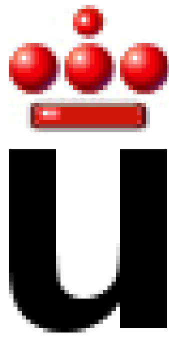

Universidad
Rey Juan Carlos

PROYECTO FIN DE CARRERA

Ingeniería Informática Superior

CURSO ACADÉMICO 2009 - 2010

Algoritmos heurísticos y metaheurísticos para el problema de localización de regeneradores.

Alumno: Carlos Rodríguez Ortiz **DNI:** 49015032p

Tutores: Abraham Duarte Muñoz

Juan José Pantrigo Fernández

Escuela Técnica Superior de Ingeniería Informática

Agradecimientos

En primer lugar quiero agradecer a mis tutores de proyecto Abraham D. y Juan José P. su trato, disponibilidad, paciencia y sobre todo el haberme enseñado tanto y haberme animado a mejorar.

A mi madre Dolores O. por cuidarme, escucharme y comprenderme, a mi padre Carlos R. por sus consejos, guiarme y protegerme y a mi hermana Ángeles R. porque eres lo que más quiero, por ser motivo de orgullo y mi ejemplo a seguir.

A mis amigos por su apoyo en los momentos difíciles, en especial a Blas Antonio R. por haber sido mi compañero de viaje estos cinco años y porque sin ti todo hubiese sido mucho más difícil.

Por último pero no menos importante quiero agradecer a todos los compañeros y colegas de la URJC que he conocido en estos años, por enseñarme, por su ayuda y por permitirme compartir mis inquietudes con vosotros.

Resumen

El objetivo de este proyecto es resolver un problema que tiene una aplicación real en el mundo de las telecomunicaciones. Es un problema de optimización en el que se pretende, mediante el uso de algoritmos, encontrar soluciones que permitan ahorrar recursos a las empresas que trabajan en el sector.

Dado un grupo de ciudades, con unas distancias entre ellas se quiere comunicar a todas con todas, de manera que cada ciudad encuentre un camino para comunicarse con cualquier otra. Este problema se modela informáticamente con una estructura de grafo, donde las ciudades son los nodos y los caminos junto con las distancias entre estas son las aristas del grafo.

Dada la atenuación de las señales eléctricas u ópticas de transmisión de información se deben utilizar aparatos que refuercen la calidad de estas, y permitan alcanzar nuevos y más lejanos destinos. Estos aparatos son los regeneradores o repetidores de la señal y deben colocarse en los nodos del grafo. Es en esta colocación es en la que se centrarán los algoritmos, con el objetivo de colocar el mínimo número posible de regeneradores, ya que son relativamente caros, para poder comunicar completamente todos los nodos entre sí.

Se han utilizado algoritmos de construcción de soluciones basados en algoritmos heurísticos los cuales construyen soluciones relativamente buenas y en ocasiones óptimas, pero en muchas ocasiones estas soluciones no son las mejores para el problema y pueden ser optimizadas. Para ello se han utilizado también algoritmos heurísticos de optimización, como la búsqueda local o el re-encaminamiento de trayectorias. Estos algoritmos reciben soluciones factibles construidas previamente e intentan mejorarlas, es decir, partiendo de ellas intentan encontrar una solución mejor.

Los algoritmos anteriormente comentados son la base para construir algoritmos metaheurísticos, como es el caso de los algoritmos GRASP (*Greedy Randomized Adaptative Search Procedures*) y GRASP-PR (*Greedy Randomized Adaptative Search Procedures with Path Relinking*). Los algoritmos metaheurísticos son más completos y complejos que los heurísticos. Utilizan a los heurísticos y consiguen explorar un rango mayor de posibles soluciones para una instancia de un problema dado.

Índice

Capítulo 1 Introducción 1
1.1.1. Grafo de comunicación 5
1.2. Aplicaciones 7
1.3. Estudio previo 8
Capítulo 2 Descripción algorítmica 11
2.1. Introducción 11
2.1.1. Algoritmos exactos 11
2.1.2. Algoritmos heurísticos 12
2.1.3. Algoritmos metaheurísticos 16
2.2. GRASP 19
2.3. Path Relinking 21
2.4. Búsqueda Local 22
2.5. Pre-proceso 23
2.5.1. Grafo de comunicación 23
2.5.2. Otras reglas 25
2.6. Construcciones 25
2.6.2. Algoritmo H1 27
2.6.3. Algoritmo H2 28
2.7. Búsqueda Local 30
2.7.1. Búsqueda local LS2x1 30
2.7.2. Búsqueda local LS1x0 31
2.8. Path Relinking 32
2.8.1. Path Relinking PR1 32
Capitulo 3 Objetivos 35
Capitulo 4 Descripción informática 37
4.1. Herramientas utilizadas 37
4.1.1. Plataforma NetBeans 37
4.1.2. Java 38
4.1.3. Dia 38
4.1.4. JXL 39
4.2. Requisitos funcionales 40

4.3. Requisitos no funcionales 40
4.4. Ciclo de vida 41
4.5. Diagrama de Gantt 43
4.6. Descripción algorítmica desde el punto de vista informático 44
4.6.1. Patrón Template 44
4.6.2. Clase grafo 45
4.6.3. Clase Instancia 47
4.6.4. Clase Solucion 49
4.6.5. Clase Constructivo 51
4.6.6. Clase AlgoritmoConstructivo1Random 53
4.6.7. Clase AlgoritmoH1Random 57
4.6.8. Clase AlgoritmoH2Random 59
4.6.9. Clase ListaAux 61
4.6.11. Clase BusquedaLocal 64
4.6.12. Clase BL2x1 65
4.6.13. Clase BL1x0 67
4.6.14. Clase BL2x1Manual 68
4.6.15. Clase PR 69
4.6.16. Clase PR1 70
4.6.17. Clase PR2 72
4.6.18. Clase Algoritmo 73
4.6.19. Clase GRASP 73
4.6.20. Clase GRASP-PR 76
4.6.21. CrearEstadisticas 79
4.6.22. Clase Main 81
Capítulo 5 Experimentos 83
5.1. Resultados 83
Capitulo 6 Conclusiones y trabajos futuros 91
6.1. Conclusiones 91
6.2. Trabajos futuros 92
Bibliografía 93
Anexo 95

Índice de figuras

Figura 1.1. Relación entre los problemas P, NP y NP – Completo. 3
Figura 1.2. Posible grafo Europeo. 4
Figura 1.3. Grafo inicial o grafo G. 4
Figura 1.4. Grafo de comunicación o grafo M. 6
Figura 1.5. Grafo solución. 7
Figura 2.1. Función multimodal. Suele acarrear el encasillamiento de los algoritmos heurísticos en óptimos locales. 15
Figura 2.2. Composición de las metaheurísticas. 17
Figura 2.3. Pseudocódigo GRASP. 19
Figura 2.4. Grafo inicial. 23
Figura 2.5. Grafo de comunicación. 24
Figura 2.6. Pseudocódigo Algoritmo Constructivo. 26
Figura 2.7. Pseudocódigo Algoritmo H1. 28
Figura 2.8. Pseudocódigo Algoritmo H2. 29
Figura 2.9. Pseudocódigo Búsqueda local dos por uno. 30
Figura 2.10. Pseudocódigo Búsqueda local uno por cero. 31
Figura 2.11. Pseudocódigo Path Relinking 1. 33
Figura 4.1. Modelo de ciclo de vida en espiral. 42
Figura 4.2. Diagrama de Gantt representando los objetivos marcados en el tiempo. 43
Figura 4.3. Jerarquía de la clase `Constructivo`. 44
Figura 4.4. Vista general de las clases del proyecto. 45
Figura 4.5. Clase `Grafo`. 45
Figura 4.6. Clase `Instancia`. 48
Figura 4.7. Clase `Solucion`. 49
Figura 4.8. Clase `Constructivo`. 52
Figura 4.9. Clase `AlgoritmoConstructivo1Random`. 53
Figura 4.10. Grafo inicial. 54
Figura 4.11. Grafo después de ubicar un regenerador en el nodo 1. 55
Figura 4.12. Grafo completo. 56
Figura 4.13. Clase `AlgoritmoH1Random`. 57
Figura 4.14. Clase AlgoritmoH2Random. 59
Figura 4.15. Grafo con aristas actualizadas después de ubicar regeneradores en los nodos 1 y 2. 60
Figura 4.16. Grafo completo con regeneradores en los nodos 1,2 y 3. 61
Figura 4.17. Clase `ListaAux`. 61
Figura 4.18. Clase `OperacionesConjuntos`. 63
Figura 4.19 Clase `BusquedaLocal`. 64
Figura 4.20. Clase `BLDosPorUno`. 65
Figura 4.21. Clase `BLUnoPorCero`. 68

Figura 4.22. Clase `BLDosPorUnoManual`. 68
Figura 4.23. Clase `PR`. 69
Figura 4.24. Clase `PR1`. 70
Figura 4.25. Clase `PR2`. 72
Figura 4.26. Clase `Algoritmo`. 73
Figura 4.27. Clase `GRASP`. 74
Figura 4.28. Clase `GRASPPR`. 76
Figura 4.29. Clase `CrearEstadisticas`. 79
Figura 4.30. Clase `Main`. 81

Capítulo 1

Introducción

El uso cada vez mayor de las comunicaciones digitales ha inspirado el desarrollo de una gran variedad de nuevas aplicaciones en los negocios y mercados de consumo. Por ejemplo, los nuevos avances en servicios como juegos on-line en tiempo real, voz sobre IP (VoIP), compartición de vídeos en Internet o Internet móvil, han sumado una gran cantidad de tráfico a las redes de telecomunicaciones. Concretamente, *Youtube*, en la actualidad sirve 100 millones de vídeos al día.

Durante los últimos años los proveedores de servicios en telecomunicaciones han construido las redes ópticas para satisfacer la creciente demanda de los usuarios. Una de las ventajas de las redes ópticas es la gran capacidad que ofrecen. Han proporcionado una cantidad de ancho de banda casi ilimitada y esto ha sido debido a que un cable de fibra óptica se compone de miles de fibras capaces de alcanzar velocidades de casi 10.000 Gbps. Además, el uso cada vez más habitual de nuevas tecnologías modernas como la multiplicación por división de onda (que utilizando diferentes longitudes de onda pueden utilizar la misma fibra para transmitir información, aumentando así su capacidad) han sido claves para conseguir este servicio tan productivo.

Ha habido una laboriosa investigación en el campo del diseño de redes ópticas, que han incluido la investigación en arquitecturas de red e infraestructura, el control, la gestión, la protección, etcétera.

En este trabajo se plantea el problema de ubicación de regeneradores y regeneración de la señal o *Regenerator Location Problem* (*RLP*), que trata de la limitación existente a la hora de transmitir señales en las redes ópticas. Esta limitación tiene que ver con la extensión geográfica que las señales ópticas recorren.

Específicamente, una señal óptica solo puede viajar una distancia máxima D_{max} antes de que se degrade su calidad, y necesite ser regenerada. Las señales ópticas recuperan su potencia y en definitiva su calidad ubicando **regeneradores** en los nodos de la red por la que viajan.

En la práctica existen tres maneras de regenerar una señal, 1R (re-amplificación), 2R (re-amplificación y re-modelación) y 3R (re-amplificación, re-estructuración y re-ajuste

temporal). Difieren en la complejidad que entrañan cada una de estas técnicas. En la 1R la señal es simplemente re-amplificada y es relativamente barato. Sin embargo, simplemente puede ser re-amplificada un número de veces determinado antes de que la señal necesite volver a ser modelada. El proceso que siguen 2R y 3R es bastante complejo. Por limitaciones tecnológicas, lo primero es transformar la señal óptica en eléctrica, operar sobre ella y volver a transformarla en una señal óptica para retransmitirla.

Los regeneradores son elementos relativamente caros, y es deseable ubicar en los nodos de la red óptica el menor número posible, siempre y cuando todos los nodos de la red de comunicación tengan al menos un camino para acceder a cualquier otro nodo.

En el *Regeneration Location Problem* se dispone de un grafo no dirigido, las distancias de los caminos existentes entre nodos y un parámetro que indica la distancia máxima D_{max}, que es la distancia que una señal puede viajar por el grafo antes de que se deteriore y deba ser regenerada para recuperar una calidad aceptable.

El problema consiste en determinar caminos que conecten todos los nodos de la red entre sí, y si es necesario, alojar regeneradores en algunos de los nodos para que este objetivo se cumpla, evitando que la señal viaje más distancia de la distancia máxima especificada sin atravesar un nodo con regenerador.

Para construir soluciones factibles se van a utilizar algoritmos *GRASP*. Este tipo de algoritmos se describen más detalladamente en otro Capítulo de la memoria, pero a modo introductorio se avanza que *GRASP* es un acrónimo que significa *Greedy Randomized Adaptative Search Procedures*, es decir, procedimientos de búsqueda miope (constructiva, voraz o ávida), aleatorizados y adaptativos. De una manera breve se puede decir que un algoritmo *GRASP* tiene dos partes fundamentales. La primera parte consiste en crear una solución al problema estudiado y la segunda en mejorar esta solución, optimizándola todo lo que sea posible.

Una vez construidas soluciones aceptablemente buenas la labor en la que centrarse será optimizarlas. Se entiende por optimizar una solución conseguir una solución mejor partiendo de una solución dada. Para ello se hará uso de las ideas descritas por los algoritmos de optimización de Búsqueda Local BL o *Local Search LS*. Estos algoritmos de tratan de alcanzar óptimos locales mediante técnicas que se describirán más adelante.

