsqueda y de una arquitectura orientada a servicios.
Comentarios	Las técnicas heurísticas serán concretamente: un algoritmo GRASP, un algoritmo genético y un algoritmo de colonia de hormigas.

OBJ-02	Generar juegos de pruebas
Versión	1.0
Autor	José Luis García Sánchez
Descripción	El sistema deberá ser capaz de generar juegos de pruebas aleatorios para el problema de la selección de requisitos.
Comentarios	Este proceso se realizará en una aplicación a parte.

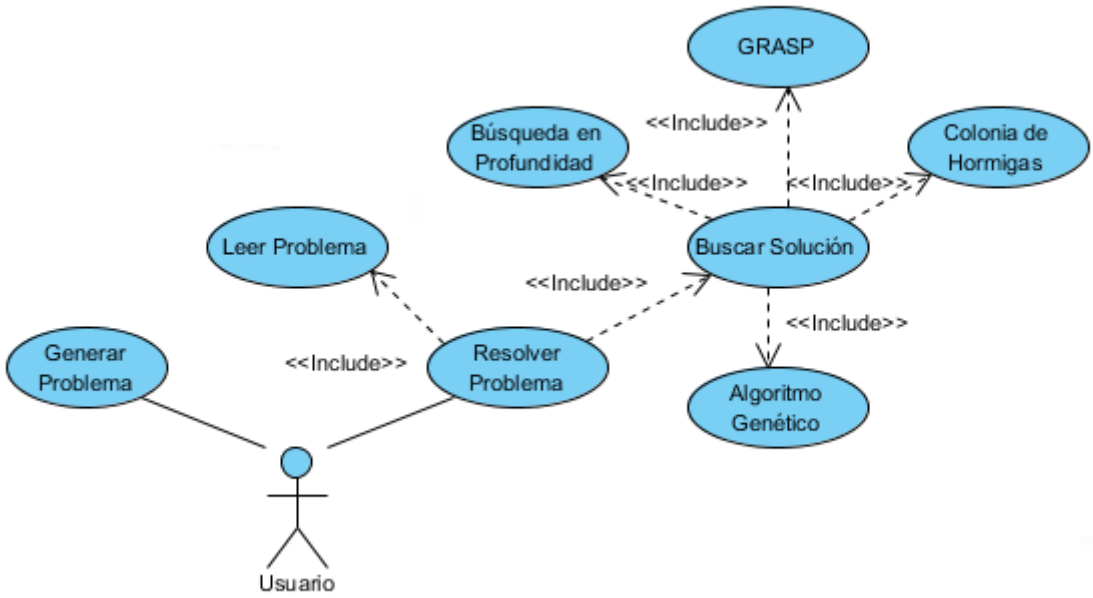
OBJ-03	Leer juegos de pruebas
Versión	1.0
Autor	José Luis García Sánchez
Descripción	El sistema deberá ser capaz de leer los juegos de pruebas aleatorios generados.
Comentarios	Ninguno.

OBJ-04	Medir la eficiencia de los procesos
Versión	1.0
Autor	José Luis García Sánchez
Descripción	El sistema deberá ser capaz de medir el tiempo de cada ejecución.
Comentarios	Ninguno.

Requisitos de información

RI-01	Problema
Objetivos asociados	OBJ-02 y OBJ-03
Requisitos asociados	
Descripción	Problema completo que contiene los requisitos, los clientes y las dependencias.
Datos Específicos	ID Requisito Esfuerzo Requisito Prerrequisitos Requisitos acoplados Requisitos excluyentes ID Cliente Peso Cliente Valor Requisito Cliente
Comentarios	No se trata de una base de datos, cada problema es un fichero.txt generado por el sistema.

Diagrama de casos de uso



Actores y roles

Act-01	Usuario
Roles asociados	Rol-01
Descripción	El usuario del sistema.
Comentarios	El único actor del sistema.

Rol-01	Usuario
Actores asociados	Act-01
Descripción	El usuario del sistema.
Comentarios	El único rol del sistema.

Requisitos funcionales

RF-01	Resolver problema.
Objetivos asociados	OBJ-01, OBJ-03, y OBJ-04
Requisitos Asociados	RF-02, RF-03
Actores	Act-01
Roles	Rol-01
Descripción	El usuario podrá cargar problemas previamente generados, así como utilizar alguna de las técnicas de búsqueda implementadas para resolverlo.
Precondición	Ninguna.
Secuencia normal	1. El usuario elige si quiere cargar un problema o resolver uno.
Postcondición	Ninguna.
Excepciones	Ninguna.
Comentarios	Es la ventana principal del sistema.

RF-02	Buscar Solución
Objetivos asociados	OBJ-01 y OBJ-04
Requisitos Asociados	RF-01, RF-04, RF-05, RF-06 y RF-07
Actores	Act-01
Roles	Rol-01
Descripción	El sistema deberá realizar técnicas de búsqueda para resolver el problema de la selección de requisitos. Se medirá el tiempo que tarde en ejecutar el proceso.
Precondición	Deberá haber algún problema precargado.
Secuencia normal	1. El usuario selecciona un problema. 2. El usuario indica un esfuerzo máximo. 3. El usuario indica un número de iteraciones. 4. El usuario selecciona una técnica de búsqueda. 5. El sistema comienza a medir el tiempo. 6. El sistema ejecuta esa técnica de búsqueda. 7. El sistema realiza un postprocesamiento. 8. El sistema deja de medir el tiempo. 9. El sistema muestra la solución y el tiempo de ejecución.
Postcondición	Ninguna.
Excepciones	Si no hay ningún problema precargado no se puede realizar.

Comentarios	El postprocesamiento consiste en separar los requisitos acoplados.

RF-03	Leer problemas
Objetivos asociados	OBJ-03
Requisitos Asociados	RF-01
Actores	Act-01
Roles	Rol-01
Descripción	El usuario podrá cargar los juegos de prueba que se hayan generado previamente.
Precondición	Debe haber algún juego de prueba en la carpeta pruebas. El fichero debe tener un formato válido.
Secuencia normal	1. El usuario selecciona el problema que quiere cargar. 2. El sistema lee el problema en el siguiente orden: a. Leer requisitos. b. Leer dependencias. c. Leer clientes. 3. El sistema realiza un preprocesamiento: a. Asignar valor dado por cada cliente a cada requisito. b. Unir los requisitos acoplados.
Postcondición	El problema queda guardado para poder ser resuelto.
Excepciones	Si no hay ningún fichero en la carpeta pruebas no se podrá realizar. Si el fichero no tiene el formato válido, se interrumpirá la lectura.
Comentarios	Ninguno

RF-04	Busqueda en profundidad
Objetivos asociados	OBJ-01
Requisitos Asociados	RF-02
Actores	Act-01
Roles	Rol-01
Descripción	El sistema utiliza un algoritmo de búsqueda en profundidad para resolver el problema.
Precondición	El número de iteraciones debe ser al menos de 1.
Secuencia normal	1. Se inicializan las dependencias. 2. Se genera un nodo inicial vacío. 3. Se exploran todas las combinaciones válidas. 4. Se devuelve la solución con más satisfacción.
Postcondición	Ninguna.
Excepciones	Ninguna.
Comentarios	Ninguno.

RF-05	Grasp
Objetivos asociados	OBJ-01
Requisitos Asociados	RF-02
Actores	Act-01
Roles	Rol-01
Descripción	El sistema utiliza un grasp para resolver el problema.
Precondición	El número de iteraciones debe ser al menos de 1.
Secuencia normal	1. Se inicializan las dependencias.

	2. Se genera una solución aleatoria. 3. Se generan los nodos padre de esa solución. 4. Se realiza una búsqueda en profundidad partiendo de esos nodos para buscar al mejor vecino. 5. Se repite hasta que la solución no mejora. 6. Se repite hasta que se acabe el número de iteraciones. 7. Se devuelve la solución con más satisfacción.
Postcondición	Ninguna.
Excepciones	Ninguna.
Comentarios	Ninguno.

RF-06	Algoritmo genético
Objetivos asociados	OBJ-01
Requisitos Asociados	RF-02
Actores	Act-01
Roles	Rol-01
Descripción	El sistema utiliza un algoritmo genético para resolver el problema.
Precondición	El número de iteraciones debe ser al menos de 1.
Secuencia normal	1. Se inicializan las dependencias. 2. Se genera una población inicial. 3. Se realiza una selección de los progenitores más propensos a tener descendencia. 4. Se realiza un cruce entre dichos progenitores. 5. Los descendientes pasan a ser los progenitores de la siguiente generación. 6. Se repite hasta que se acabe el número de iteraciones. 7. Se devuelve la solución con más satisfacción.
Postcondición	Ninguna.
Excepciones	Ninguna.
Comentarios	Ninguno.

RF-07	Colonia de hormigas
Objetivos asociados	OBJ-01
Requisitos Asociados	RF-02
Actores	Act-01
Roles	Rol-01
Descripción	El sistema utiliza un algoritmo de colonia de hormigas para resolver el problema.
Precondición	El número de iteraciones debe ser al menos de 1.
Secuencia normal	1. Se inicializan las dependencias. 2. Se genera una hormiga. 3. Se mueve la hormiga hasta encontrar una solución completa. 4. Se repite hasta que se hayan generado el número de hormigas de cada generación. 5. Se actualizan las feromonas. 6. Se repite hasta que se acabe el número de iteraciones. 7. Se devuelve la solución con más satisfacción.
Postcondición	Ninguna.

Excepciones	Ninguna.
Comentarios	Ninguno.

RF-08	Generar problemas
Objetivos asociados	OBJ-02
Requisitos Asociados	Ninguno.
Actores	Act-01
Roles	Rol-01
Descripción	El usuario podrá generar juegos de prueba aleatorios a partir de los parámetros que el indique.
Precondición	Los parámetros deben ser válidos
Secuencia normal	1. El usuario introduce los parámetros: a. Introducir el número de clientes. b. Introducir el número de requisitos. c. Introducir el esfuerzo de los requisitos. d. Introducir el peso de los clientes. e. Introducir el valor que le dan los clientes a los requisitos. f. Introducir el número de prerrequisitos. g. Introducir el número de requisitos acoplados. h. Introducir el número de requisitos excluyentes. i. Introducir el nombre del fichero. 2. El sistema genera el problema: a. Generar los requisitos b. Generar las dependencias: i. Generar requisitos acoplados. ii. Generar prerrequisitos. iii. Generar requisitos excluyentes. c. Generar los clientes
Postcondición	Un guardará un fichero en la carpeta pruebas dentro del proyecto.
Excepciones	Si algún parámetro no es válido, aparecerá una ventana emergente indicando los problemas. Si ya existe un problema con ese nombre el sistema nos preguntará si deseamos reemplazarlo.
Comentarios	Ninguno.

Requisitos no funcionales

RNF-01	Arquitectura Orientada a Servicios
Objetivos asociados	OBJ-01
Requisitos asociados	RF-02
Descripción	El sistema deberá estar implementado utilizando una Arquitectura Orientada a Servicios, concretamente su estándar para java OSGi.
Comentarios	Se utiizará el plugin Equinox para la implementación de OSGi.

RNF-02	Java
Objetivos asociados	Todos.
Requisitos asociados	Todos.
Descripción	El sistema se desarrollará utilizando java como lenguaje.
Comentarios	Ninguno.