También se van a utilizar algoritmos de optimización como el re-encadenamiento de trayectorias o *PR* (*Path Relinking*). Una vez construida una solución para un problema intentan encontrar una solución mejor ejecutando una serie de métodos que caracterizan a este tipo de algoritmos y que serán explicados en Capítulos posteriores. Si estos algoritmos no son capaces de encontrar una solución más eficiente devuelven la solución que recibieron.

1.1. Problema

Matemáticamente, los problemas pueden caracterizarse atendiendo a la dificultad que entraña su resolución por un ordenador. Se han definido varias clases de problemas, entre las que destacan las clases *P, NP, NP−completo* y *NP−duro*. En la Figura 1.1 se muestra la relación entre éstos.

Se dice que un problema se puede resolver en un tiempo polinómico cuando el tiempo de ejecución de un algoritmo que lo resuelve se puede relacionar con el tamaño de la entrada con una fórmula polinómica. Los problemas para los que existe un algoritmo polinómico se denominan *P*.

Hay problemas que no se pueden resolver en un tiempo polinómico, pero dado un valor a ese problema sí se puede comprobar en tiempo polinómico si es solución o no, estos son los problemas *NP*.
No obstante, hay otro tipo de problemas, llamados *NP−completos* que no tienen un algoritmo en tiempo polinómico que los resuelva. No se ha podido demostrar formalmente que no exista, pero los matemáticos creen que realmente no existe. Este tipo de problemas es un subconjunto de los problemas *NP*. En este último grupo se ubica el *RLP*.

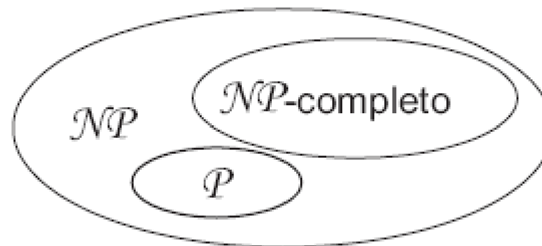

Figura 1.1. Relación entre los problemas P, NP y NP – Completo.

RLP estudia la posible mejora de determinadas áreas de las telecomunicaciones. Se trabaja con grafos no dirigidos $G=\{N,V\}$, un conjunto de nodos representado por N y un conjunto de vértices representados por V, que unen estos nodos. El objetivo es que todos los nodos puedan comunicarse con todos los demás, y si alguno no puede llegar a algún destino, proporcionar un camino que posibilite la comunicación entre ambos nodos.

Para facilitar el entendimiento de la idea es mejor pensar en los nodos del grafo como en ciudades, por ejemplo de Europa. Supóngase que se quieren conectar entre sí Madrid, París, Roma, Londres, Moscú, Berlín y Ámsterdam, nuestro grafo tendría siete nodos, es decir, N={Madrid, París, Roma, Londres, Moscú, Berlín, Ámsterdam}. En este ejemplo las aristas se podrían entender como líneas de tendido eléctrico que unen dos

ciudades. Además, las aristas tienen un número asociado que simboliza la distancia entre nodos, necesaria para valorar y actualizar la vida de la señal de transmisión eléctrica.

Figura 1.2. Posible grafo Europeo.

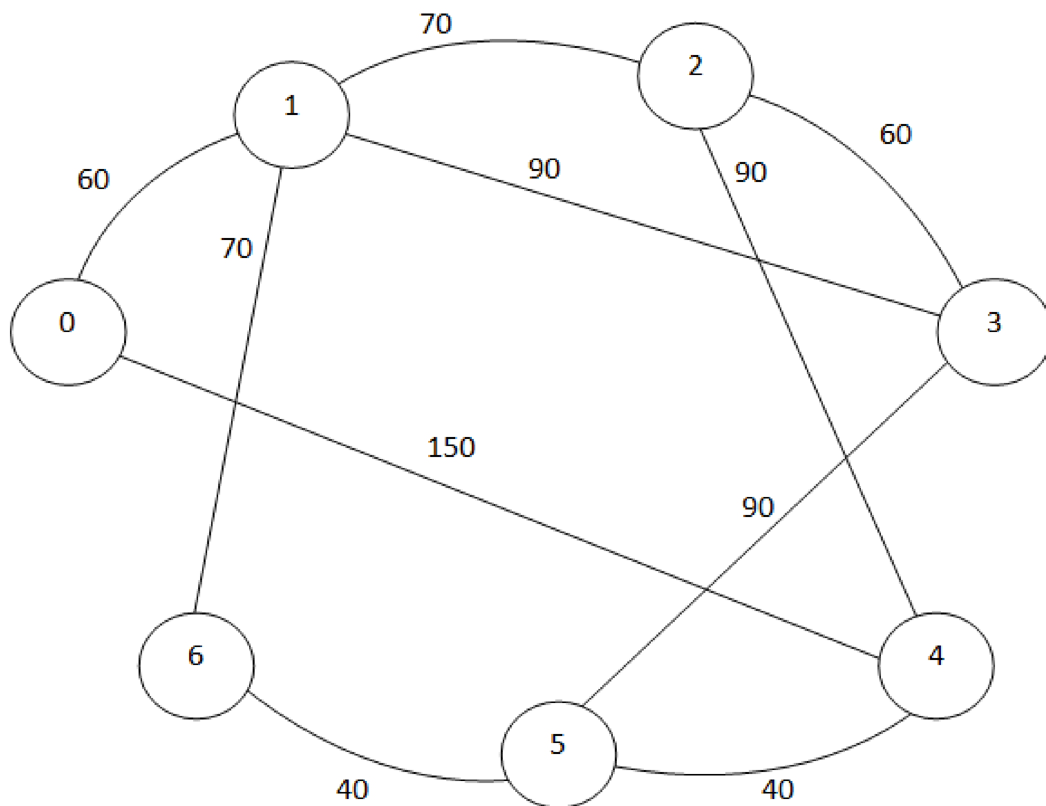

Figura 1.3. Grafo inicial o grafo G.

Esto lleva al planteamiento del problema, ya que la señal eléctrica se atenúa con la distancia. Esta distancia viene dada por un parámetro en el problema (Pueden ser 100 Km, 10Km…). Esto significa que, por ejemplo, la distancia entre Madrid y Berlín es demasiado grande para que una señal llegue del origen al destino, sin **regenerarse**.

Para solucionar el problema se dispone de unos aparatos capaces de regenerar la señal, los regeneradores o repetidores. Estos regeneradores pueden alojarse en los nodos del grafo, es decir en las ciudades del plano.

¿Qué significa esto? Supóngase como parámetro de atenuación de la señal 1500 Km. En el momento que la señal recorre esta distancia deja de ser legible, por lo tanto debe conducirse la señal a una ciudad más cercana, por ejemplo París (1260 Km) para que se reconstruya. Si en París se ha alojado un regenerador y Berlín está a menos de 1500 Km de París, se da por comunicado mediante una nueva línea eléctrica (arista en el grafo) Madrid – Berlín, Berlín – Madrid.

El problema añadido es que los regeneradores son aparatos muy caros. Nuestra labor es estudiar todas las posibilidades de ubicar estos y seleccionar la mejor, que en definitiva es la que permita conectar todos los nodos del grafo con el menor número de regeneradores, es decir, la que permita ahorrarse el mayor coste posible a las compañías encargadas de realizar esta comunicación.

1.1.1. Grafo de comunicación

El grafo que se recibe es el grafo de la Figura 1.3., denominado grafo inicial o grafo *G*, pero ese grafo no es apto para trabajar sobre él. Antes, se debe convertir en lo que denominamos un **grafo de comunicación** o grafo ***M***.

Para transformar un grafo en un grafo de comunicación hay que realizar dos operaciones esencialmente:

- Controlar la distancia máxima. Si por ejemplo la distancia máxima D_{max} que puede recorrer una señal en nuestro ejemplo son 100 u.l. (unidades de longitud) toda arista que comunique dos nodos, o línea eléctrica que comunique dos ciudades y sea superior a esta medida debe eliminarse.

- Creación de nuevos caminos sin necesidad de utilizar regeneradores. Si la distancia que se recorre de la ciudad 1 a la 2 es X, y la distancia de 2 a 3 es Y, y si X+Y es menor que la distancia máxima D_{max} que puede recorrer una señal (100 u.l. en este ejemplo), se puede crear una arista de la ciudad 1 a la 3 con coste X+Y sin necesidad de ubicar un regenerador en la ciudad 2.

Esto se aprecia mejor con los datos del ejemplo. En la Figura 1.3. la ciudad 4 está a 40 u.l de la ciudad 5, y la ciudad 5 está a 40 u.l. de la ciudad 6, por lo tanto, la ciudad 4 está comunicada con la ciudad 6 a una distancia de 80 u.l porque es menor que la distancia máxima D_{max} = 100 u.l.

Después de realizar estas dos operaciones, el grafo que se consigue es el grafo de comunicaciones, tal y como se muestra en la Figura 1.4.

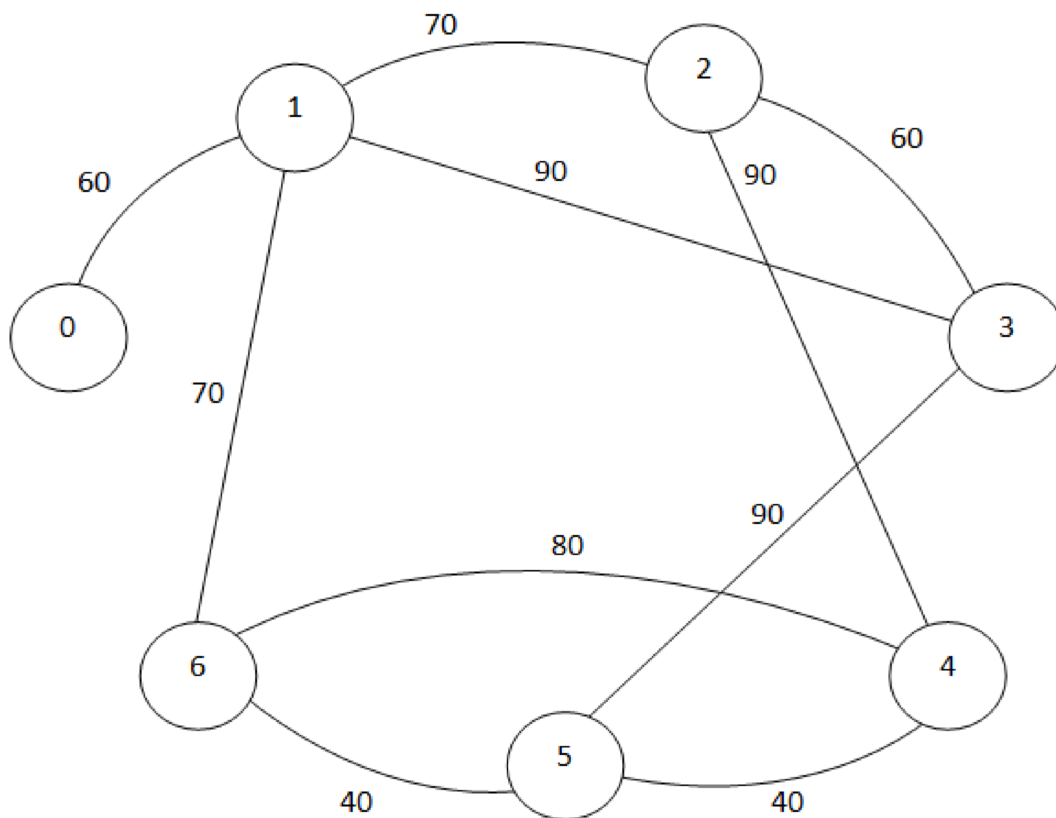

Figura 1.4. Grafo de comunicación o grafo M.

Los algoritmos desarrollados se ejecutan sobre este grafo, y desde él que se crearán las soluciones, como la que se muestra a continuación. En este caso, el primer nodo en albergar un regenerador es el nodo 1. Los nuevos caminos que se crean a consecuencia de esto se muestran en azul en la Figura 1.5. Después, el nodo 6 es el siguiente en albergar un regenerador, los nuevos caminos se muestran en rojo. Ahora el grafo *M* ya es completo, todos los nodos están comunicados con todos los demás y se puede proporcionar una solución, que entre otra información, la más valiosa es la del conjunto de nodos con regenerador, en este caso *Regeneradores*={1,6}.

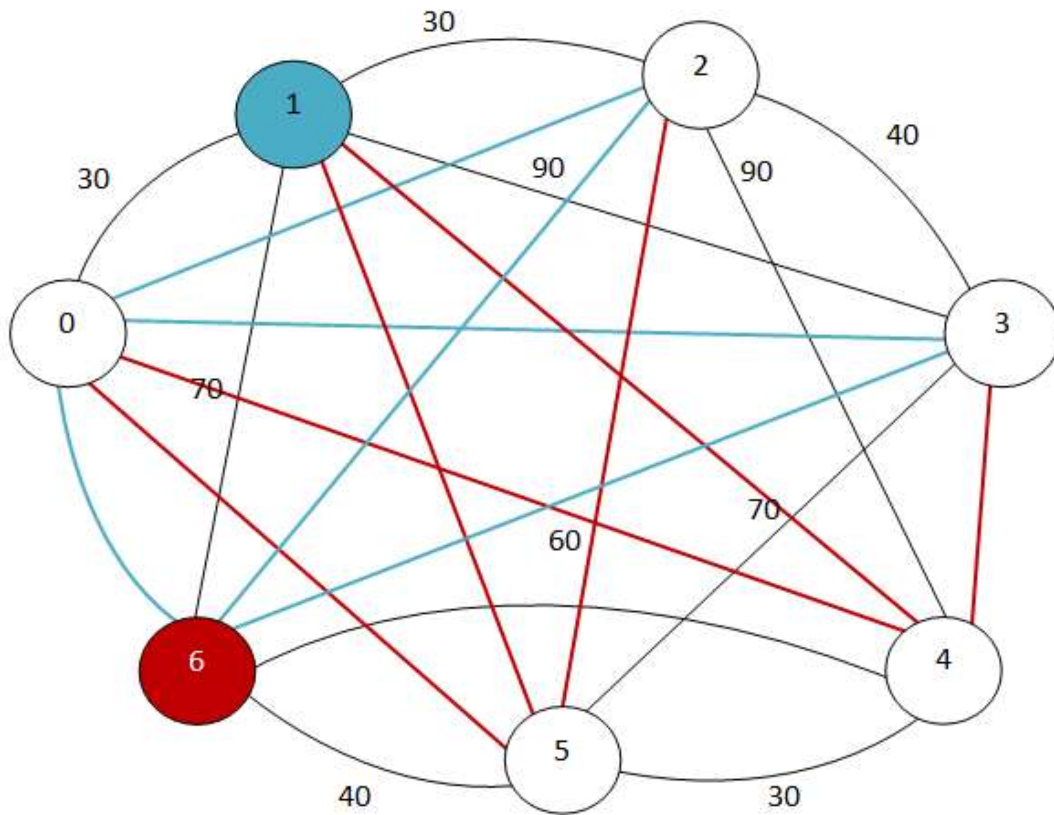

Figura 1.5. Grafo solución.

1.2. Aplicaciones

RLP es un problema perteneciente al campo de las telecomunicaciones. La resolución de *Regeneration Location Problem*, supondría un gran avance y un gran ahorro. Debido a la cada vez más creciente demanda de servicios soportados por las infraestructuras de las redes. Cada vez más servicios y cada vez más usuarios. Hasta ahora siempre se ha solventado la situación aplicando las mejoras en la tecnología y la cada vez más potencia y eficiencia de los elementos físicos de las redes, hardware con mayores prestaciones, cables capaces de transmitir la señal con mayor calidad y rapidez…pero llega un momento en que esto no es suficiente y hay que buscar otras vías de mejora.

Resolver este problema por medio de la inteligencia y la algoritmia supondría un ahorro considerable en infraestructura y un uso más eficiente de todos los elementos hardware de gran calidad de los que se dispone en la actualidad. Las compañías de las telecomunicaciones podrían ahorrar cantidades de dinero significativas en elementos hardware utilizándolos más eficientemente. Las comunicaciones seguirían siendo de la misma calidad o superior a las existentes hasta el momento. Problemas similares podrían ser abordados con mayor facilidad y garantía de obtener resultados satisfactorios.

1.3. Estudio previo

En esta Sección del Capítulo 1 se revisa el trabajo de Shi Chen y colaboradores. El primer algoritmo que proponen en su estudio es un algoritmo voraz. Tiene una serie de pasos que se repiten de manera cíclica.

1. Se selecciona el nodo con mayor número de vecinos del grafo.
2. Se actualiza el grafo ubicando un regenerador en ese nodo.
3. Si la solución es factible se devuelve y si no se vuelve al paso 1.

El segundo algoritmo es un heurístico que compone el árbol de recubrimiento del grafo. Se basa en los siguientes pasos.

1. Se selecciona el nodo con menor número de vecinos en el grafo
2. Entre todos los vecinos del nodo recibido se mira cual es el que más vecinos tiene que no hayan sido visitados y se selecciona.
3. Si el número de vecinos de ese nodo es mayor que 0 se selecciona como nodo dónde ubicar un regenerador, es decir, parte de la solución. Se vuelve al paso 2 con el nodo seleccionado como parte de la solución.
4. El algoritmo acaba cuando todos los nodos han sido visitados.

El tercer y último algoritmo es otro algoritmo heurístico que sigue los siguientes pasos.

1. Se selecciona el nodo de con menor número de vecinos del grafo.
2. Entre los vecinos del nodo seleccionado se selecciona cuál es el que tiene un mayor número de vecinos y se selecciona.
3. Se actualiza el grafo con la información correspondiente a ubicar un regenerador en ese nodo.
4. Si no es solución factible se vuelve al paso 1, si es solución se devuelve.

Después de estos algoritmos los investigadores explican la intención de construir un algoritmo de *Brach-and-Cut* o Ramificación y Corte para el *Regenerator Location Problem*. Son algoritmos de optimización combinatoria para resolver problemas enteros lineales. El método es un híbrido entre el *Branch and Bound* y *Cutting Plane*.

Por último, definen que hay tres tipos de redes, grafos o instancias que se pueden dar para el problema:

- Redes generadas aleatoriamente. En estas redes se proporciona directamente el grafo de comunicación. Esto significa que no se necesitan las distancias concretas de las aristas que conectan los nodos, simplemente se necesita saber si existe arista o no y por lo tanto, tampoco es necesario el dato de la distancia máxima.

- Redes con distancias aleatorias. Se refiere a la posibilidad de crear grafos con distancias aleatorias entre sus nodos. Estas redes necesitan ser procesadas para construir el grafo de comunicación, sobre el que se debe empezar a trabajar y no sobre el grafo inicialmente recibido, ya que el hecho de creas distancias aleatorias entre los nodos puede suponer problemas.

- Redes euclideas. Los nodos son ubicados aleatoriamente sobre una superficie de N x N. Los arcos entre los nodos son definidos matemáticamente como distancias euclideas. Para este caso se trabaja con dos parámetros, el número de nodos y la distancia máxima.

Capítulo 2.

Descripción algorítmica

En este Capítulo se presentarán los algoritmos que se han utilizado, las ideas que definen y cómo gracias a éstas se han creado algoritmos propios.

2.1. Introducción

Son muchos los algoritmos que pueden aplicarse sobre el *Regenerator Location Problem* en busca de buenos resultados. Pero el estudio ha ido evolucionando desde el uso de Algoritmos exactos hacia Heurísticos y Metaheurísticos finalmente. Más concretamente se han utilizado las ideas que describen *GRASP*, *Local Search* y *Path Relinking*.

2.1.1. Algoritmos exactos

Usando una computadora para contestar preguntas como: ¿Cuántos caminos hay para…?¿Listar todas las posibles soluciones para…?¿Hay un camino para…? usualmente requiere de una búsqueda exhaustiva dentro del conjunto de todas las soluciones potenciales. Por eso, los algoritmos que resuelven este tipo de problemas reciben el nombre de algoritmos exactos o de búsqueda exhaustiva. Por ejemplo, si se desean encontrar todos los números primos menores de 104, no hay método conocido que no requiera de alguna manera, examinar cada uno de los números enteros entre 1 y 104. De otra manera si se desea encontrar todos los caminos de un laberinto, se deben examinar todos los caminos iniciando desde la entrada.

Un ejemplo es la búsqueda con "retroceso" o (*backtracking*), trabaja tratando continuamente de extender una solución parcial. En cada etapa de la búsqueda, si una extensión de la solución parcial actual no es posible, se "va hacia atrás" para una solución parcial corta y se trata nuevamente. El método "retroceso" se usa en un amplio rango de problemas de búsqueda, incluyendo el análisis gramatical (*parsing*), juegos, y planificación (*scheduling*). La segunda técnica es "tamiz o criba" (*sieves*), es el complemento lógico de "retroceso" en que se tratan de eliminar las no-soluciones en

lugar de tratar de encontrar la solución. El método "tamiz" es útil principalmente en cálculos numéricos teóricos

Se debe tener en mente, sin embargo, que "retroceso" y "tamiz" son solamente técnicas generales. Se aplicarán en algoritmos cuyos requerimientos en tiempo son prohibitivos. En general, la velocidad de las computadoras no es práctica para una búsqueda exhaustiva de más de 100 elementos. Así, para que estas técnicas sean útiles, deben considerar solamente una estructura dentro de la cual se aproxima el problema. La estructura debe ser hecha a medida, a menudo con gran ingenio, para cuadrar con el problema particular, de modo que el algoritmo resultante será de uso práctico.

Los métodos exactos de resolución de problemas se han aplicado con éxito a una cantidad elevada de problemas. Algunos ejemplos de estos métodos son los algoritmos voraces, algoritmos de divide y vencerás, algoritmos de ramificación y poda, *backtraking*, etc. Todos estos procedimientos resuelven problemas que pertenecen a la clase *P* de forma óptima y en tiempo razonable.

Como se ha comentado anteriormente, existe una clase de problemas, denominada *NP*, con gran interés práctico para los cuales no se conocen algoritmos exactos con tiempos de convergencia en tiempo polinómico. Es decir, aunque existe un algoritmo que encuentra la solución exacta al problema, tardaría tanto tiempo en encontrarla que lo hace completamente inaplicable. Además, un algoritmo exacto es completamente dependiente del problema (o familia de problemas) que resuelve, de forma que cuando se cambia el problema se tiene que diseñar un nuevo algoritmo exacto y demostrar su optimalidad.

2.1.2. Algoritmos heurísticos

Para la mayoría de problemas de interés no existe un algoritmo exacto con complejidad polinómica que encuentre la solución óptima a dicho problema. Además, la cardinalidad del espacio de búsqueda de estos problemas suele ser muy grande, lo cual hace inviable el uso de algoritmos exactos ya que la cantidad de tiempo que necesitaría para encontrar una solución es inaceptable. Debido a estos dos motivos, se necesita utilizar algoritmos aproximados o heurísticos que permitan obtener una solución de calidad en un tiempo razonable. El término heurística proviene del vocablo griego *heuriskein*, que puede traducirse como encontrar, descubrir o hallar.

Desde un punto de vista científico, el término heurística se debe al matemático George Polya quien lo empleó por primera vez en su libro *How to solve it*. Con éste término, Polya englobaba las reglas con las que los humanos gestionan el conocimiento común y que, a grandes rasgos, se podían simplificar en:

- Buscar un problema parecido que ya haya sido resuelto.
- Determinar la técnica empleada para su resolución así como la solución obtenida.
- En el caso en el que sea posible, utilizar la técnica y solución descrita en el punto anterior para resolver el problema planteado.

Existen dos interpretaciones posibles para el término heurística. La primera de ellas concibe las heurísticas como un procedimiento para resolver problemas. Para esta interpretación, las definiciones más interesantes que se extraen de la literatura son las siguientes:

- T. Nicholson: “Procedimiento para resolver problemas por medio de un método intuitivo en el que la estructura del problema puede interpretarse y explotarse inteligentemente para obtener una solución razonable”.

- H. Müller-Merbach: “En investigación operativa, el término heurístico normalmente se entiende en el sentido de un algoritmo iterativo que no converge hacia la solución (óptima o factible) del problema”.

- Barr et al.: “Un método heurístico es un conjunto bien conocido de pasos para identificar rápidamente una solución de alta calidad para un problema dado”.

La segunda interpretación de heurística entiende que éstas son una función que permiten evaluar la bondad de un movimiento, estado, elemento o solución. Algunas de las definiciones más interesantes que se encuentran en la literatura son:

- Rich et al.: “Una función heurística es una correspondencia entre las descripciones de los estados del problema hacia alguna medida de idoneidad, normalmente representada por números. Los aspectos del problema que se consideran, cómo se evalúan estos aspectos y los pesos que se dan a los aspectos individuales, se eligen de forma que el valor que la función da a un nodo del proceso de búsqueda sea una estimación tan buena como sea posible para ver si ese nodo pertenece a la ruta que conduce a la solución”.

- Russell et al.: “Actualmente, el término heurística se utiliza más bien como adjetivo para referirse a cualquier técnica que permita mejorar el desempeño del caso promedio en una tarea de resolución de problemas, aunque no necesariamente permita mejorar el desempeño del peor de los casos. Específicamente en el área de los algoritmos de búsqueda, se refiere a una función mediante la cual se obtiene una estimación del coste de una solución”.

De todas las definiciones, quizá las más intuitivas son las siguientes:

- Zanakis et al: "Procedimientos simples a menudo basados en el sentido común que se supone que obtendrán una buena solución (no necesariamente óptima) a problemas difíciles de un modo sencillo y rápido".

- Pearl: "considerándola como una regla del pulgar que se utiliza para guiar una acción".

Existen métodos heurísticos (también llamados algoritmos aproximados, procedimientos inexactos, algoritmos basados en el conocimiento o simplemente heurísticas) de diversa naturaleza, por lo que su clasificación es bastante complicada. Se sugiere la siguiente clasificación:

1. **Métodos constructivos:** Procedimientos que son capaces de construir una solución a un problema dado. La forma de construir la solución depende fuertemente de la *estrategia* seguida. Las estrategias más comunes son:

    - *Estrategia voraz:* Partiendo de una semilla, se va construyendo paso a paso una solución factible. En cada paso se añade un elemento constituyente de dicha solución, que se caracteriza por ser el que produce una mejora más elevada en la solución parcial para ese paso concreto. Este tipo de algoritmos se dice que tienen una visión *"miope"* ya que eligen la mejor opción actual sin que les importe que ocurrirá en el futuro.