RNF-03	Gestión de dependencias
Objetivos asociados	OBJ-01
Requisitos asociados	RF-02
Descripción	El sistema deberá poder gestionar las dependencias de los requisitos para el correcto funcionamiento de los algoritmos.
Comentarios	Ninguno.

RNF-04	Fiabilidad
Objetivos asociados	Todos.
Requisitos asociados	Todos.
Descripción	El sistema debe ser fiable.
Comentarios	Para comprobar su fiabilidad se desarrollaran test de JUnit.

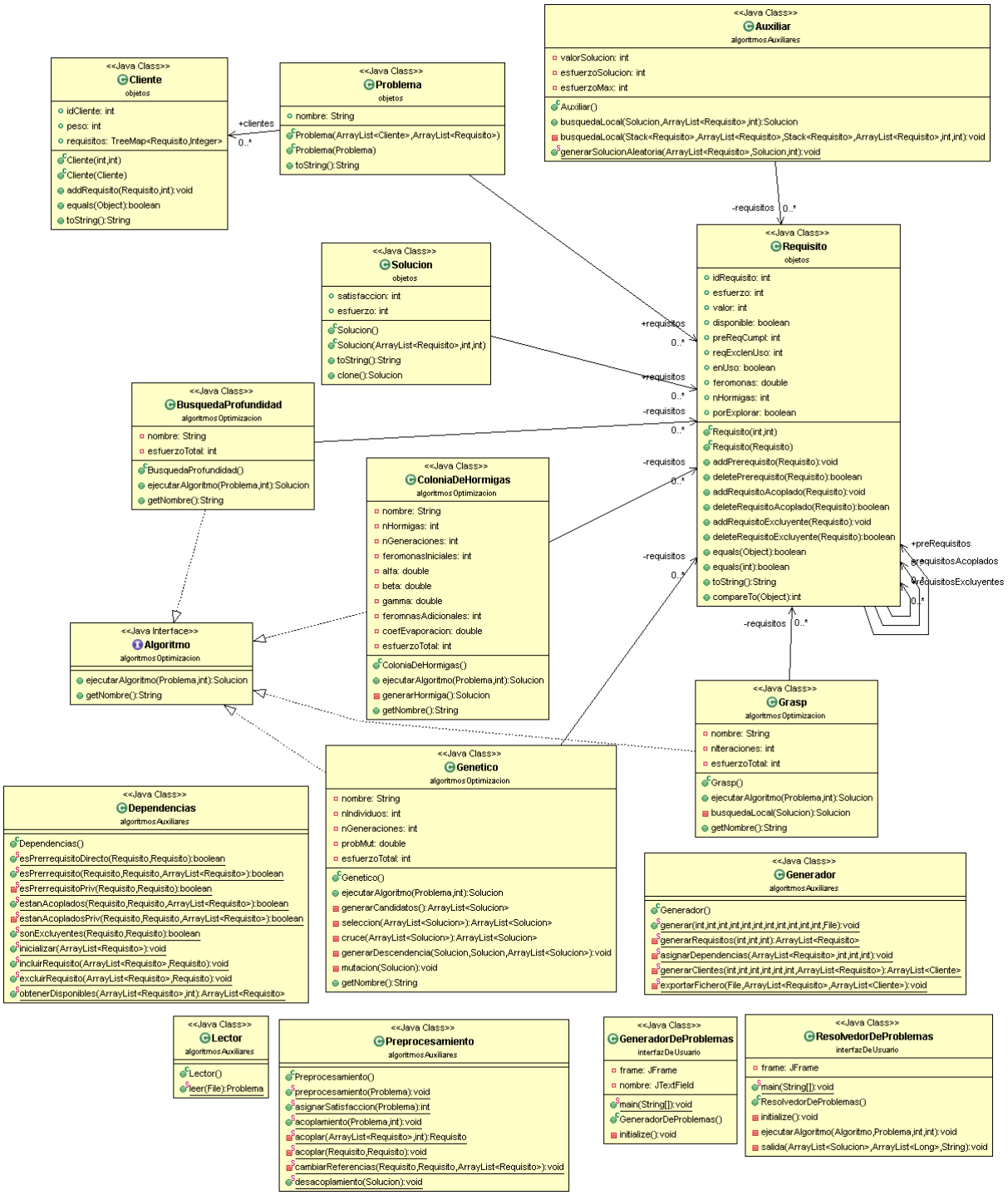
RNF-05	Eficiencia
Objetivos asociados	OBJ-01
Requisitos asociados	RF-02
Descripción	Las técnicas heurísticas de búsqueda renuncian a ser óptimas a cambio de ser más eficientes.
Comentarios	Ninguno.

RNF-06	Robustez
Objetivos asociados	OBJ-01, OBJ-02 y OBJ-03
Requisitos asociados	RF-02, RF-03 y RF-04
Descripción	El sistema debe ser capaz de dar una respuesta a cualquier entrada que proporcione el usuario.
Comentarios	Los test de JUnit también trabajan en situaciones límite para poner a prueba la robustez del sistema.

5. Diseño del proyecto

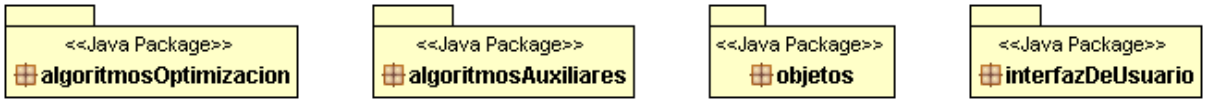
El proyecto tiene dos implementaciones, en primer lugar una implementación tradicional en un proyecto java y además una implementación adicional del solucionador de problemas siguiendo una arquitectura orientada a servicios.

Diagramas de clases de la implementación tradicional



Organización del proyecto

Todas las clases mostradas anteriormente en el diagrama están agrupadas en 4 packages en función de su utilidad:

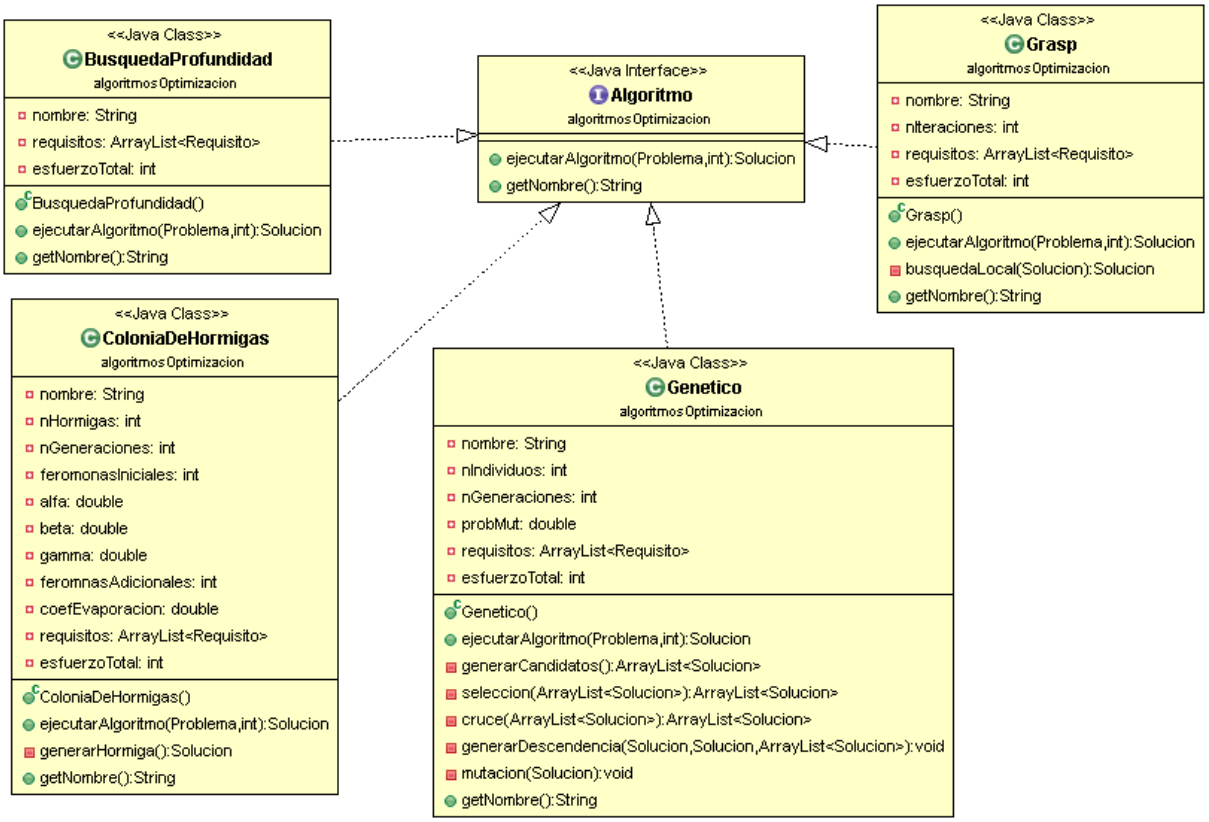


Algoritmos de Optimización contiene la interfaz Algoritmo y sus 4 implementaciones, Algoritmos Auxiliares contiene métodos comunes a varias implementaciones, objetos contiene las clases de las que estos métodos hacen uso e interfaz de usuario contiene las clases de la interfaz gráfica.

Algoritmos de Optimización

La interfaz algoritmo tiene dos métodos, el método ejecutar algoritmo que es el encargado de buscar la solución del problema, y el método getNombre que indica la implementación que estamos utilizando.

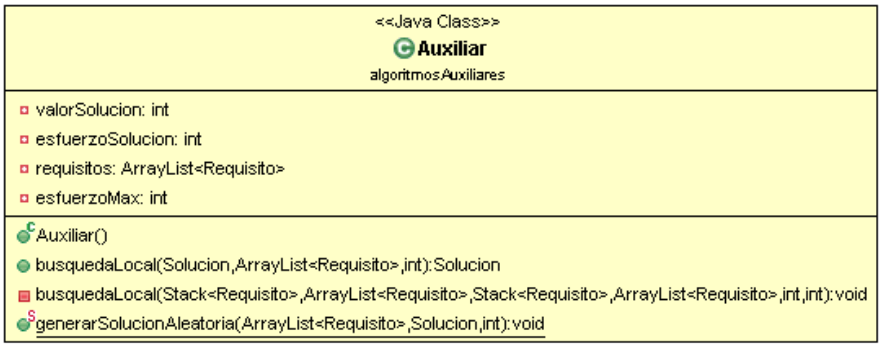
Cada implementación tiene un constructor además de los dos métodos que implementan así como algunos métodos y atributos privados.