    - *Estrategia de descomposición:* Se divide sistemáticamente el problema en subproblemas más pequeños. Este proceso se repite (generalmente de forma recursiva) hasta que se tenga un tamaño de problema en el que la solución a dicho subproblema es trivial. Después el algoritmo combina las soluciones obtenidas hasta que se tenga la solución al problema original. Los algoritmos más representativos de los métodos de descomposición son los algoritmos de divide y vencerás tanto en su versión exacta como aproximada.

    - *Métodos de reducción:* Identifican características que contienen las soluciones buenas conocidas y se asume que la solución óptima también las tendrá. De esta forma se puede reducir drásticamente el espacio de búsqueda.

    - *Métodos de manipulación del modelo:* Consisten en simplificar el modelo del problema original para obtener una solución al problema simplificado. A partir de esta solución aproximada, se extrapola la solución al problema original. Entre estos métodos se pueden destacar: la linealización, la agrupación de variables, introducción de nuevas restricciones, etc.

2. **Métodos de búsqueda:** Parten de una solución factible dada y a partir de ella intentan mejorarla. Algunos son:

- *Estrategia de búsqueda local 1:* Parte de una solución factible que la mejora progresivamente. Para ello examina su vecindad y selecciona el primer movimiento que produce una mejora en la solución actual (*first improvement*)

- *Estrategia de búsqueda local 2:* Parte de una solución factible que la mejora progresivamente. Para ello examina su vecindad y todos los posibles movimientos seleccionando el mejor movimiento de todos los posibles, es decir aquel que produzca un incremento (en el caso de maximización) más elevado en la función objetivo (*best improvement*).

- *Estrategia aleatorizada:* Para una solución factible dada y una vecindad asociada a esa solución, se seleccionan aleatoriamente soluciones vecinas de esa vecindad.

El principal problema que presentan los algoritmos heurísticos es su incapacidad para escapar de los óptimos locales. En la Figura 2.1 se muestra como para una vecindad dada el algoritmo heurístico basado en un método búsqueda local se quedaría atrapado en un máximo local. En general, ninguno de los métodos constructivos descritos en la sección anterior tendrían por qué construir la solución optima.

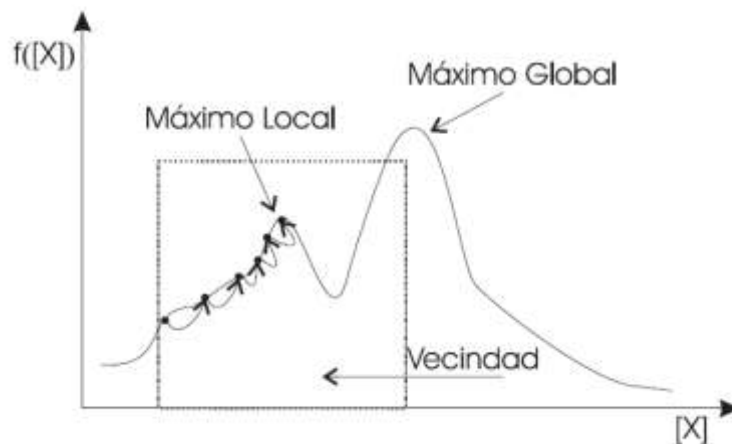

Figura 2.1. Función multimodal. Suele acarrear el encasillamiento de los algoritmos heurísticos en óptimos locales.

Los algoritmos heurísticos no poseen ningún mecanismo que les permita escapar de los óptimos locales. Para solventar este problema se introducen otros algoritmos de búsqueda más inteligentes que eviten en la medida de lo posible quedar atrapados en óptimos locales. Estos algoritmos de búsqueda más inteligentes, denominados metaheurísticas, son procedimientos de alto nivel que guían a algoritmos heurísticos conocidos evitando que éstos caigan en óptimos locales.

2.1.3. Algoritmos metaheurísticos

El término metaheurística o meta-heurística fue acuñado por F. Glover en el año 1986. Con este término, pretendía definir un "procedimiento maestro de alto nivel que guía y modifica otras heurísticas para explorar soluciones más allá de la simple optimalidad local".

Actualmente, existe una cantidad muy importante de trabajos científicos publicados que abordan problemas de optimización a través de las metaheurísticas, investigaciones sobre nuevas metaheurísticas o extensiones de las metaheurísticas ya conocidas. Existen bastantes foros donde se publican todos estos trabajos de investigación.

A partir de la definición original de F. Glover, en la literatura se pueden encontrar otras definiciones alternativas de metaheurísticas o heurísticas modernas, entre las que se pueden destacar las siguientes:

- J.P. Kelly et al.: "Las metaheurísticas son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los procedimientos estadísticos".

- S. Voss et al.: "Una metaheurística es un proceso iterativo maestro que guía y modifica las operaciones de una heurística subordinada para producir eficientemente soluciones de alta calidad. Las metaheurísticas pueden manipular una única solución completa (o incompleta) o una colección de soluciones en cada iteración. La heurística subordinada puede ser un procedimiento de alto (o bajo) nivel, una búsqueda local, o un método constructivo".

La idea básica general es siempre la misma: enriquecer a los algoritmos heurísticos de forma que éstos no se queden atrapados en óptimos locales.

La evolución de las metaheurísticas durante los últimos 25 años ha tenido un comportamiento prácticamente exponencial. En el tiempo que transcurre desde las primeras reticencias (por su supuesta falta de rigor científico) hasta la actualidad, se han encontrado soluciones de muy alta calidad a problemas que hace tiempo parecían inabordables.

De modo general, se puede decir que las metaheurísticas combinan ideas que provienen de cuatro campos de investigación bien distintos, ver Figura 2.2:

- Las técnicas de diseño de algoritmos (resuelven una colección de problemas).

- Algoritmos específicos (dependientes del problema que se quiere resolver).
- Fuente de inspiración (del mundo real).
- Métodos estadísticos.

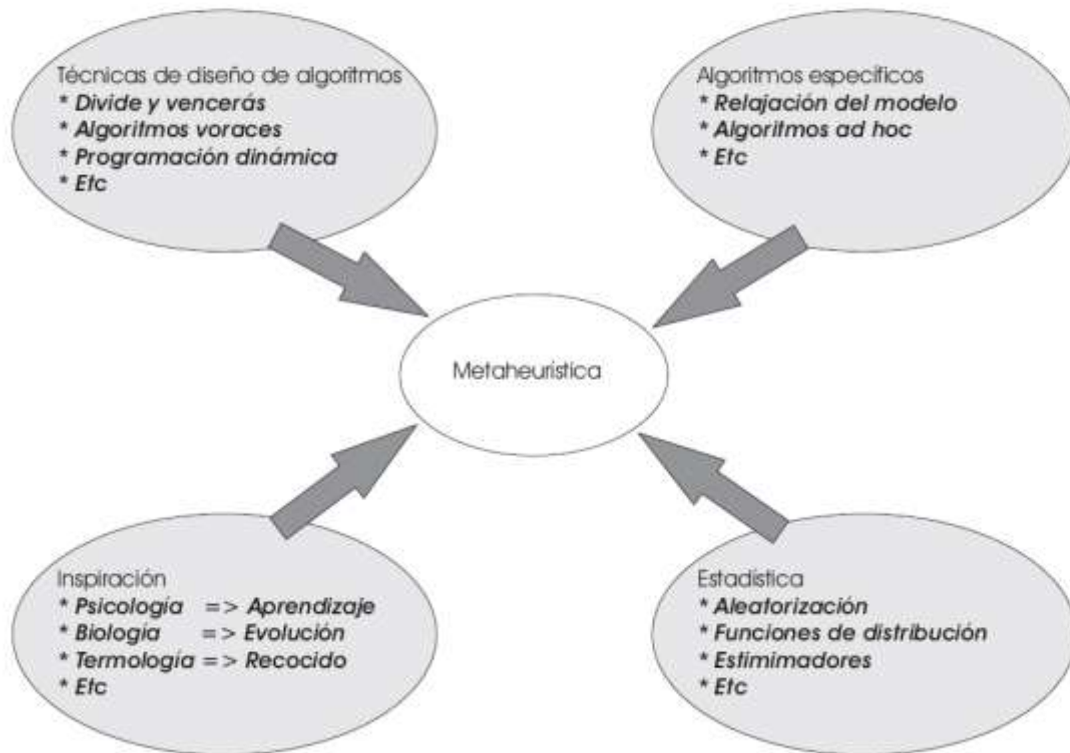

Figura 2.2. Composición de las metaheurísticas.

Una primera conclusión que se puede extraer de las definiciones dadas es que, en muchos casos, son reglas de sentido común que permiten hacer una búsqueda "inteligente". Debido a esta característica, para bastantes metaheurísticas no existe un marco teórico que las sustente, sino que es a través de los buenos resultados experimentales donde encuentran su justificación.

Definir un marco general en el que definir las metaheurísticas resulta un poco complicado hoy en día aunque se está estudiando la manera de englobarlas a todas. Por el momento se pueden clasificar de la siguiente manera:

- **Atendiendo a la Inspiración:**
  Natural: algoritmos que se basan en un símil real, ya sea biológico, social, cultural, etc.
  Sin inspiración: algoritmos que se obtienen directamente de sus propiedades matemáticas.

- **Atendiendo al número de soluciones:**
  Poblacionales: buscan el óptimo de un problema a través de un conjunto de soluciones.

Trayectoriales: trabajan exclusivamente con una solución que mejoran iterativamente.

- **Atendiendo a la función objetivo:**
  Estáticas: no hacen ninguna modificación sobre la función objetivo del problema.
  Dinámicas: modifican la función objetivo durante la búsqueda.

- **Atendiendo a la vecindad:**
  Una vecindad: durante la búsqueda utilizan exclusivamente una estructura de vecindad.
  Varias vecindades: durante la búsqueda modifican la estructura de la vecindad.

- **Atendiendo al uso de memoria:**
  Sin memoria: se basan exclusivamente en el estado anterior.
  Con memoria: utilizan una estructura de memoria para recordar la historia pasada.

Cualquiera de las alternativas descritas por sí solas no es de grano suficientemente fino como para permitir una separación clara entre todas las metaheurísticas. Generalmente, estas características (pueden incluirse más) se suelen combinar para permitir una clasificación más elaborada.

Según el teorema *NFL* (*No Free Lucnh Theorem*), que demuestra que al mismo tiempo que una metaheurística es muy eficiente para una colección de problemas, es muy ineficiente para otra colección), los métodos generales de búsqueda, entre los que se encuentran las metaheurísticas, se comportan exactamente igual cuando se promedian sobre todas las funciones objetivo posibles, de tal forma que si un algoritmo *A* es más eficiente que un algoritmo *B* en un conjunto de problemas, debe existir otro conjunto de problemas de igual tamaño para los que el algoritmo *B* sea más eficiente que el *A*. Esta aseveración establece que, en media, ninguna metaheurística (algoritmos genéticos, búsqueda dispersa, búsqueda tabú, etc.) es mejor que la búsqueda completamente aleatoria. Una segunda característica que presentan las metaheurísticas es que existen pocas pruebas sobre su convergencia hacia un óptimo global; es decir, que a *priori* no se puede asegurar ni que la metaheurística converja ni la calidad de la solución obtenida. Por último, las metaheurísticas más optimizadas son demasiado dependientes del problema o al menos necesitan tener un elevado conocimiento heurístico del problema. Esto hace que, en general, se pierda la genericidad original con la que fueron concebidas.

A pesar de estos aparentes problemas, la realidad es que el comportamiento experimental de la mayoría de las metaheurísticas es extraordinario, convirtiéndose para muchos problemas difíciles de resolver en la única alternativa factible para encontrar una solución de calidad en un tiempo razonable. En general, las metaheurísticas se comportan como métodos muy robustos y eficientes que se pueden aplicar con relativa facilidad a una colección amplia de problemas. Además, la demostración del teorema

NFL se basa en que el algoritmo de búsqueda no visita dos veces la misma solución y en que no se introduce conocimiento heurístico en el diseño del método metaheurístico. Estas hipótesis habitualmente no son ciertas.

2.2. GRASP

El nombre de esta metaheurística viene de su acrónimo en inglés *Greedy Randomized Adaptive Search Procedure* (*GRASP*), que en castellano se podría traducir como *procedimientos de búsqueda miope (constructiva, voraz o ávida), aleatorizados y adaptativos*. Cada uno de los términos incluidos en el nombre se corresponde con una característica distintiva de la metaheurística. *GRASP* se basa en el siguiente principio de operación:

GRASP es un procedimiento multi-arranque en el que cada arranque se corresponde con una iteración. Cada iteración tiene dos fases bien diferenciadas: la fase de construcción, que se encarga de obtener una solución factible de alta calidad; la fase de mejora, que se basa en la optimización (local) *de la solución obtenida en la primera fase*.

Originalmente *GRASP* fue desarrollado por T. Feo y M. Resende como un algoritmo para resolver problemas de *cubrimiento de conjuntos*. No fue hasta el año 1995 cuando adquiere una terminología y forma definitiva como metaheurística de propósito general.

Los orígenes algorítmicos de *GRASP* provienen de la *heurística semi-constructiva* (*semi-constructive heuristic*). Esta técnica también se caracteriza por ser un método multi-arranque basado en una construcción miope aleatorizada. La diferencia fundamental con respecto a *GRASP* es que la primera técnica no utilizaba un procedimiento de mejora (búsqueda local).

Figura 2.3. Pseudocódigo GRASP.

{x:TipoSolucion} = GRASP(NUMREP: Entero, F: TipoFuncionObjetivo)
variables
    i:Entero
    X_{aux}: TipoSolucion
inicio
    para i = 1 **hasta** NUMREP **hacer**
        {X_{aux} } = ConstrucciónAleatorizadaMiope();
        {X_{aux} } =MejorarSolucion(X_{aux},F)
        {x} = ActualizarSolucion(F,x, X_{aux})
    fin para
fin

GRASP es un procedimiento multi-arranque, donde cada iteración está compuesta de dos fases: una *fase constructiva* y una *fase de mejora*. En la Figura 2.3 se muestra un pseudo-código de alto nivel para esta metaheurística. La fase constructiva es un procedimiento iterativo encargado de construir una solución elemento a elemento. Inicialmente, se parte de una semilla que es una componente o un conjunto de componentes que determinan una solución parcial. Las componentes introducidas en ella se señalan como elementos no seleccionables. El resto de componentes constituyen el *conjunto de los elementos seleccionables*. Posteriormente, se ordenan todos los elementos seleccionables, utilizando para ello una función *voraz, constructiva o miope* (*greedy*), que les asigna un coste relacionado con el cambio que se produce en la función objetivo si se introduce cada uno de los elementos en la solución parcial. Esta fase del algoritmo es la que le aporta el nombre *Greedy*.

Una vez que se tienen ordenados todos los elementos seleccionables, se plantea el problema de elegir uno *"bueno"*. En el contexto de *GRASP* no se selecciona el mejor candidato posible, ya que esta opción no asegura que se obtenga una solución óptima, sino que se elige aleatoriamente un candidato de un conjunto de buenos candidatos.
Este conjunto recibe el nombre de *lista de candidatos restringida o RCL* (*Restricted Candidate List*). Numéricamente, la RCL se construye utilizando los valores máximo y mínimo del coste asignado a los elementos seleccionables en una iteración dada. Si se supone que c_{max} y c_{min} son respectivamente los valores más alto y más bajo del coste, la RCL estaría formada por todos aquellos elementos cuyo coste superase (para problemas de maximización) el umbral dado por la siguiente expresión:

$$RCL_{umbral} = (C_{min} + \alpha (C_{max} - C_{min}))$$

Donde el parámetro α : <0 $\leq \alpha \leq 1$ determina el tamaño de la RCL. Si $\alpha = 1$ sólo estaría el mejor candidato (función miope pura). Por contra, si $\alpha = 0$, estarían todos los candidatos (función aleatoria pura), esta fase es la que aporta la palabra *Randomized*.

Una vez que se ha seleccionado un candidato perteneciente a la RCL, éste se introduce en la solución parcial y se marca como elemento no seleccionable. El resto de elementos siguen siendo seleccionables; por consiguiente, para ellos se calcula de nuevo (mediante la función miope) la variación que se produciría en la función objetivo si se seleccionase cada uno de los elementos. Por lo tanto, según se van introduciendo candidatos a la solución parcial GRASP, ésta se va adaptando al nuevo escenario. A esta fase del procedimiento se debe el calificativo de *Adaptive*.

La solución construida en la primera fase no tiene por qué ser un óptimo local, ya que existen bastantes elecciones estocásticas. En otras palabras, la fase constructiva no garantiza una optimalidad de la solución. Para resolver este problema, GRASP introduce una segunda fase, conocida como fase de mejora, que consiste en un procedimiento de optimización local basado en una función de búsqueda local o,

incluso, en una metaheurística. Por lo general, esta fase mejora la solución construida pero tampoco garantiza la optimalidad de la solución (aunque experimentalmente se ha comprobado que mejora bastante la calidad). La segunda fase termina cuando no se pueda hacer ningún movimiento que mejore la solución actual.

2.3. Path Relinking

El método re-encadenamiento de trayectorias o *Path Relinking – PR*, fue originalmente propuesto como una estrategia que integra procesos de intensificación y diversificación. Es un método que está en desarrollo y su propuesta es muy reciente. El principio de operación que sigue *PR* es el siguiente:

PR genera nuevas soluciones mediante la exploración de trayectorias en el espacio de soluciones. Para ello, partiendo de dos soluciones de alta calidad, una actuando como origen y la otra como destino, se genera un camino en el espacio de soluciones entre origen y el destino. El método de combinación en *PR* se basa en la generación de trayectorias entre soluciones en el espacio de búsqueda, en lugar de llevar a cabo combinaciones lineales entre ellas. En otras palabras, dadas dos buenas soluciones x_1 y x_2, *PR* construye un camino entre ambas, comenzando por la solución x_1, denominada "solución de partida" y, llevando a cabo una serie de movimientos, intenta llegar a la solución x_2, denominada "solución guía". *PR* construye la trayectoria $x_1 \rightarrow x_2$ eliminando paulatinamente los atributos de la solución de partida para introducir atributos que pertenecen a la solución guía. El objetivo es encontrar en estos caminos soluciones que mejoren a aquéllas que originaron la trayectoria.

PR comparte los cinco procedimientos generales con la búsqueda dispersa o (*Scatter Search* - *SS*):

- **Método de generación de soluciones diversas (*Diversification Generation Method*):** genera una población de *N* soluciones diversas (típicamente *N* = 100). Cada una de estas soluciones se optimiza utilizando el método de mejora. A continuación, se extrae un subconjunto pequeño de buenas soluciones (usualmente alrededor de 10).

- **Método de mejora (*ImprovementMethod*):** transforma las soluciones de prueba en una o más soluciones de prueba mejoradas. En PR se suele aplicar el procedimiento de mejora únicamente a un subconjunto de tamaño *NumImp* de las soluciones generadas. Usualmente se utiliza un procedimiento de Búsqueda Local, aunque se puede utilizar una metaheurística.

- **Método de actualización del conjunto de referencia (*RefSet Update Method*):** construye y mantiene el conjunto de referencia (*RefSet*), compuesto por las *b*

mejores soluciones encontradas hasta el momento. A partir del conjunto de soluciones iniciales se extrae el *RefSet* siguiendo un criterio de convivencia entre soluciones de calidad y diversidad. Éstas son ordenadas de mayor a menor calidad.

- **Método de generación de subconjuntos (*Subset Generation Method*):** PR se basa en la generación de trayectorias entre conjuntos (típicamente parejas) de soluciones de calidad. La generación de estas trayectorias es aplicado a todos y cada uno de los subconjuntos generados.

- **Método de combinación de soluciones (*Solution CombinationMethod*):** combina las soluciones de los subconjuntos generados en una o más soluciones. Para parejas, el método de combinación de PR elige dos soluciones del *Ref- Set* $X_{inicial}$ y X_{guia} y construye un camino entre ambas, comenzando por $X_{inicial}$ y llevando a cabo una serie de movimientos (que generan nuevas soluciones) para llegar a la solución X_{guia}. Este procedimiento se implementa de forma distinta si las trayectorias se generan a partir de grupos de soluciones de mayor tamaño.

2.4. Búsqueda Local

El método de búsqueda local (*Local Search, LS*) recorre el conjunto de elementos seleccionados buscando el mejor intercambio para reemplazar un elemento seleccionado con un elemento no seleccionado. El método realiza movimientos siempre que el valor de la función objetivo aumente. Finaliza su ejecución cuando no se encuentra ningún intercambio de elementos que mejore. Este método de mejora se clasifica en los métodos que seleccionan el movimiento que más mejora la solución (*best improvement*).

Este método es, además, la base de muchos de los algoritmos utilizados en problemas de optimización. El algoritmo recibe una solución inicial y trata de hallar entre su vecindad un candidato que garantice una mejor solución, para esto el concepto de vecindad es clave. Se considera vecindad a todas las posibles soluciones que están próximas a la solución dada.
La técnica de selección en la vecindad es conocida como la regla del pivoteo o *pivoting rule*, en general puede ser o *best-improvement rule* (explicado al inicio de este apartado) o *first-improvement rule*, que selecciona el primer vecino que mejora la solución en lugar de al mejor entre todos.

2.5. Pre-proceso

Hasta este momento se han introducido algunas de las principales características y cualidades que han servido para abordar nuestro problema de forma general, pero más concretamente hay algunas reglas o ideas básicas que se han de tener en cuenta antes de proporcionar una solución a un escenario del *RLP*.

2.5.1. Grafo de comunicación

El escenario que se recibe debe cumplir una serie de condiciones antes de trabajar sobre él. El grafo de entrada o grafo *G* recibido debe ser revisado, comprobando que es un escenario válido. Si esto se cumple se habrá conseguido el grafo de comunicación.

Para todos los escenarios ha de conocerse un parámetro de vital importancia. Es el parámetro que indica a partir de qué distancia la señal deja de tener calidad suficiente y debe ser regenerada. Este parámetro al que se le llama distancia máxima o D_{max} será el primer elemento para conseguir el grafo de comunicación o grafo *M*.

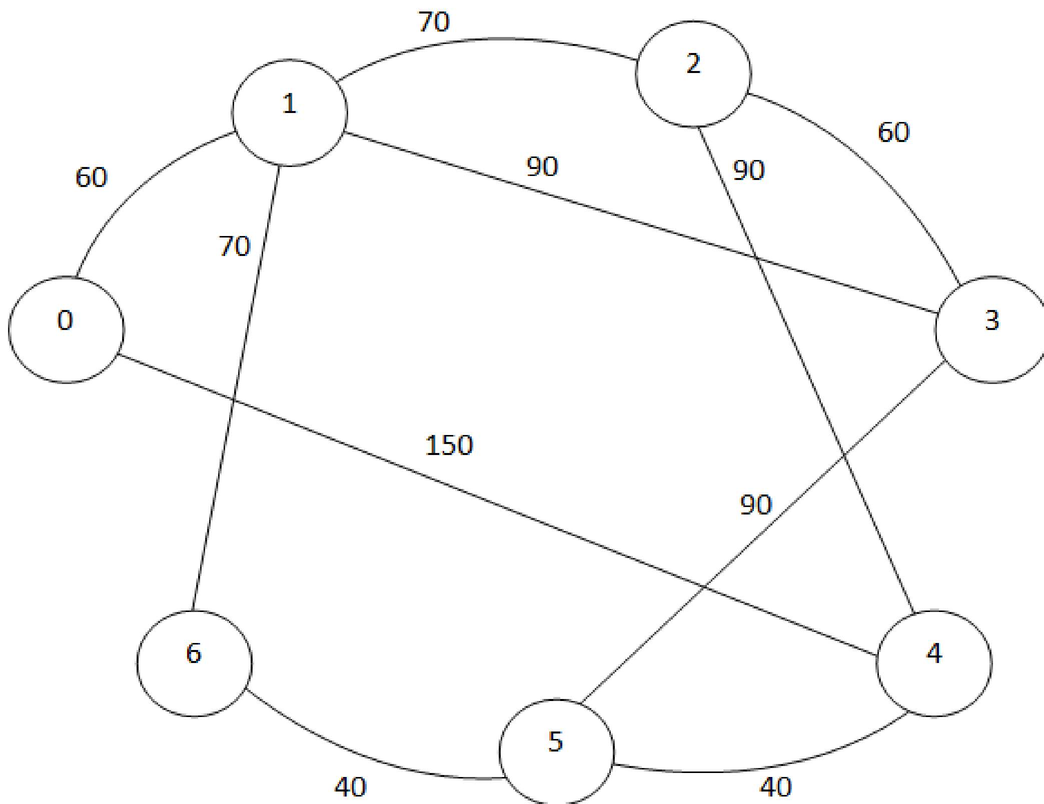

Figura 2.4. Grafo inicial.

Ningún camino que sea de una distancia superior a D_{max} es válido, porque una señal emitida desde el nodo origen nunca podrá llegar al destino con la suficiente potencia y calidad, por lo que debe ser eliminado. Utilizando la Figura 2.4 como ejemplo se observa que la arista que une los nodos 0 y 4 tiene un valor de 150, como es superior a la distancia máxima permitida en el ejemplo, $D_{max} = 100$, la arista se elimina.

El segundo y último paso, después de eliminar las aristas inconsistentes, es comprobar si se pueden generar nuevos caminos entre nodos sin necesidad de utilizar regeneradores. Esta comprobación consiste en verificar si se puede llegar de un nodo ‘A’ a un nodo ‘C’, pasando por un nodo ‘B’, sin que se rompa la regla de que la distancia de ‘A’-‘B’ más la distancia de ‘B’-‘C’ sea mayor que D_{max} .Lo que es lo mismo que:

$$D_{A-B} + D_{B-C} < D_{max}$$

Si esta condición es cierta, el coste de la nueva arista entre ‘A’ y ‘C’ sería de $D_{A-B} + D_{B-C}$. En la Figura 2.5 existe este caso entre los nodos 4 y 6, porque la distancia que la señal debe recorrer desde 4 hasta 5 más la distancia que se debe recorrer desde 5 hasta 6 no es mayor que la distancia máxima del ejemplo. Lo que es lo mismo que:

$$40 + 40 < 100$$

Por lo tanto se puede crear un nuevo arco entre 4 y 6, con coste 80. Con estas dos operaciones se consigue el grafo de comunicación M, que para nuestro ejemplo quedaría como se muestra en la Figura 2.5.