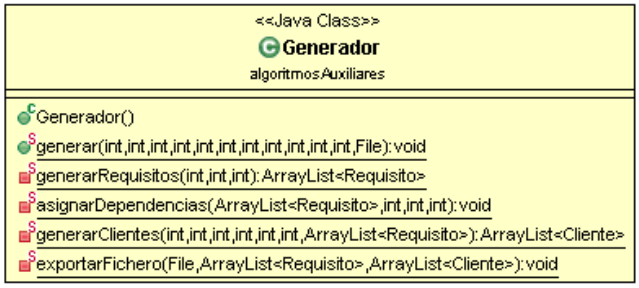


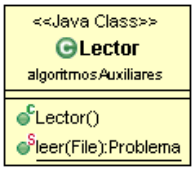
Algoritmos Auxiliares

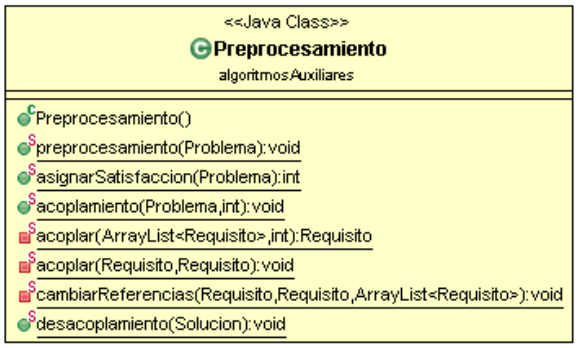
Este grupo está formado por 5 clases:

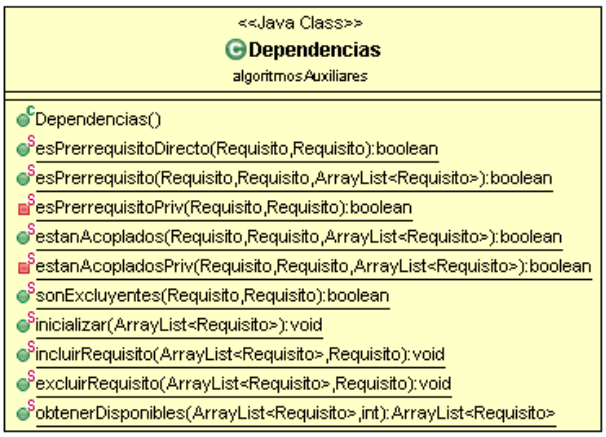
- La clase Auxiliar que contiene los métodos para realizar una búsqueda local y generar una solución aleatoria, además de su constructor y varios atributos privados.
- La clase Generador contiene un método público para generar un fichero con un problema, contiene varios métodos privados que se corresponden con varias etapas de dicho proceso.
- La clase Lector contiene un método para leer el fichero.
- La clase Preprocesamiento contiene un método para realizar el preprocesamiento, y también se pueden ejecutar por separado las dos etapas de dicho proceso: asignar satisfacción y acoplar requisitos, teniendo este proceso varios métodos privados por debajo. También está contenido en esta clase un método para desacoplar los requisitos.
- La clase Dependencias que contiene todos los métodos para identificar las dependencias entre dos requisitos, para inicializar las dependencias, para actualizarlas cuando se incluye o excluye un requisito y un método para obtener una lista con los requisitos disponibles.









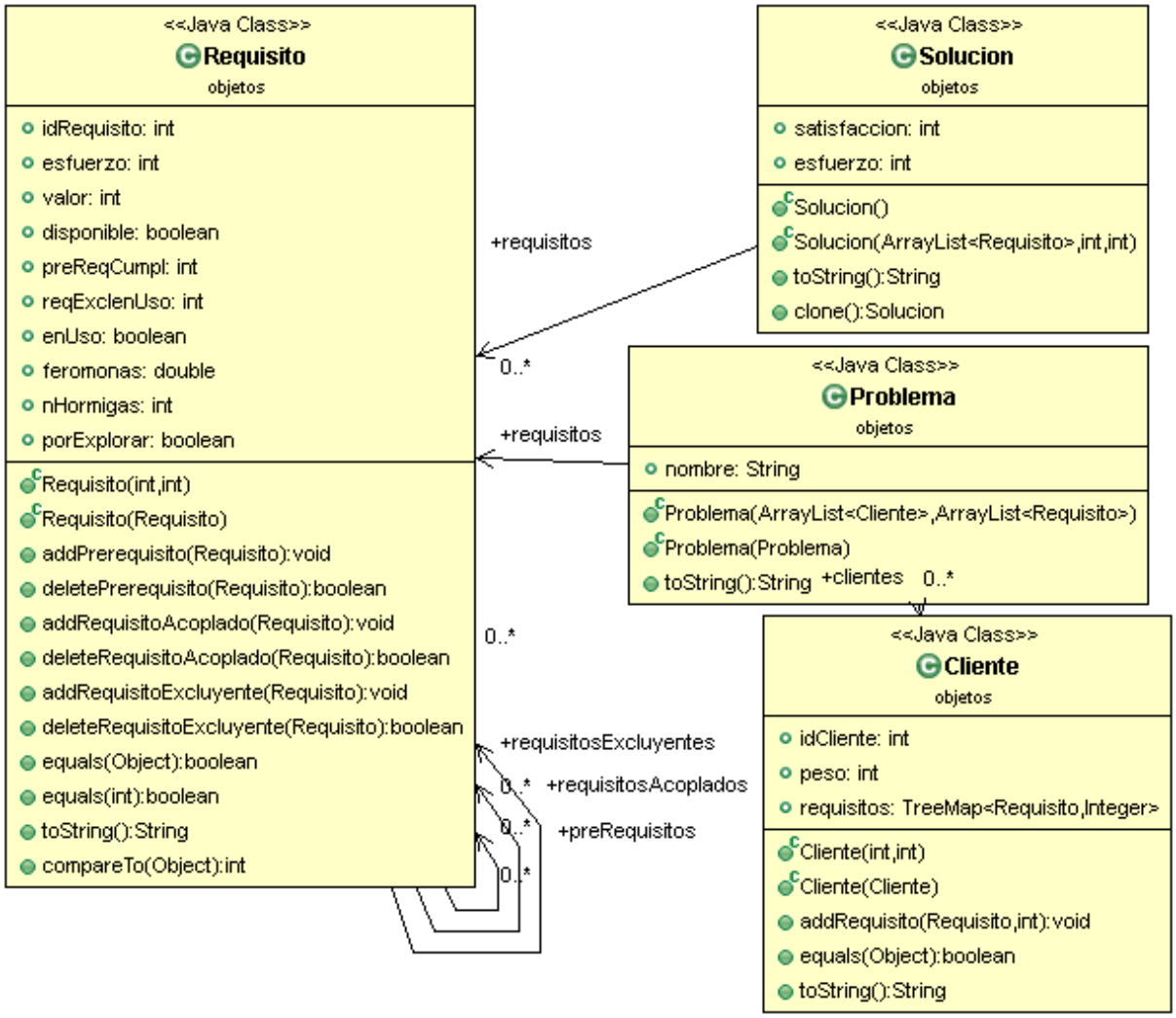


Objetos

Este grupo está formado por 4 clases:

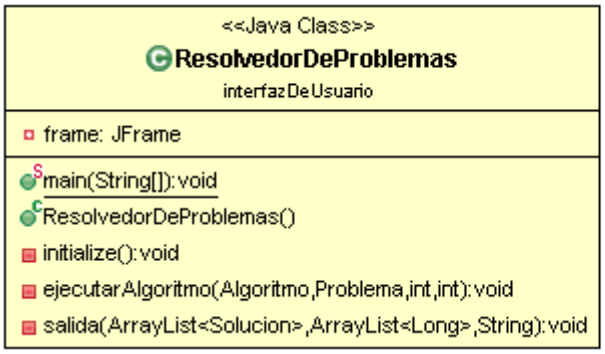
- La clase Problema contiene un conjunto de requisitos y clientes y la clase Solución contiene un conjunto de requisitos.
- La clase Cliente contiene la información relativa a los clientes de nuestro problema.
- La clase Requisito contiene la información relativa a los requisitos de nuestro problema, varios atributos para poder gestionar si se están cumpliendo las dependencias y un par de atributos para gestionar las feromonas en la colonia de hormigas.

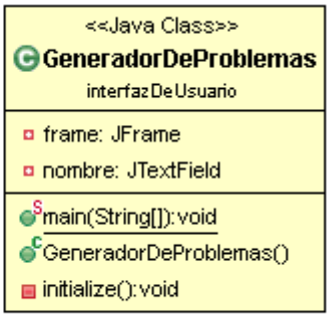
Todos los atributos de estas clases son públicos para ganar eficiencia en la gran cantidad de accesos que hay que realizar a estas clases.



Interfaz de Usuario

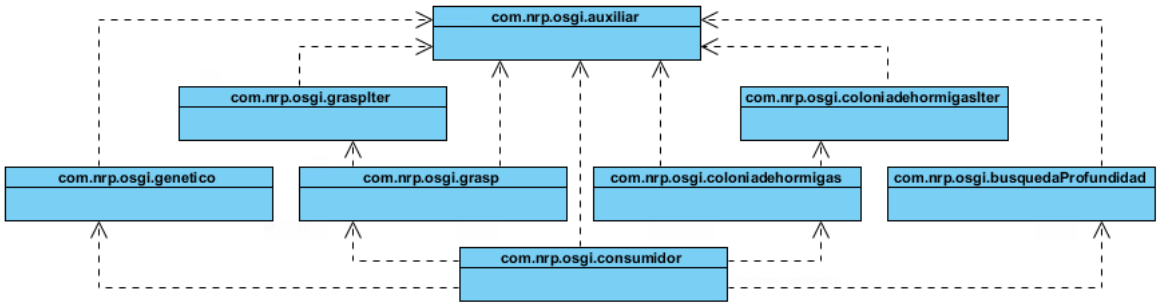
Estas dos clases contienen un atributo privado de tipo JFrame y un método main para ejecutar la aplicación.



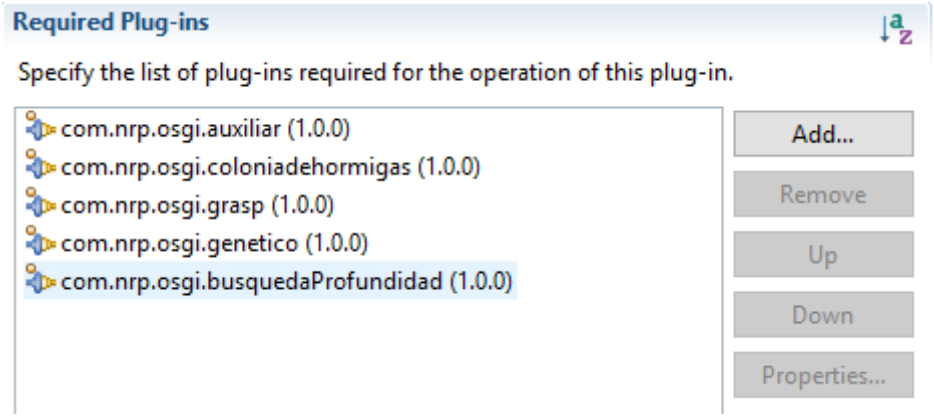


Diagramas de clases de la implementación SOA

Como hemos visto antes cada servicio se implementa en un bundle que es un proyecto plug-in independiente. La estructura de los bundles es la siguiente:



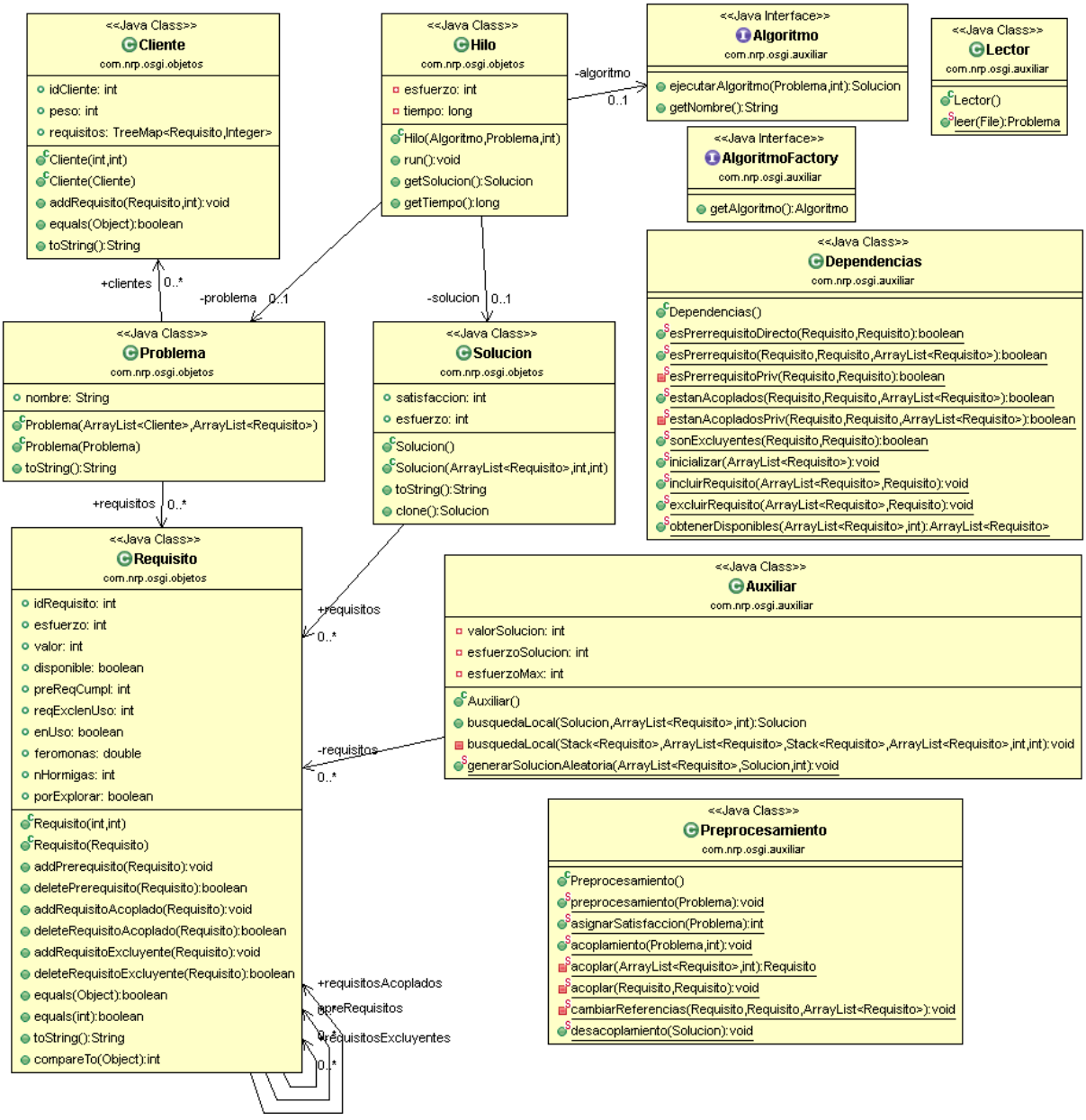
Las flechas representan las dependencias entre los distintos bundles. Estas dependencias las añadimos en un fichero llamado Manifest.MF:



com.nrp.osgi.auxiliar

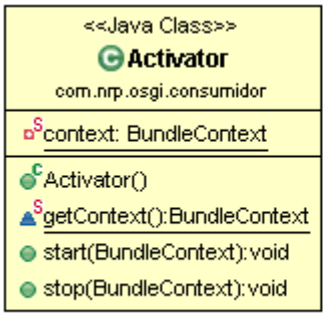
Este proyecto no contiene una clase Activator ya que actúa como librería. Contiene una copia de las clases contenidas en las packages objetos y algoritmos auxiliares excepto Generador.

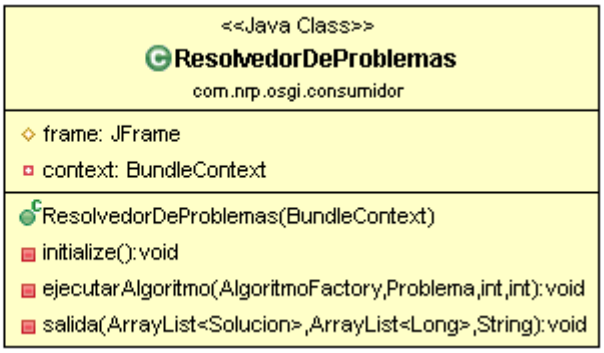
Contiene también la interface Algoritmo, la interface AlgoritmoFactory, que tiene un método getAlgoritmo que devuelve un objeto de tipo Algoritmo y la clase Hilo que implementa Runnable. A su constructor hay que pasarle por parámetros el algoritmo que se quiere utilizar, el problema que se quiere resolver y el esfuerzo del que se dispone. El método run que implementa realiza la ejecución del algoritmo. Con los métodos getSolución y getTiempo devuelven la Solución que obtiene y el tiempo que tarda en realizar la ejecución.



com.nrp.osgi.consumidor

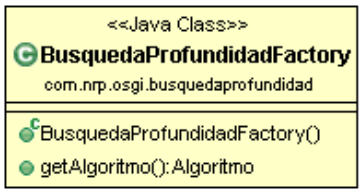
Este proyecto contiene la clase Activator y una modificación de la clase ResolvedorDeProblemas. Esta no tiene el método main y JFrame es protegido en vez de privado, esto se debe a que es Activator quien realiza la ejecución.

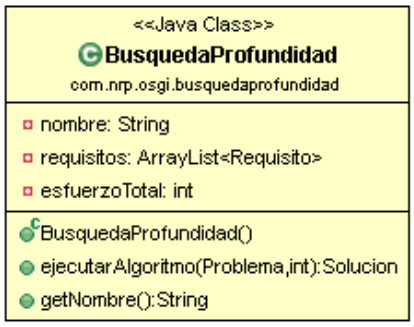


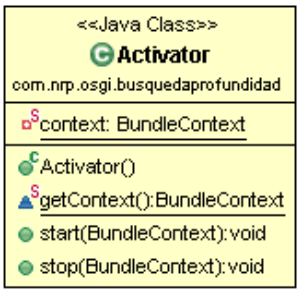


Todas los demás proyecto se limitan a tener una clase que implemente la interfaz Algoritmo, otra que implemente la interfaz AlgoritmoFactory y la clase Activator.