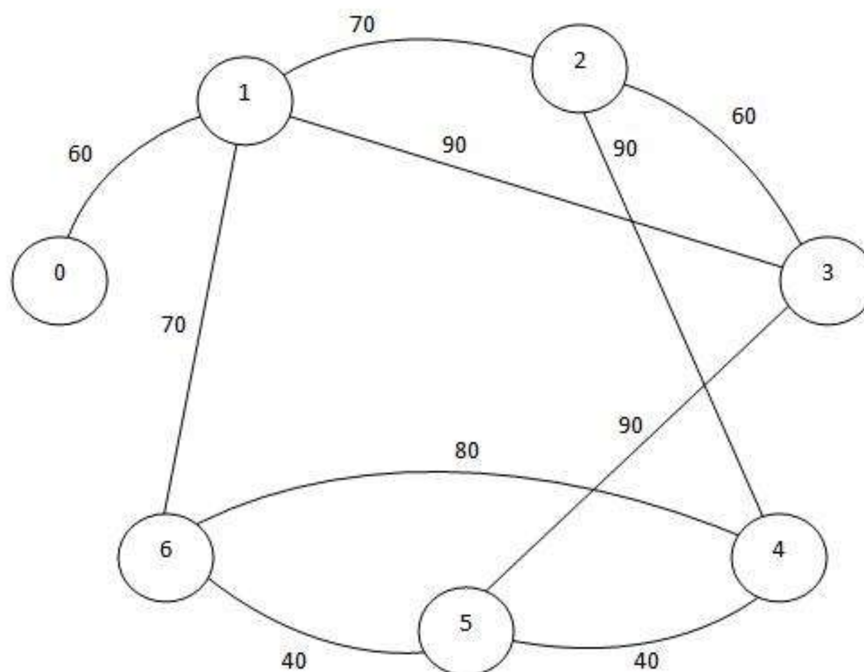

Figura 2.5. Grafo de comunicación.

2.5.2. Otras reglas

Además de la creación del grafo de comunicación existen otras reglas de pre-proceso que la tipología de los grafos nos permite deducir. Está demostrado que en todo grafo de comunicación o grafo *M* sobre el que se pueden ejecutar algoritmos heurísticos, metaheurísticos y de optimización no se debe permitir alojar un regenerador en los nodos que únicamente tengan un vecino. Esto es debido a que ubicar un regenerador en un nodo con un solo vecino no proporcionaría mejora, ya que sería incapaz de unir a su único vecino con algún otro nodo. Por esta misma razón, su único vecino es obligatorio que albergue un regenerador, porque es a través de este último por el que podemos acceder a ese nodo tan aislado.

Apoyándose en la Figura 2.5 del grafo de comunicación podemos ver el ejemplo en el nodo 0 y en el 1. Ubicar un regenerador en el nodo 0 no reporta ningún beneficio ya que no se crea ningún camino nuevo. Mientras que si no ubicamos un regenerador en el nodo 1 ninguno de los demás nodos del grafo podría acceder al nodo 0, se puede concluir que utilizar el nodo 1 para colocar un regenerador resulta obligatorio si queremos conseguir una solución factible.

2.6. Construcciones

Se ha decidido implementar tres algoritmos capaces de construir soluciones factibles y en muchas ocasiones óptimas, dos que optimicen las soluciones construidas atendiendo al método de Búsqueda Local o *Local Search* y otros dos algoritmos atendiendo al método de Re-encadenamiento de Trayectorias o *Path Relinking*.

2.6.1. Algoritmo Constructivo

Está basado en la idea de ubicar regeneradores en los nodos que más vecinos tengan. Es un algoritmo aleatorizado, lo que significa que no siempre se seleccionará como candidato al nodo de mayor grado[1], sino que se creará una lista de candidatos y entre todos ellos y de forma aleatoria se selecciona al nodo elegido para ubicar un regenerador.

El algoritmo actúa de forma cíclica hasta que consigue crear una solución factible. En primer lugar siempre se explora cada uno de los nodos del grafo con el fin de crear la lista de candidatos (*Candidated List* - *CL*). La lista de candidatos guarda cómo de bueno es un candidato en función de un criterio de evaluación. En este algoritmo el criterio de evaluación es el grado, los nodos con mayor grado serán considerados como nodos más óptimos para ubicar un regenerador.

[1] El grado de un nodo es el número de vecinos de este

Una vez se dispone de la lista de candidatos se calcula el parámetro *th*, que indicará que nodos son lo suficientemente buenos para ser considerados (no elegimos al mejor nodo, sino a uno de forma aleatoria entre los mejores) candidatos a alojar un regenerador. Es gracias a este parámetro con el que decidimos qué nodos formarán la lista de candidatos restringida (*Restricted Candidated List-RCL*). De esta lista es de donde se sacará aleatoriamente el nodo que albergue el regenerador. Este parámetro se calcula con una fórmula matemática.

$$th = \min\big(eval(e)\big) + \alpha\,(\max\big(eval(e)\big) - \min(eval(e))\,)$$

Donde min(*eval*(*e*)) es el valor mínimo de todos los nodos estudiados en la iteración. max(*eval*(*e*)) es el valor máximo de los nodos estudiados en la iteración. Y α es el parámetro que define la aleatoriedad o la voracidad del algoritmo, que varía entre 0 y 1 y es fijado según el comportamiento deseado. Si α es 0 el algoritmo es completamente aleatorio, porque el valor de corte siempre será el valor mínimo del nodo menos favorecido, y todos pasarán el corte formando parte de la RCL. Si α es 1, el algoritmo es completamente voraz (*Greedy*) porque el parámetro obligará a que se elija el nodo con el máximo valor de esa iteración.

Figura 2.6. Pseudocódigo Algoritmo Constructivo.

```
{x:TipoSolucion} = Constructivo()
variables
        i,numeroNodos,min,max,nodo:Entero
        th,α:Real
        Solucion: TipoSolucion
        CL,RCL : Vector
inicio
        mientras NoFactible(Solucion) do
                para i = 1 hasta numeroNodos hacer
                        CL[i] = grado(i)
                        min = esMinimo(grado(i))
                        max = esMaximo(grado(i))
                fin para
                th=min + α (max - min)
                para i=1 hasta CL do
                        si CL[i]>=th entonces
                                añadir(RCL,i)
                        fin si
                fin para
                nodo = seleccionarAleatoriamente(RCL)
                actualizarSolucion(nodo)
        fin mientras
        devolver Solucion
fin
```

2.6.2. Algoritmo H1

El algoritmo heurístico 1 o *H1* construye una solución a partir de la idea de crear el árbol de recubrimiento del grafo. La idea es que una vez construido el árbol de recubrimiento del grafo, todos los nodos que sean internos en el árbol alojarán un regenerador. Esto garantiza que las soluciones devueltas por *H1* serán factibles.

H1 también es un algoritmo aleatorizado, al igual que ocurría con el algoritmo anterior, se sigue el mismo procedimiento, creando la lista de candidatos en función de un criterio de evaluación de nodos. En este algoritmo, el criterio de evaluación es el grado de un nodo, respecto al conjunto de nodos que quedan por visitar para el algoritmo. Es decir, si un nodo tiene grado 3 porque tiene 3 vecinos, pero sólo dos de ellos están en el conjunto de nodos no visitados, ese nodo tiene un grado con valor de 2 para el algoritmo.

Una vez calculada la lista de candidatos *CL*, se calcula el parámetro de aleatoriedad *th* para crear la lista restringida de candidatos *RCL*, seleccionando un nodo de ella y repitiendo la operación.

Este algoritmo funciona de forma recursiva. Para iniciarlo se debe buscar entre todos los nodos del grafo aquél con menor grado. Este nodo será la raíz del árbol. La función recursiva se llama con este nodo inicial, el conjunto de nodos visitados (ya contiene el nodo inicial), el conjunto de nodos no visitados (conjunto complementario al conjunto anterior) y el conjunto solución. El procedimiento recursivo calcula los nodos vecinos del nodo recibido que no han sido visitados todavía (hace la intersección con el conjunto de nodos no visitados), de este conjunto saldrá la lista de candidatos, la lista de candidatos restringida y el nodo seleccionado con el que repetir el procedimiento.

Después de calcular ese conjunto de nodos el algoritmo actualiza los conjuntos de nodos visitados y no visitados, selecciona el nodo candidato de la manera explicada anteriormente y se repite la recursividad o se vuelve de ella.

Figura 2.7. Pseudocódigo Algoritmo H1.

```
{R:Vector} = ArbolAleatorio(u: Entero,S:Vector,SBarra:Vector,R:Vector)
variables
        i,min,max,nodo:Entero
        th,α:Real
        U, CL, RCL: Vector
Inicio
        U(u) = Vecinos(u) ∩ SBarra
        si noPrimeraVez() entonces
                S = S ∪ U(u) ∪ u
                SBarra = V \ S
        fin si
        mientras U(u) ≠ ∅ do
                para i = 1 hasta U(u) hacer
                        CL[i] = grado_SBarra (i)
                        min = esMinimo(grado_SBarra (i))
                        max = esMaximo(grado_SBarra (i))
                fin para
                th=min + α (max - min)
                para i=1 hasta CL do
                        si CL[i]>=th entonces
                                añadir(RCL,i)
                        fin si
                fin para
                nodo = seleccionarAleatoriamente(RCL)
                si grado_SBarra (nodo)) entonces
                        R = R ∪ nodo
                        ArbolAleatorio(nodo,S,SBarra,R)
                fin si
                U(u) = U(u) \ nodo
        fin mientras
        devolver
fin
```

2.6.3. Algoritmo H2

El algoritmo *H2* es el último algoritmo constructivo implementado. Es un algoritmo heurístico que se basa en una idea que mezcla las dos anteriores. También es un algoritmo aleatorizado, siguiendo el mismo método que los dos anteriores, utilizando las listas de candidatos *CL* y candidatos restringidos *RCL*.

Es un algoritmo iterativo que no termina hasta que ha construido una solución factible. *H2* empieza seleccionando el nodo de menor grado entre todos los nodos del grafo. Una vez lo tenemos seleccionado se calcula el conjunto de sus vecinos, de todos y cada uno de los nodos vecinos se evalúa su grado y entre todos los vecinos que formen parte de la lista de candidatos restringida se selecciona el nodo que albergará un regenerador. Este proceso se repite hasta llegar a una solución factible.
En este caso, el criterio de evaluación para construir la lista de candidatos restringida RCL es el grado de los nodos vecinos del nodo menor seleccionado en primera instancia.

Figura 2.8. Pseudocódigo Algoritmo H2.

```
{x:TipoSolucion} = AlgoritmoH2()
variables
        i,numeroNodos,min,max,nodo:Entero
        Solucion: TipoSolucion
        CL,RCL : Vector
Inicio
        mientras NoFactible(Solucion) do
                para i = 1 hasta numeroNodos hacer
                        nodoMenor = Min(vecinos(i))
                fin para
                conjuntoVecinos = vecinos(nodoMenor)
                para i = 1 hasta conjuntoVecinos hacer
                        CL[i] = grado(i)
                        min = esMinimo(grado(i))
                        max = esMaximo(grado(i))
                fin para
                th=min + α (max - min)
                para i=1 hasta CL do
                        si CL[i]>=th entonces
                                añadir(RCL,i)
                        fin si
                fin para
                nodo = seleccionarAleatoriamente(RCL)
                actualizarSolucion(nodo)
        fin mientras
fin
```

2.7. Búsqueda Local

Los algoritmos de búsqueda local son algoritmos de optimización, es decir, que dado una solución inicial, intentan mejorarla lo máximo posible. Si no lo consigue devuelve la solución tal y como la recibió. Se han implementado dos algoritmos de búsqueda local.

2.7.1. Búsqueda local LS2x1

El algoritmo de búsqueda local *LS2x1* intenta reducir el número de regeneradores de la solución que recibe. La idea es intercambiar dos de los regeneradores del conjunto solución que hay alojados en nodos del grafo, por sólo uno alojándolo en cualquier nodo del grafo, manteniendo la solución factible. Esto no siempre puede ocurrir, lógicamente. En ese caso, la solución se deja como se recibe.

El método elige dos de los nodos con regenerador a sustituir. Esta elección se puede realizar de forma aleatoria o probando todos los pares posibles para agotar todas las posibilidades. Una vez seleccionados los dos nodos a sustituir, se evalúan todos los vecinos de estos, y se comprueba si existe algún nodo, que alojando un regenerador pueda conectarlos a todos entre sí.

Figura 2.9. Pseudocódigo Búsqueda local dos por uno.

```
{x:TipoSolucion} = DosPorUno(solución:TipoSolucion)
variables
        RPrima,C : Vector
        i,j: Entero
Inicio
        RPrima = R \ {i,j}
        C = V \ RPrima
        para u ∈ (vecinos(i) ∪ vecinos(j)) hacer
                si vecinos(u) ∩ RPrima = ∅ entonces
                        C = C ∩ vecinos(u)
                fin si
                si C = ∅ entonces
                        devuelve
                fin si
        Fin para
        Seleccionar v ∈ C
        RPrima = RPrima ∪ v
        Devuelve RPrima
fin
```

El pseudocódigo muestra cómo se consigue implementar la idea anteriormente mencionada sin hacer un algoritmo de fuerza bruta. El conjunto *RPrima* es un vector auxiliar dónde se copiará el vector *R* (es el conjunto solución, contiene los nodos con regeneradores) que viene dado en la solución. Es con el vector *RPrima* con el que se trabajará en el código. El vector *C* es el conjunto de nodos que pueden albergar un regenerador, del que finalmente y si no está vacío, se saca uno de los nodos que será el que albergue el nuevo regenerador en detrimento de los regeneradores del nodo *i* y *j*.
Para todos los vecinos de los nodos *i* y *j*, el nodo llamado *u* en el pseudocódigo se comprueba si sus vecinos no tienen ningún elemento en común con el conjunto solución y si es así, se refina el conjunto de candidatos donde ubicar un regenerador. Dejando este conjunto solo con los nodos comunes a esos vecinos de *u* y a los nodos que ya tenía el conjunto. Después de todas las iteraciones se consigue encontrar el nodo que pueda conectar a todos esos vecinos de *i* y *j* que quedaron desconectados.

2.7.2. Búsqueda local LS1x0

El algoritmo de optimización *LS1x0*, como su nombre indica, intenta eliminar un regenerador del conjunto de nodos manteniendo la solución como una solución factible. Este algoritmo tiene sentido y además funciona bastante bien porque las soluciones construidas por los algoritmos de la sección anterior son aleatorios. De manera que puede existir una solución con cuatro nodos con regeneradores, cuando a lo mejor simplemente con dos de esos cuatro nodos se consigue una solución factible.

La idea es intentar quitar de una solución dada un regenerador de un nodo, y comprobar si con los regeneradores restantes la solución es factible. Si se optimiza la solución termina y devuelve la solución con la mejora encontrada. Si no, se intenta probar quitando el regenerador de otro nodo.

Figura 2.10. Pseudocódigo Búsqueda local uno por cero.

{x:TipoSolucion} = BLUnoPorCero(solución:TipoSolucion)
variables
    i:Entero
    completo : Booleana = falso
    solucionAux : TipoSolucion
Inicio
    mientras i<numeroRegeneradores(solucion) **y NO**completo **do**
        solucionAux = solución
        solucionAux = solucionAux – i
        completo = EsFactible(solucionAux)
    fin mientras
fin

2.8. Path Relinking

Los algoritmos de optimización *Path Relinking* o Re-encadenamiento de Trayectorias tratan de explorar un espacio mayor de soluciones mediante la transformación de una solución en otra. En este camino de transformación se espera descubrir una solución intermedia mejor que las dos soluciones de partida. En base a esta idea se han construido dos algoritmos.

2.8.1. Path Relinking PR1

El algoritmo recibe dos soluciones y se parte de una de las dos como solución origen. La transformación consiste en operar sobre el conjunto de nodos con regenerador que tiene la solución y transformarla en la segunda solución recibida o solución destino.

Se va transformando la solución origen (o solución de partida) en la solución destino (o solución guía) en pasos sucesivos. De la solución origen se selecciona un nodo a eliminar (que no esté en la solución destino) del conjunto solución de nodos que alojan regeneradores y se selecciona un nodo a insertar (que no esté en la solución origen) del conjunto de nodos que albergan regeneradores de la solución destino. Después de esta transformación se comprueba si la solución es factible. Si no es así, se completa la solución con más nodos hasta dejar una solución completa. Una vez realizada esta comprobación y con una solución factible se intenta mejorar mediante los métodos de búsqueda local anteriormente mencionados, si se consigue mejorar se para la búsqueda y si no se repite el proceso, intentando convertir la solución construida después del intercambio de nodos en la solución destino.

También se ha implementado una segunda estrategia llamada *PR2* que es conceptualmente igual que la primera, excepto que se realizan las transformaciones desde la solución 2 hasta la solución 1. Ahora la solución origen (solución de partida) es la solución 2 y la solución destino (solución guía) la solución 1.
Puede parecer un cambio insignificante pero esta estrategia es comúnmente usada, ya que el espacio de soluciones se explora desde otro ángulo, alcanzando antes (o después) cierto grupo de soluciones que podrían ser las óptimas.

Figura 2.11. Pseudocódigo Path Relinking 1.

```
{x:TipoSolucion} = PR1(solución1:TipoSolucion,solucion2:TipoSolucion)
variables
    RPrima,C : Vector
    i,j: Entero
    solucionAux ,mejorSolucion: TipoSolucion;
Inicio
    mejorSolucion = solucion1
    solucionAux = solucion1
    mientras solucionAux ≠ solucion2 y NO mejora hacer
        eliminarNodo(solucionAux,elijeNodo(solucionAux,solucion2))
        insertarNodo(solucionAux,elijeNodo(solucion2,solucionAux))
        si NoFactible(solucionAux) entonces
            completar(solucionAux)
        fin si
        solucionAux = DosPorUno(solucionAux)
        solucionAux = UnoPorCero(solucionAux)
        si esMejor(solucionAux,mejorSolucion) entonces
            mejorSolucion = solucionAux
        fin si
    fin mientras
    devuelve mejorSolucion
fin
```

Capitulo 3.

Objetivos

El objetivo que se persigue es el diseño, implementación y mantenimiento de un producto software que resuelva el problema de la ubicación de dispositivos que regeneran señales en una red de comunicaciones mediante la creación de reglas heurísticas y optimización de éstas. Este objetivo global puede dividirse en una serie de sub-objetivos que se detallan a continuación:

- Aprender a programar en lenguaje Java.
- Familiarizarse y dominar el entorno de desarrollo *NetBeans IDE 6.0.1*.
- Familiarizarse y dominar la librería jxl de Java que permite crear y editar ficheros con extensión .xls (de Excel) en el entorno de desarrollo *NetBeans IDE 6.0.1*.
- Familiarizarse con el entorno *Dia* de creación y edición de diagramas UML.
- Familiarizarse con los términos de heurística y metaheurística y las estrategias que definen.
- Familiarizarse con los distintos términos y estrategias de optimización de soluciones.
- Aprender a crear algoritmos constructivos que generen soluciones al problema
- Aprender a crear algoritmos de optimización de soluciones para el problema.
- Aprender a crear algoritmos heurísticos y metaheurísticos que proporcionen soluciones al problema.

Capitulo 4

Descripción informática

En este capítulo se expone lo referente al estudio realizado desde un punto de vista Informático.

4.1. Herramientas utilizadas

Se va a hacer una breve introducción y explicación de las herramientas utilizadas en el desarrollo del proyecto.

4.1.1. Plataforma NetBeans

NetBeans es un proyecto de código abierto de gran éxito, como demuestra la cantidad de usuarios que lo usan aumentan día a día. El proyecto *NetBeans* se inició en el año 2000 por *Sun MicroSystem*. En la actualidad hay dos productos de libre distribución disponibles, el *NetBeans IDE* y *NetBeans Platform*. Aunque se ha trabajado sobre *NetBeans IDE* se va a introducir los dos productos:

- *NetBeans IDE* es un entorno de desarrollo, es decir, una herramienta que puedan usar los desarrolladores de *software* para escribir, depurar, compilar y ejecutar programas. Este programa está escrito en Java aunque soporta multitud de lenguajes de programación. Además al ser un *software* de código abierto existen multitud de módulos que extienden su funcionalidad.

- *NetBeans Platform* es una base modular y extensible usada para crear grandes aplicaciones de escritorio. Empresas independientes asociadas, especializadas en el desarrollo de *software*, proporcionan extensiones adicionales de fácil integración en la plataforma que ayudan a desarrollar sus propias herramientas y soluciones.

Ambos son productos de código abierto, que cualquiera puede obtener y reutilizar para satisfacer sus necesidades. Se ha usado *NetBeans IDE 6.0.1* y algunas de las características más importantes que incorpora dicho entorno de desarrollo son:

4.1.2. Java

First Person, una filial de *Sun Microsystems* especializada en software para pequeños dispositivos, decidió desarrollar un nuevo lenguaje adecuado a sus necesidades. Entre estas necesidades estaban la reducción del coste de pruebas en relación a otros lenguajes como C o C++, la orientación a objetos, la inclusión de bibliotecas gráficas y la independencia del sistema operativo. Así, de la mano de James Glosing, nació *Oak*. Al poco tiempo, en 1994, cerró *First Person* al no despegar ni los proyectos de TV interactiva, ni los electrodomésticos inteligentes.

Uno de los desarrolladores de Unix y fundadores de *Sun Microsystems*, Bill Joy, pensó que *Oak* podía ser el lenguaje que Internet necesitaba y en 1995, tras una pequeña adaptación de *Oak*, nacía Java.

Entre las principales características de Java se pueden citar:

- Sintaxis similar a la de C++. Se simplifican algunas características del lenguaje como: la sobrecarga de operadores, la herencia múltiple, el paso por referencia de parámetros, la gestión de punteros, la liberación de memoria y las instrucciones de pre-compilación.

- Soporte a la Programación Orientada a Objetos. A diferencia de C++, que puede considerarse un lenguaje multi-paradigma, Java está diseñado específicamente para utilizar el paradigma de orientación a objetos.

- Independencia de la plataforma. En Java se pretende que con una sola compilación se obtenga código ejecutable en diferentes Sistemas Operativos e incluso sobre diferente hardware.

4.1.3. Dia

Dia es una aplicación informática de propósito general para la creación de diagramas, desarrollada como parte del proyecto GNOME. Está concebido de forma modular, con diferentes paquetes de formas para diferentes necesidades.

Dia está diseñado como un sustituto de la aplicación comercial Visio de Microsoft. Se puede utilizar para dibujar diferentes tipos de diagramas. Actualmente se incluyen diagramas entidad-relación, diagramas UML, diagramas de flujo, diagramas de redes, diagramas de circuitos eléctricos, etc. Nuevas formas pueden ser fácilmente agregadas, dibujándolas con un subconjunto de SVG e incluyéndolas en un archivo XML.

El formato para leer y almacenar gráficos es XML (comprimido con gzip, para ahorrar espacio). Puede producir salida en los formatos EPS, SVG y PNG. También conviene recordar que Dia, gracias al paquete dia2code, puede generar el esqueleto del código a escribir, si utilizáramos con tal fin un UML.

4.1.4. JXL

Java Excel API es un proyecto de código abierto de java API que permite leer, escribir y modificar hojas de cálculo Excel dinámicamente. Ahora, los desarrolladores de Java pueden leer las hojas de cálculo Excel, modificarlas con una API conveniente y simple, y escribir los cambios a cualquier flujo de salida (por ejemplo, el disco, HTTP, base de datos, etcétera).

En cualquier sistema operativo donde pueda correr una máquina virtual Java (es decir, no sólo Windows) se puede procesar y entregar las hojas de cálculo Excel. Debido a que es Java, la API se puede invocar desde dentro de un *servlet*, dando así acceso a hojas de cálculo Excel a través de Internet y aplicaciones de intranet.

Algunas características:

- Lee datos de Excel 95, 97, 2000, XP, 2003 y 2007.
- Lee y escribe las fórmulas de cálculo (Excel 97 y posteriores)
- Genera hojas de cálculo en formato de Excel 2000
- Soporta los tipos de fuentes, número y formato de las fechas.
- Soporta la configuración de celdas
- Modifica libros de cálculo existentes.

4.2. Requisitos funcionales

Un requisito funcional, debe cumplir las siguientes condiciones:

- Son declaraciones de los servicios que debe proporcionar el sistema.
- Especifica la manera en que éste debe reaccionar a determinadas entradas.
- Especifica cómo debe comportarse el sistema en situaciones particulares.
- Pueden declarar explícitamente lo que el sistema no debe hacer

Los requisitos funcionales de nuestro proyecto:

- Todos los algoritmos de construcción de soluciones implementados deben proporcionar soluciones factibles independientemente del problema dado.

- Todos los algoritmos de optimización deben devolver soluciones coherentes independientemente de la instancia del problema que se esté estudiando.

- Los resultados obtenidos por los algoritmos deben ser correctamente codificados en un fichero Excel.

- Se debe desarrollar el proyecto con un diseño software lo más sostenible posible, atendiendo a buenas prácticas de diseño e implementación.

- Se deben diseñar e implementar pruebas que garanticen y demuestren el correcto funcionamiento del proyecto.

4.3. Requisitos no funcionales

Los requisitos no funcionales no se refieren a funciones específicas que proporciona el sistema, sino que son las restricciones de los servicios o funciones ofrecidas por el sistema. Por lo general son requisitos que surgen de las necesidades de los usuarios y se aplican al sistema en su totalidad. Se pueden clasificar los requisitos no funcionales en tres subgrupos, en los que se englobarán los requisitos no funcionales del proyecto:

- Los requisitos no funcionales del producto son requisitos que especifican el comportamiento del producto:

    - El programa debe ser razonablemente rápido en cuanto a su ejecución, teniendo en cuenta que lógicamente a instancias más complejas, mayor será el tiempo de ejecución.

  - En ningún caso la ejecución del programa debe suponer el agotamiento de la memoria.

  - El programa debe ser de una alta fiabilidad, independientemente de la instancia del grafo que se estudie o los algoritmos que se utilicen el programa debe ser capaz de terminar su ejecución correctamente.

- Los requisitos no funcionales organizacionales son los derivados de políticas y procedimientos existentes en la organización del cliente y/o el desarrollador.

  - La codificación del proyecto debe realizarse en el lenguaje Java.

  - El diseño de los diagramas de clases debe realizarse en el lenguaje de modelado *UML*.

- Los requisitos no funcionales externos son los derivados de factores externos al sistema y a su sistema de desarrollo.

  - Compromiso por parte del estudiante que realiza el proyecto en cuanto a realizar un trabajo minucioso y de calidad.

4.4. Ciclo de vida

El ciclo de vida es un marco de referencia que contiene todos los procesos, actividades y tareas que se realizan durante el desarrollo, la explotación y el mantenimiento del *software*, desde que este se define hasta que se deja de utilizar. Para el desarrollo de *software* se pueden utilizar principalmente cuatro modelos de ciclo de vida:

- Modelo en cascada, el inicio de cada etapa del ciclo debe esperar a la finalización de la etapa inmediatamente anterior.