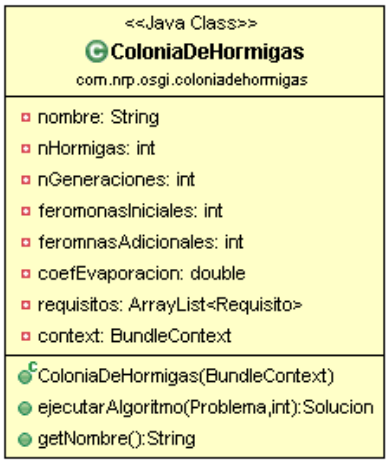
com.nrp.osgi.busquedaProfundidad

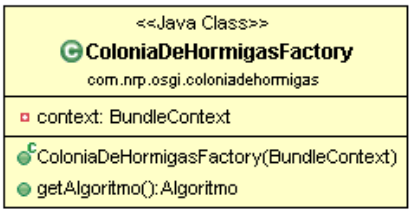


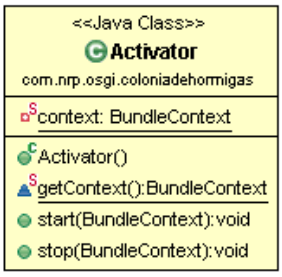




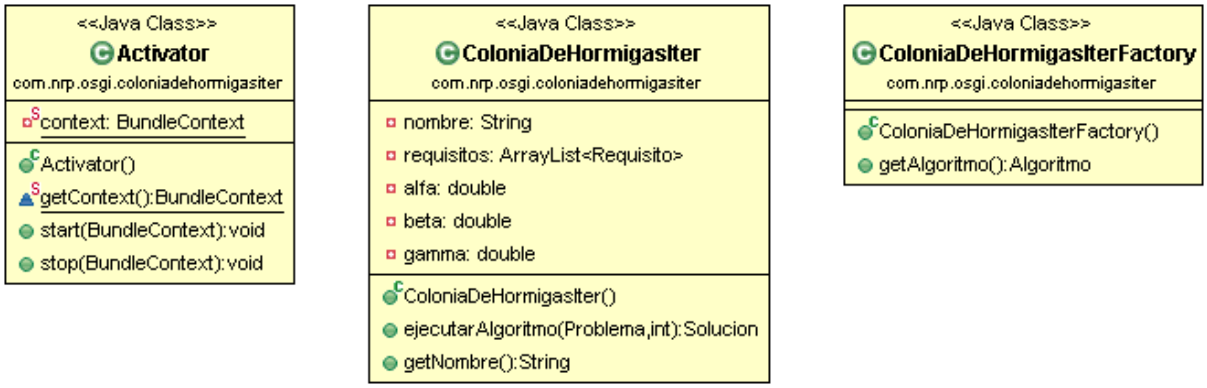
com.nrp.osgi.coloniadehormigas



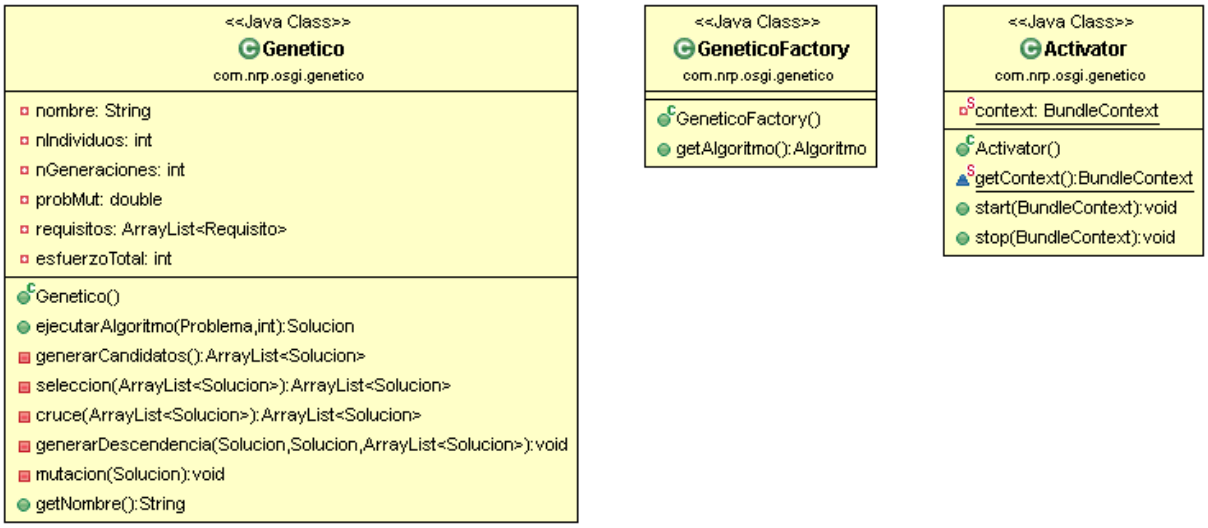




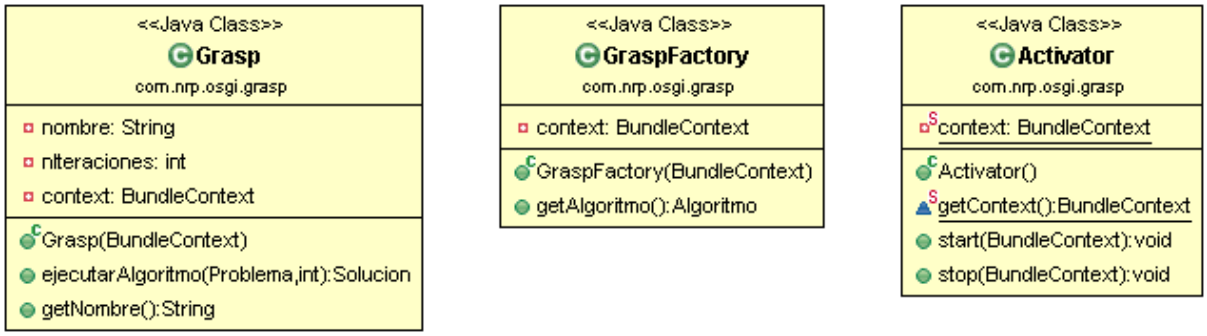
com.nrp.osgi.coloniadehormigasIter



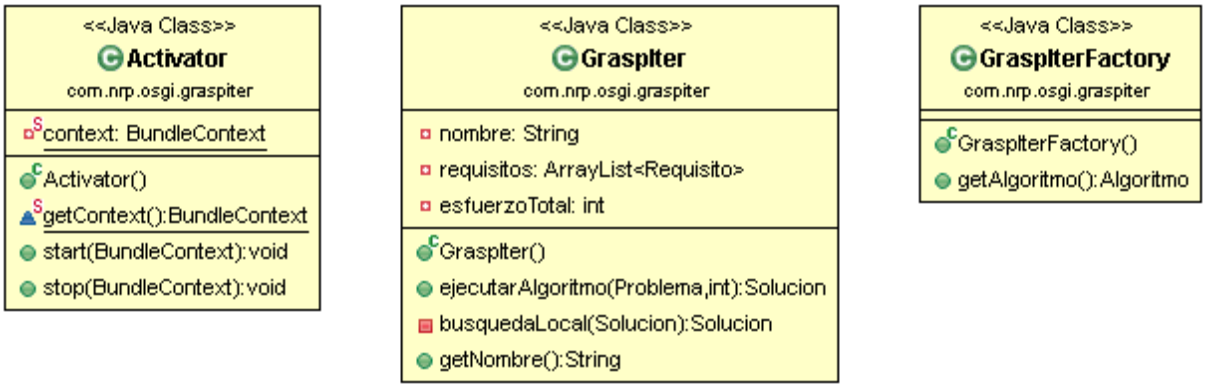
com.nrp.osgi.genetico



com.nrp.osgi.grasp



com.nrp.osgi.grasplter

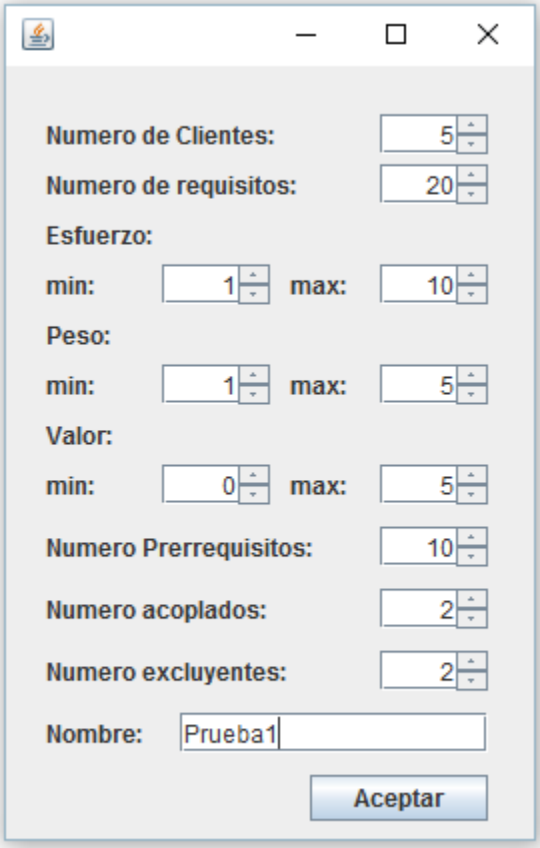


Interfaces de Usuario

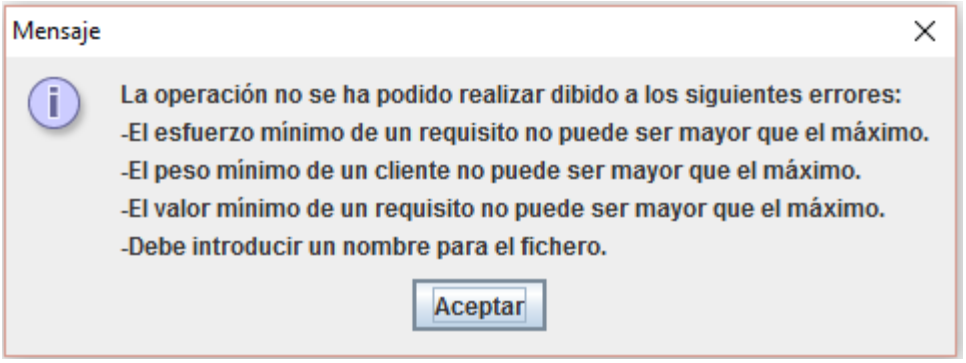
Se han desarrollado dos aplicaciones, una para generar y otra para resolver los problemas.

Generador

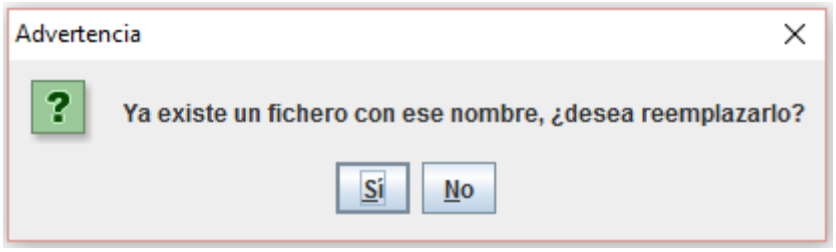
La ventana principal es la siguiente:



Si hay algún error en algún campo aparece un mensaje de error indicando el problema:



Si ya existe un fichero con ese nombre, aparece una ventana de confirmación:

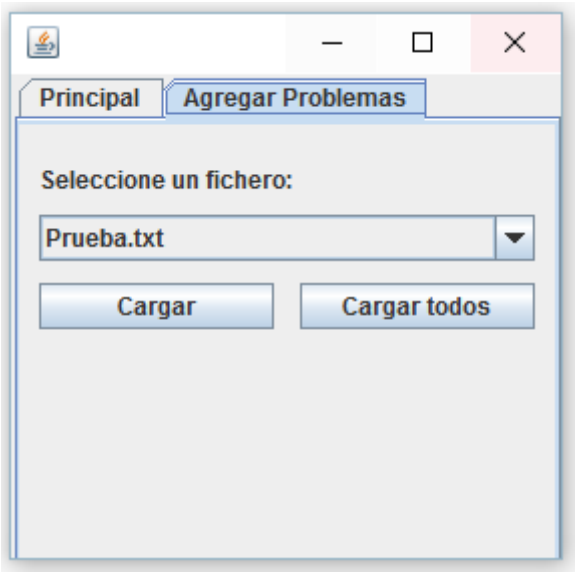


Si todo funciona correctamente se genera un fichero con la siguiente estructura:

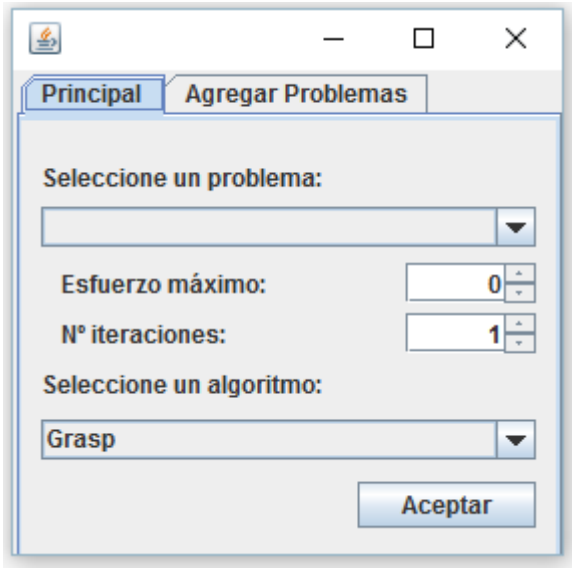
```
REQUISITOS
0 5 1 4 2 8 3 6 4 1 5 8 6 4 7 3 8 3 9 2
DEPENDENCIAS
0 / / /
1 / / / 0
2 / 7 / /
3 / / /
4 / / / 7
5 / / /
6 / / / 7
7 / / /
8 / / 0 / 3
9 / 5 4 2 / /
CLIENTES
0 3 / 0 1 1 1 2 4 3 2 4 1 5 4 6 4 7 4 8 3 9 1
1 2 / 0 4 1 3 2 2 3 4 4 2 5 1 6 2 7 3 8 3 9 3
2 4 / 0 2 1 1 2 1 3 3 4 2 5 2 6 4 7 1 8 1 9 2
3 3 / 0 3 1 1 2 2 3 3 4 4 5 2 6 4 7 2 8 1 9 2
4 2 / 0 2 1 1 2 1 3 3 4 2 5 4 6 3 7 2 8 4 9 2
```

Aplicación para resolver problemas

Esta aplicación tiene dos pestañas, una para leer los problemas:



Y otra para resolver los problemas leído previamente:



La salida muestra la solución y los tiempos:

```
Se ha compleatado la ejecución del algoritmo Grasp
-------------------------------------------
La ejecución más larga duró 5 ms
La ejecución más corta duró 4 ms
La ejecución media duró 4 ms
La mejor solución fue Satisfacción=179
Esfuerzo=21
Requisitos: 8 0 6 5 4
La peor solución fue Satisfacción=179
Esfuerzo=21
Requisitos: 8 0 6 5 4
```

6. Uso y Pruebas

Guía de uso

Para usar el generador se deben rellenar todos los campos con las características que se deseen que tenga el problema generado. Siendo estas características el número de clientes, el número de requisitos, los valores mínimos y máximos del esfuerzo de los requisitos, del peso de los clientes y del valor que tienen los requisitos para los clientes. Una gran diferencia entre el mínimo y el máximo nos permitirá identificar más fácilmente una buena y una mala solución. Además también habrá que indicar el número de prerrequisitos, requisitos excluyentes y requisitos acoplados. Recordando que este número estará acotado como se indicó al comienzo de este documento.

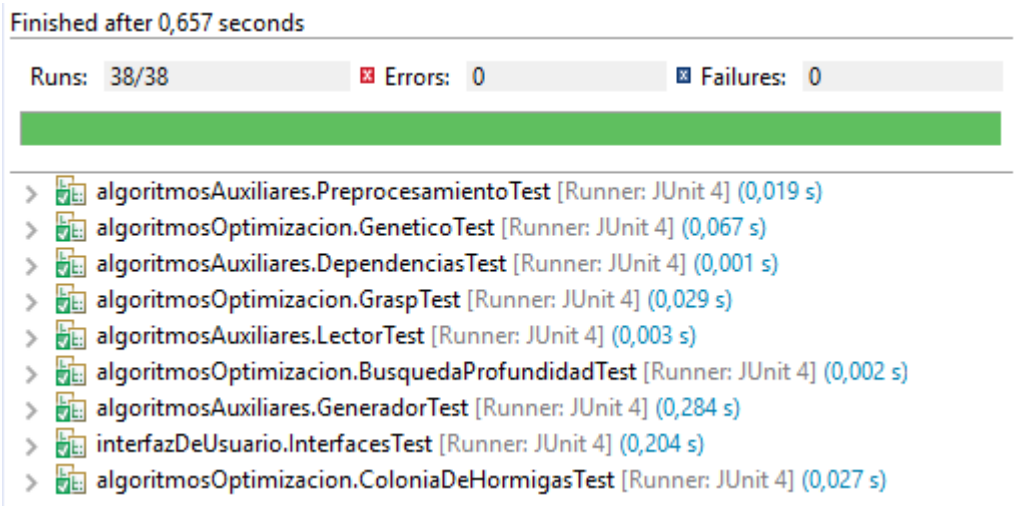
Una vez generado algunos problemas se podrá utilizar la otra aplicación, desde el punto de vista del usuario será igual el uso de la aplicación SOA y la aplicación tradicional. Sin embargo, si vamos a ejecutar SOA es importante configurar el directorio de trabajo, en este caso se ha decidido que sea en el propio proyecto nrp para no tener que copiar las pruebas. Esto se indica en run configuratios, OSGi framework, Argumets:

En primer lugar deberemos cargar los problemas que queramos resolver, podemos cargarlos de un en uno o cargarlos todos a la vez. Una vez cargados podremos seleccionarlas, indicar el esfuerzo máximo, el número de ejecución y el algoritmo deseado.

Pruebas

Para garantizar la calidad de la aplicación se comprueba su correcto funcionamiento de la aplicación mediante la creación varias pruebas unitarias utilizando JUnit y se ha utilizado el plug-in de eclipse EclEmma Java Code Coverage, para comprobar que porcentaje del código está probado.

En total disponemos de 38 test unitarios:



Esto supone una cobertura de un 80%, correspondiendo ese 20% fundamentalmente a métodos de las clases objeto que no se llegan a utilizar y a partes de la interfaz de usuario que necesitan interacción.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
nrp	84,8 %	6.586	1.177	7.763
src	79,6 %	4.353	1.118	5.471
interfazDeUsuario	56,3 %	949	736	1.685
ResolvedorDeProblemas.java	51,5 %	443	417	860
GeneradorDeProblemas.java	61,3 %	506	319	825
objetos	63,3 %	368	213	581
Solucion.java	26,3 %	26	73	99
Problema.java	12,7 %	9	62	71
Requisito.java	84,1 %	244	46	290
Cliente.java	73,6 %	89	32	121
algoritmosOptimizacion	91,7 %	1.100	99	1.199
Grasp.java	59,6 %	115	78	193
ColoniaDeHormigas.java	96,0 %	316	13	329
Genetico.java	99,2 %	635	5	640
BusquedaProfundidad.java	91,9 %	34	3	37
algoritmosAuxiliares	96,5 %	1.936	70	2.006
Generador.java	93,8 %	473	31	504
Auxiliar.java	91,2 %	312	30	342
Dependencias.java	99,3 %	405	3	408
Lector.java	99,1 %	338	3	341
Preprocesamiento.java	99,3 %	408	3	411
test	97,4 %	2.233	59	2.292

Pruebas de eficiencia

Las pruebas de eficiencia de los métodos de búsqueda se han realizado sobre la implementación clásica de los mismos debido a que la falsa concurrencia falsea los datos de la implementación SOA.

Los problemas tendrán un tamaño de 25, 50 y 100 requisitos respectivamente y se permitirá un esfuerzo máximo de 150. Se realizarán 20 ejecuciones y se obtendrá el máximo, el tercer cuartil, la mediana, el primer cuartil y el mínimo del tiempo de ejecución (en milisegundos), la satisfacción de la solución y el esfuerzo de la solución.

Tiempo de ejecución

Grasp	25 reqs	50 reqs	100 reqs
Max	17	27	43
Q3	13	22	42
Mediana	13	21	37,5
Q1	13	21	31
Min	13	16	31

Genético	25 reqs	50 reqs	100 reqs
Max	243	1361	8300
Q3	239,25	870	6610,5
Mediana	237	803	5962,5
Q1	235	743,5	4464,75
Min	220	496	3566

Colonia de Hormigas	25 reqs	50 reqs	100 reqs
Max	37	66	167
Q3	30,25	65	165
Mediana	29	62,5	163
Q1	29	61	160
Min	27	59	159

Satisfacción total

Grasp	25 reqs	50 reqs	100 reqs
Max	4999	5567	5341
Q3	4999	5434	5170,5
Mediana	4999	5406	5090
Q1	4999	5265,5	4951,25
Min	4999	5154	4813

Genético	25 reqs	50 reqs	100 reqs
Max	4999	6008	6383
Q3	4999	6008	6383
Mediana	4999	6008	6383
Q1	4999	6008	6383
Min	4999	6008	6383

Colonia de Hormigas	25 reqs	50 reqs	100 reqs
Max	4999	5369	6173
Q3	4999	5328	5984,75
Mediana	4999	5313,5	5912,5
Q1	4999	5260	5875
Min	4999	5257	5871

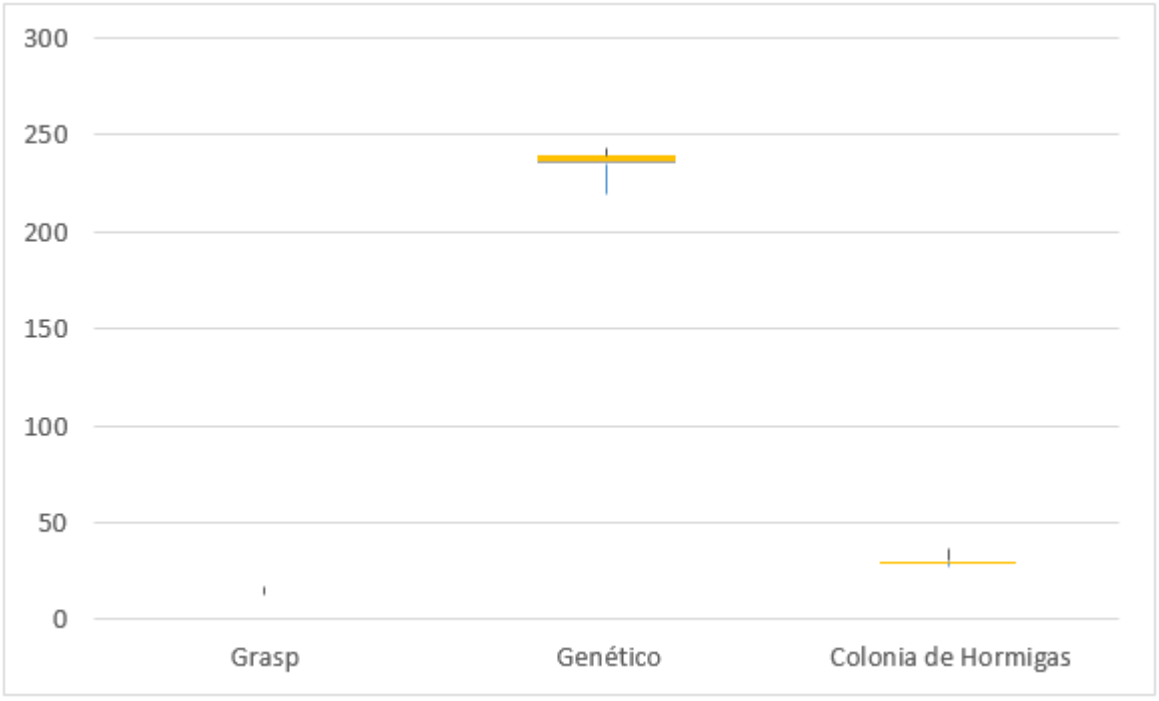
Esfuerzo total

Grasp	25 reqs	50 reqs	100 reqs
Max	127	150	150
Q3	127	150	150
Mediana	127	149	149
Q1	127	147,75	149
Min	127	144	148

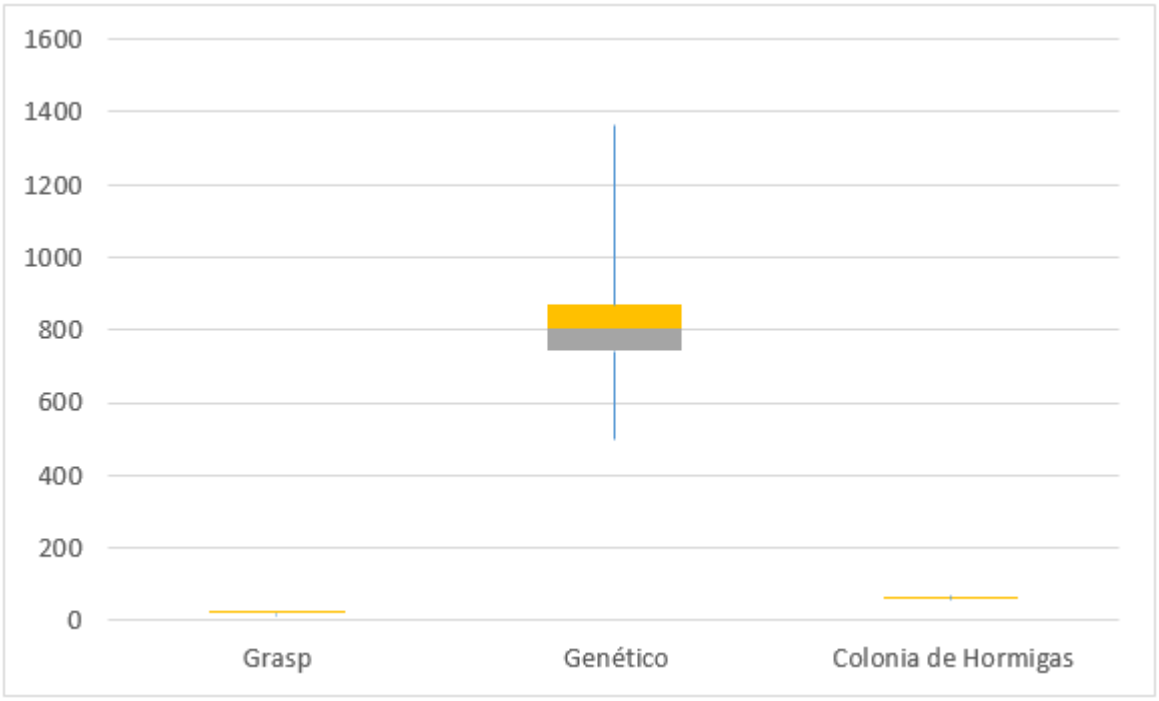
Genético	25 reqs	50 reqs	100 reqs
Max	127	150	150
Q3	127	150	150
Mediana	127	150	150
Q1	127	150	150
Min	127	150	150

Colonia de Hormigas	25 reqs	50 reqs	100 reqs
Max	127	150	150
Q3	127	149	150
Mediana	127	149	150
Q1	127	149	149
Min	127	145	149

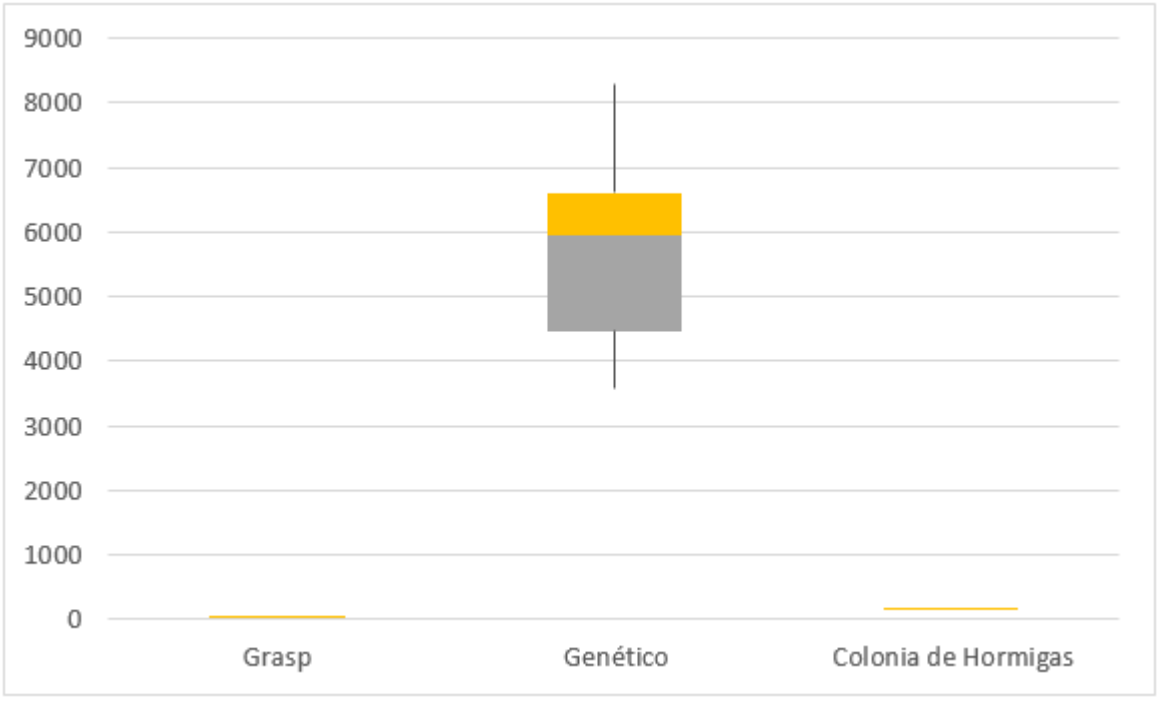
Representación del tiempo para 25 Requisitos



Representación del tiempo para 50 Requisitos

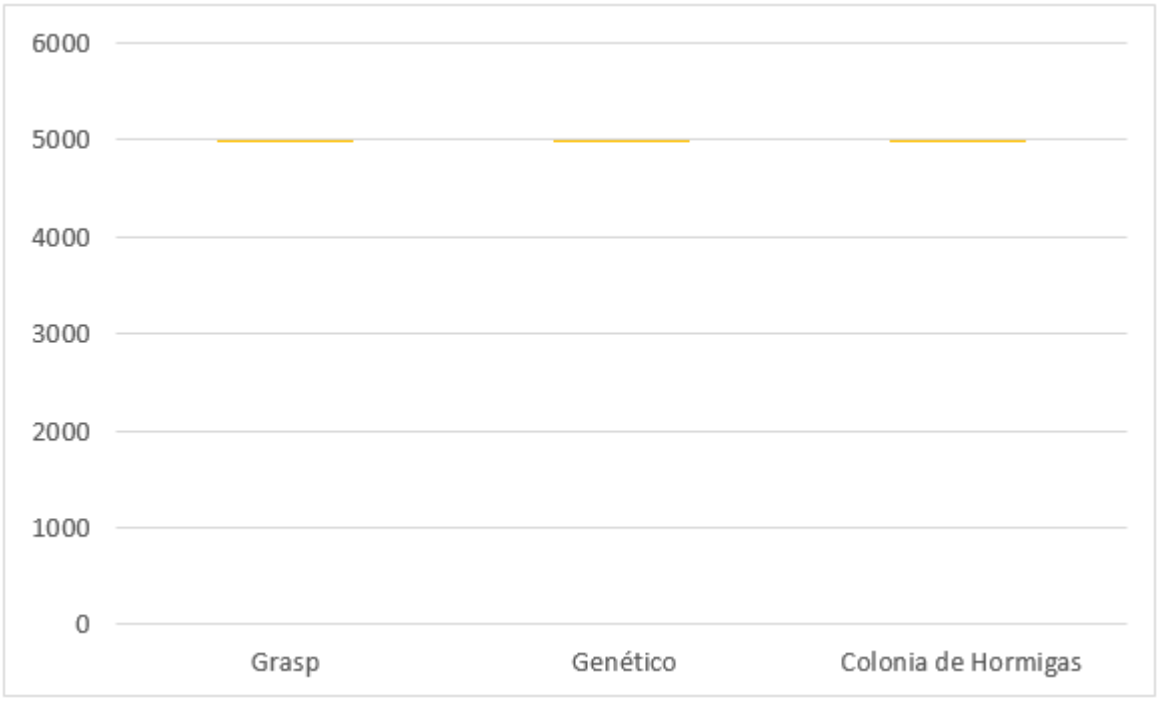


Representación del tiempo para 100 Requisitos

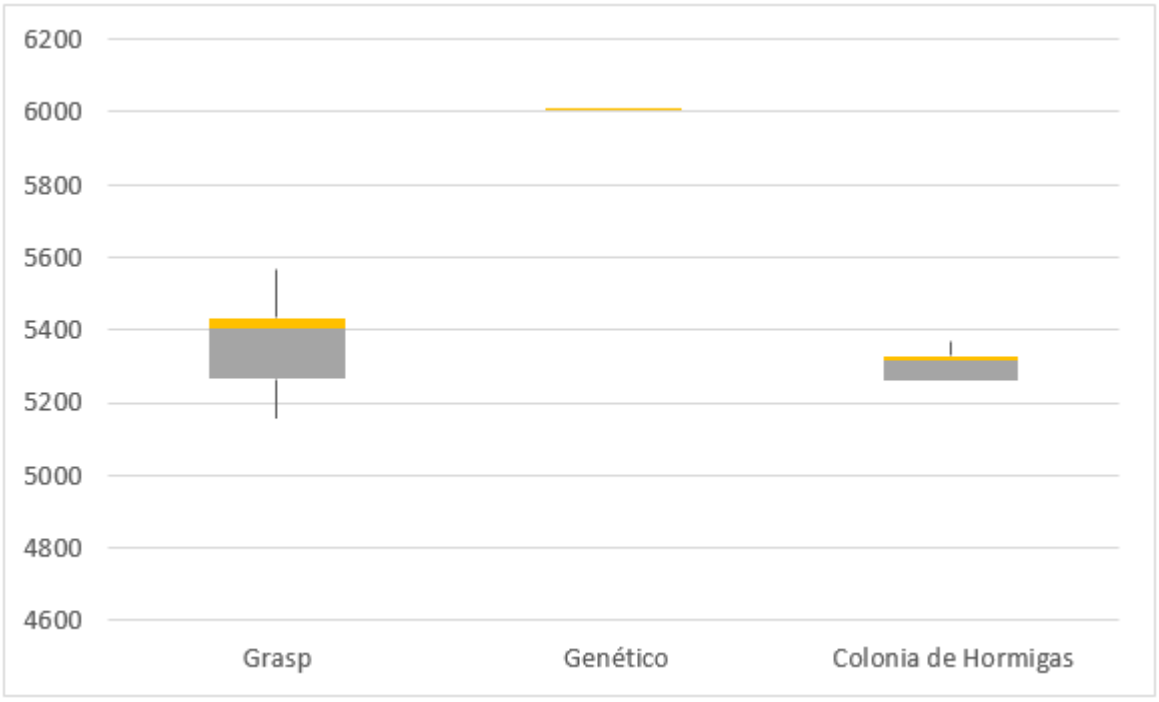


El tiempo del genético es mucho mayor y además varía mucho más, mientras que el de la colonia de hormigas es ligeramente a grasp y ambos varían muy poco de una ejecución a otra.

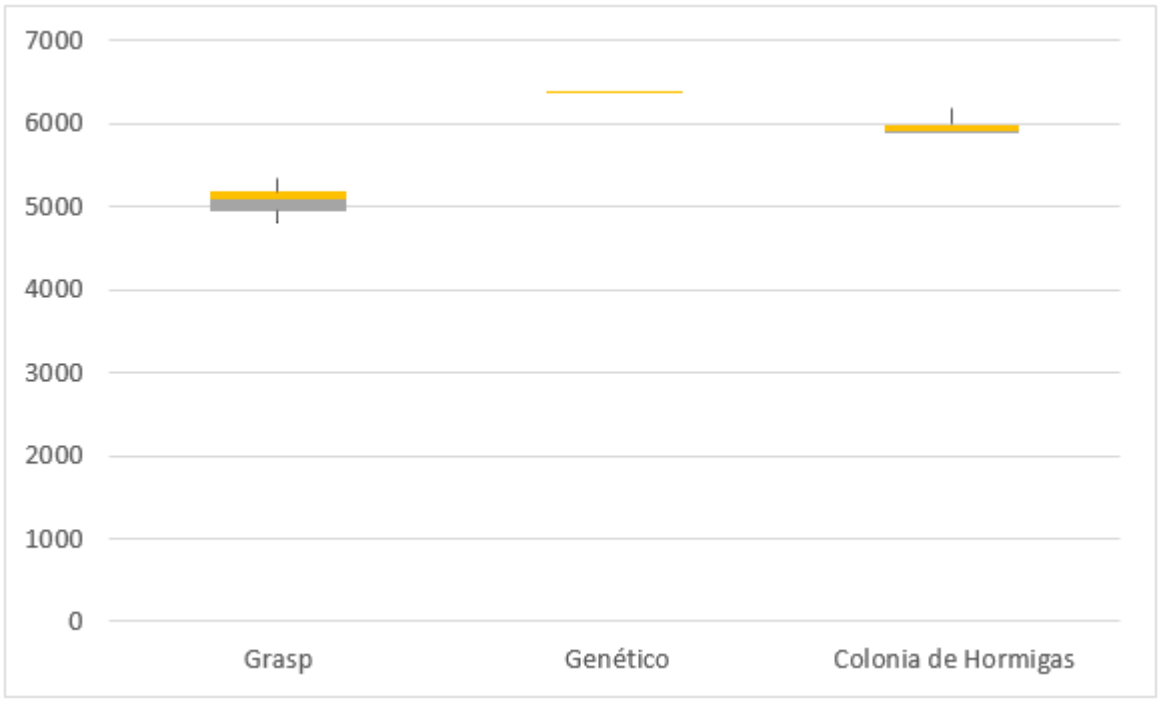
Representación de la satisfacción para 25 requisitos



Representación de la satisfacción para 50 requisitos

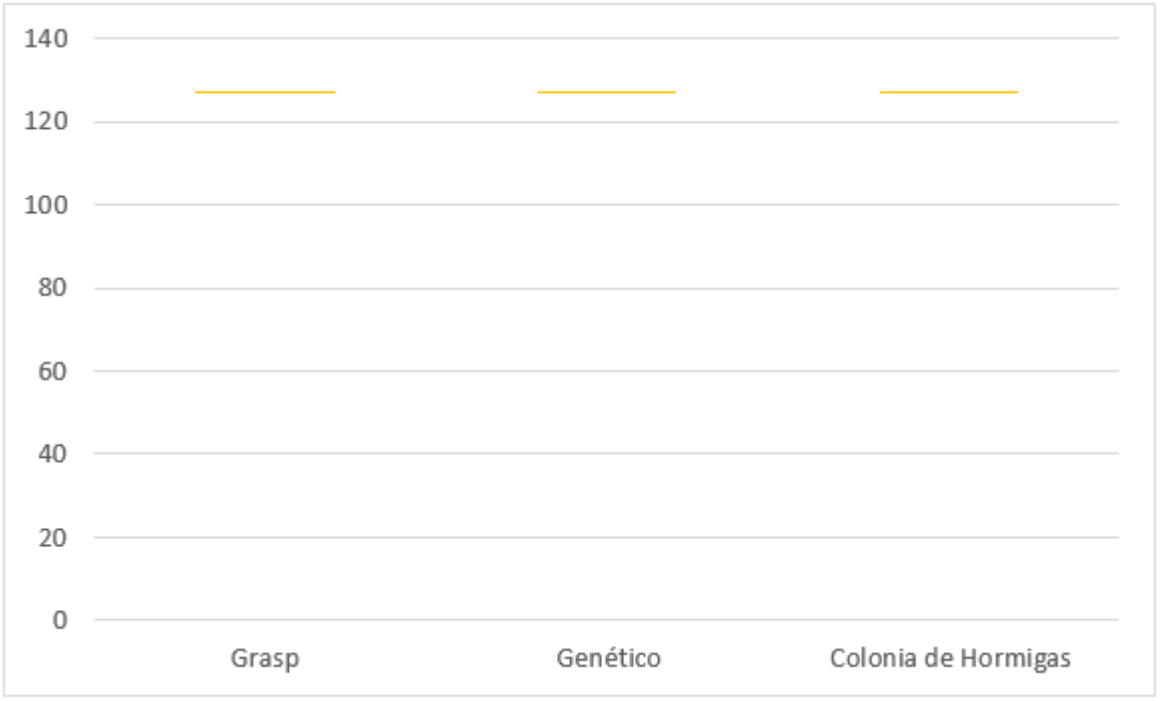


Representación de la satisfacción para 100 requisitos

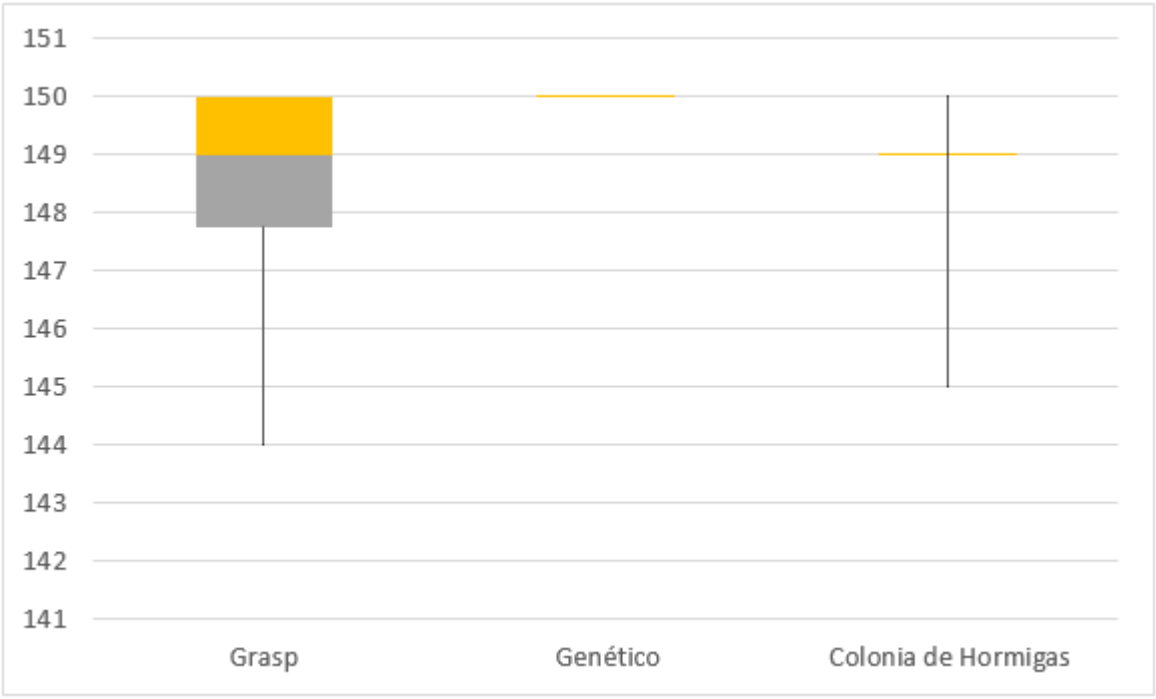


El problema de 25 requisitos es muy pequeño por lo que todas las ejecuciones de todos los algoritmos encuentran el óptimo. El algoritmo genético encuentra el óptimo en todas las ejecuciones. Dependiendo del problema grasp encuentra mejores soluciones que la colonia de hormigas o al revés, aunque la diferencia entre una ejecución y otra de grasp es mucho mayor.

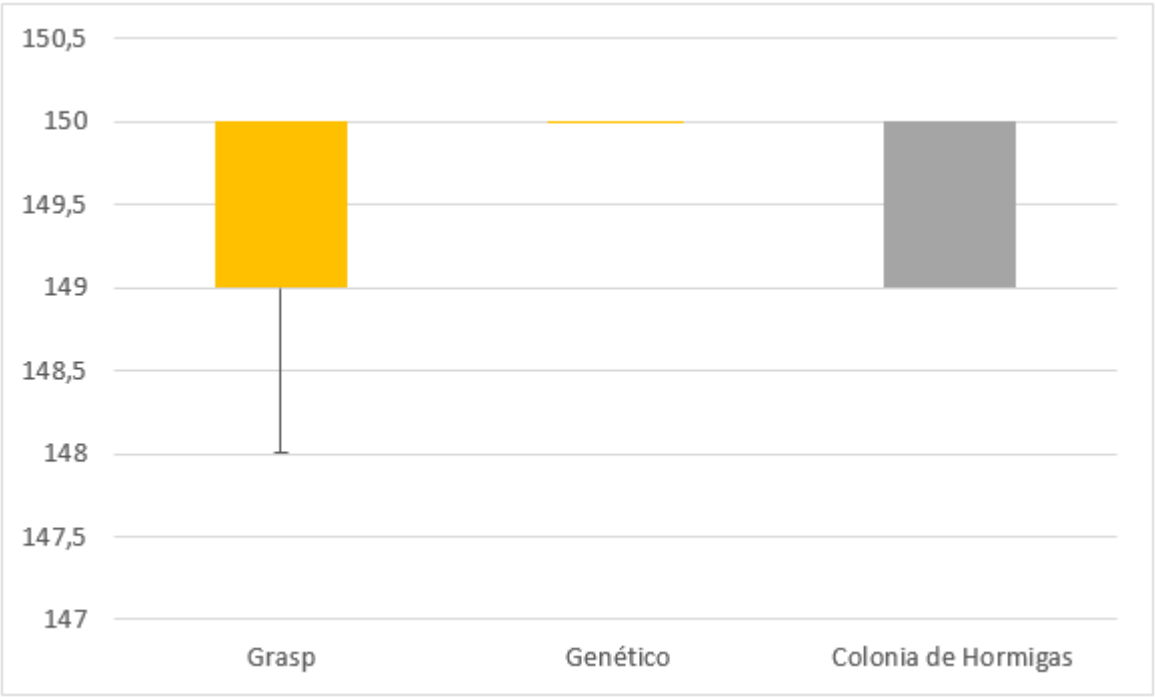
Representación del esfuerzo para 25 requisitos



Representación del esfuerzo para 50 requisitos



Representación del esfuerzo para 100 requisitos



Como hemos visto antes, todas las ejecuciones en el problema de 25 requisitos encuentran la misma solución, la cual no alcanza el techo de esfuerzo de 150 requisitos. En los otros dos problemas el esfuerzo tampoco varía mucho, teniendo un mínimo en 144 y estando la mayoría cercanos a esos 150 que hay como tope.

7. Bibliografía

[ISEP02] *Pressman, Roger s., Ingeniería del Software un enfoque práctico, 5ª ed., Mc Graw Hill, 2002.*

[SRE97] *Thayer, R. H., y M. Dorfman, Software Requirements Engineering, 2ª ed., IEEE Computer Society Press, 1997.*

[MSSR12] *Águila, Isabel María, del Sagrado, José y Orellana, F.J.: Metaheurísticas como soporte a la selección de requisitos del software. XVII Jornadas de Ingeniería del Software y de Bases de Datos, Septiembre 17-19 2012, Almería, España.*

[IANS01] *Nilsson, Nils J., Inteligencia artificial: Una nueva síntesis, 1ª ed., Mc Graw Hill, 2001.*

[IAEM04] *Russel, Stuart y Norving, Peter, Inteligencia artificial: un enfoque moderno, 2ª ed., Pearson Prentice Hall, 2004.*

[IAD15] *Mathivet, Virginie, Inteligencia artificial para desarrolladores: conceptos e implementación en C#, 1ª ed., Eni, 2015.*

[SMS12] *Dikmans, Lonneke, SOA made simple, 1ª ed., Packt Pub., 2012.*

[SIP07] *Josutties, Nicolai M., SOA in practice, 1ª ed., O'Reilly, 2007.*

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

El problema de la selección de requisitos se presenta cuando no podemos satisfacer todas las necesidades de nuestros clientes y debemos elegir que requisitos realizamos con el esfuerzo limitado del que disponemos. Este es un problema de optimización al que le hemos dado solución utilizando tres estrategias de la ingeniería del software basada en búsqueda: grasp, un algoritmo genético y un algoritmo de colonia de hormigas. Estas técnicas han sido implementadas siguiendo una arquitectura orientada a servicios, el cuál es un patrón de diseño que define servicios para solucionar las necesidades de los usuarios.

GRADO EN INGENIERÍA INFORMÁTICA, 2015/2016