- Modelo iterativo incremental, es un modelo de tipo evolutivo basado en varios ciclos Cascada realimentados y aplicados repetidamente, con una filosofía iterativa.

- Modelo de construcción de prototipos, basado en la construcción de prototipos que evalúa el cliente y sobre los que se va añadiendo funcionalidades.

- Modelo en espiral, las actividades se conforman en una espiral en la que cada iteración o bucle representa un conjunto de actividades.

En este proyecto, se ha utilizado el modelo en espiral. Este modelo consiste en la división del proyecto en distintos ciclos. En cada ciclo se implementa una nueva funcionalidad hasta que finalmente se llega al final del mismo. Cada uno de estos ciclos se divide en las siguientes etapas:

- Análisis de requisitos: en esta etapa se deben identificar y especificar los objetivos del ciclo de desarrollo.

- Discusión de alternativas: se debate sobre las distintas posibilidades que se han propuesto y se decide cuál es la más conveniente.

- Desarrollo: en esta etapa se implementa la alternativa que se eligió anteriormente.

- Validación: se debe validar el sistema utilizando pruebas para comprobar que se cumplen todos los requisitos acordados.

- Planificación: Se revisa el proyecto para, posteriormente, empezar a planificar la siguiente iteración.

Figura 4.1. Modelo de ciclo de vida en espiral.

4.5. Diagrama de Gantt

En el diagrama de Gantt de la Figura 4.2 se muestra de forma esquemática el tiempo necesario que el alumno ha debido invertir para cumplir los objetivos mencionados en el Capítulo 3.

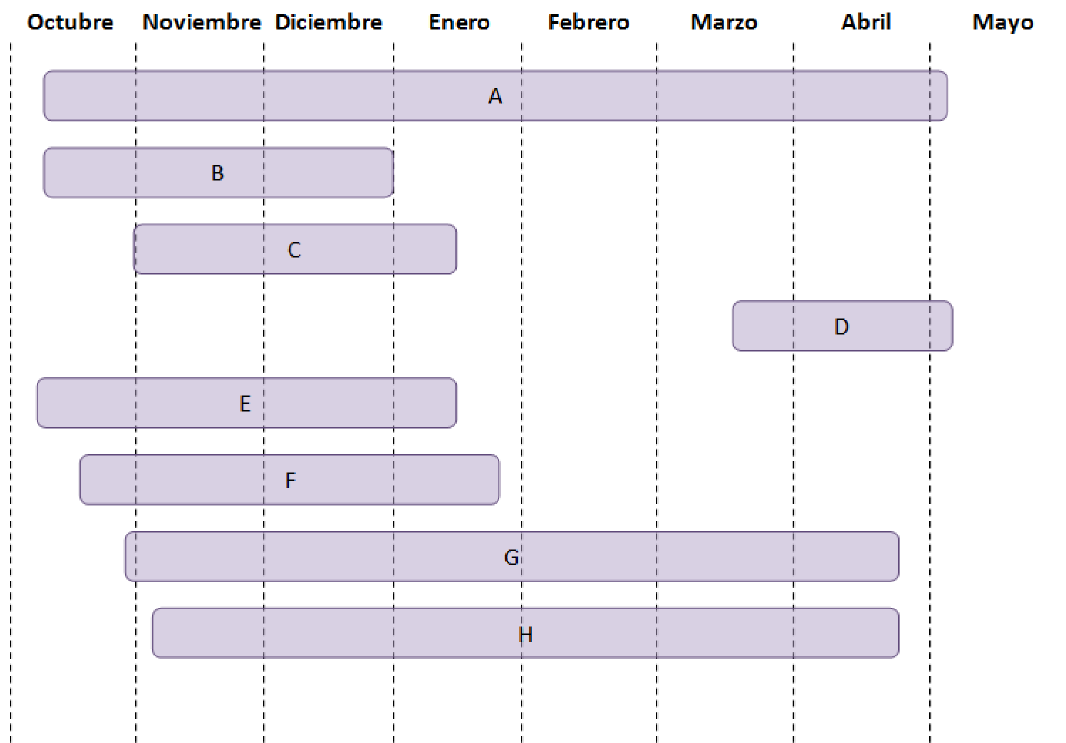

Figura 4.2. Diagrama de Gantt representando los objetivos marcados en el tiempo.

A. Aprender a programar en lenguaje Java.

B. Familiarizarse y dominar el entorno de desarrollo *NetBeans IDE 6.0.1*.

C. Familiarizarse y dominar la librería jxl de Java que permite crear y editar ficheros con extensión .xls (de Excel) en el entorno de desarrollo *NetBeans IDE 6.0.1*.

D. Familiarizarse con el entorno *Dia* de creación y edición de diagramas UML.

E. Familiarizarse con los términos de heurísticas y metaheurística y las estrategias que definen.

F. Familiarizarse con los distintos términos y estrategias de optimización de soluciones.

G. Aprender a crear algoritmos heurísticos y metaheurísticos que proporcionen soluciones al problema.

H. Aprender a crear algoritmos de optimización de soluciones para el problema.

4.6. Descripción algorítmica desde el punto de vista informático

Se ha utilizado el paradigma de orientación a objetos (*POO*) para implementar el código del proyecto. Este paradigma permite la utilización de buenas prácticas de diseño, como el patrón *template*.

4.6.1. Patrón Template

Este patrón de diseño se ha aplicado en el proyecto en varias ocasiones, como en el diseño de los algoritmos constructivos, las búsquedas locales, el re encaminamiento de trayectorias, etcétera.

El patrón *Template* permite hacer del proyecto un proyecto más general y sostenible. Contemplando la posibilidad de que en el futuro se pueda ampliar la funcionalidad o añadir nuevas clases que describan nuevos algoritmos.

En este patrón se define una clase padre o superclase que reúne las características comunes a las que serán sus clases hija. En nuestro caso por ejemplo, y como se muestra en la Figura 4.3., la clase Constructivo sería una clase padre que define los métodos a implementar `crearSolucion()` y `getNombreAlgoritmo()` para que las implementen las clases hijo.

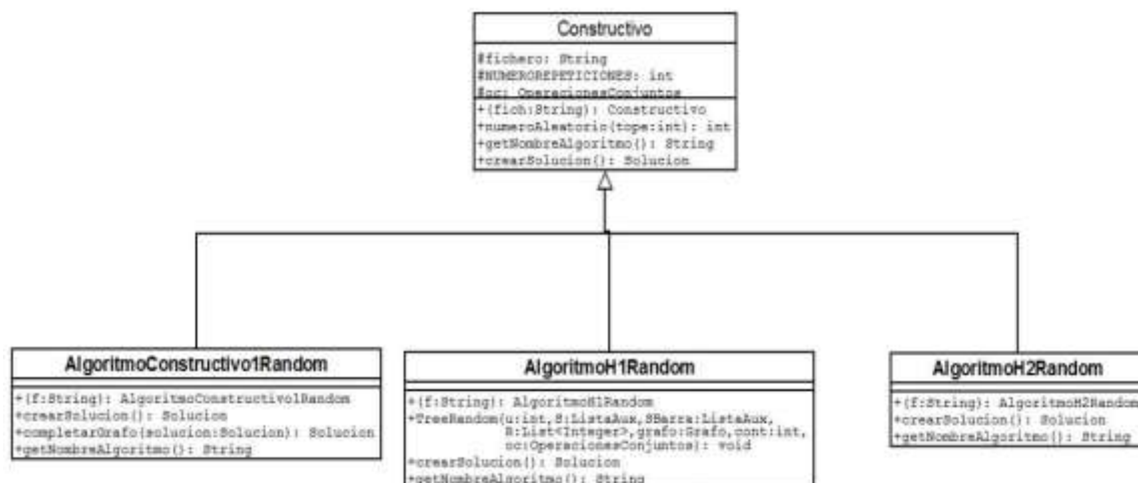

Figura 4.3. Jerarquía de la clase `Constructivo`.

En la Figura 4.4. se muestra una vista global de las clases o módulos que componen el proyecto. En esta vista se han suprimido los métodos u operaciones y los atributos o propiedades con el fin de simplificar el esquema, pero a más adelante se describirán en detalle.

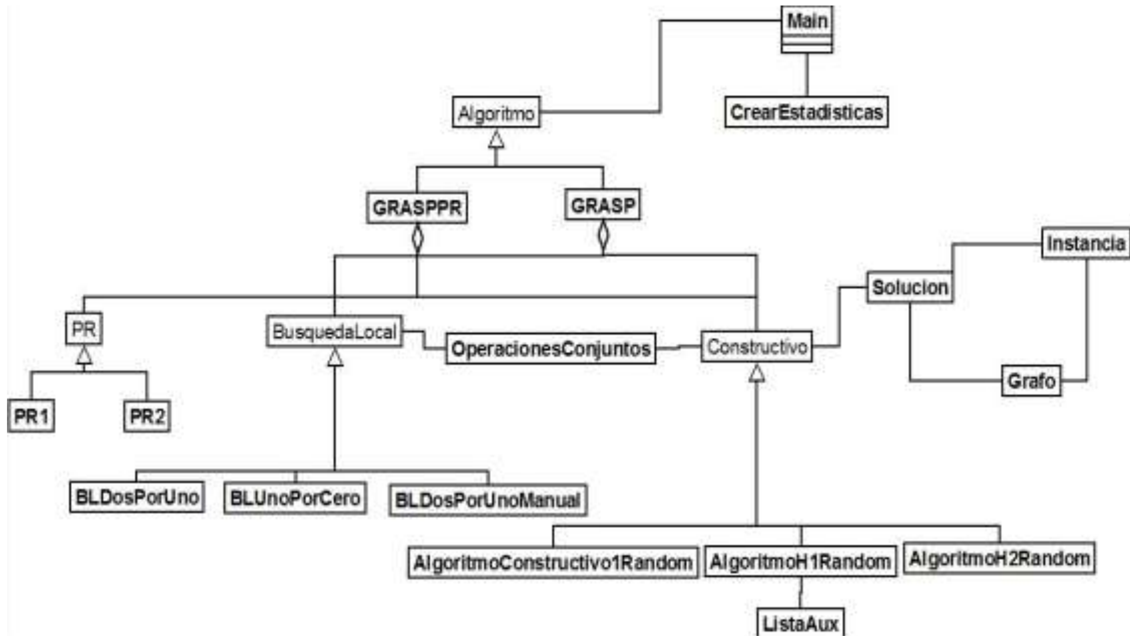

Figura 4.4. Vista general de las clases del proyecto.

4.6.2. Clase grafo

El módulo `Grafo` pretende recoger toda la información relevante a los grafos del problema, modelando así esta información de interés.

Figura 4.5. Clase `Grafo`.

Atributos o propiedades

Contiene una estructura de matriz que representa el grafo, donde los índices enteros representan los nodos del grafo, y el cruce de estos el valor de las aristas que une ese par de nodos. Además contiene un atributo que representa el número de nodos del grafo que se utilizará en las operaciones.

Constructor

Cabecera: `public Grafo(float[][] g, int n)`
Objetivos: Construir un objeto de la clase `Grafo`.
Entrada: Se recibe una estructura matriz que representa el grafo y un entero que representa el número de nodos del grafo.
Salida: Se devuelve un objeto `Grafo`.
Descripción: Cuando se invoca el método devuelve un objeto de la clase `Grafo`.

Clone

Cabecera: `public Grafo clone() throws CloneNotSupportedException`
Objetivos: Devolver un objeto de la clase `Grafo` en una nueva posición de memoria.
Entrada: El método no recibe parámetros.
Salida: Se devuelve un objeto `Grafo`.
Descripción: Cuando se invoca el método se devuelve un objeto de la clase `Grafo` en una nueva posición de memoria.

Completo

Cabecera: `public boolean completo()`
Objetivos: Saber si un grafo es completo o no.
Entrada: El método no recibe parámetros.
Salida: Se devuelve un valor booleano.
Descripción: Cuando se invoca el método se devuelve un valor booleano que indica si el objeto que lo invoca es o no un grafo completo.

Vecinos

Cabecera: `public List<Integer> vecinos(int nodo)`
Objetivos: Obtener los vecinos de un nodo.
Entrada: El método recibe un valor entero.
Salida: Se devuelve una lista de enteros.
Descripción: Cuando se invoca el método se devuelve una lista de enteros que contiene los nodos que son vecinos del nodo recibido.

VisualizarGrafo

Cabecera: `public void visualizarGrafo()`
Objetivos: Mostrar en pantalla el grafo.
Entrada: El método no recibe parámetros.
Salida: El método no devuelve ningún valor.
Descripción: Cuando se invoca el método se visualiza en pantalla el grafo, en forma de matriz y con el valor de las aristas que unen sus nodos.

ActualizarGrafo

Cabecera: `public void actualizarGrafo(int nodoElegido)`
Objetivos: Actualizar los valores del grafo ubicando un regenerador en un nodo dado.
Entrada: El método recibe un valor entero.
Salida: El método no devuelve ningún valor.
Descripción: Cuando se invoca el método se actualiza el valor de las aristas que unen los nodos del objeto `Grafo` invocador como resultado de ubicar un regenerador en el nodo dado.

NuevasConexiones

Cabecera: `public int nuevasConexiones(List<Integer> vecinos)`
Objetivos: Dada una lista de vecinos calcula el número de nuevas conexiones que se crean entre estos.
Entrada: El método recibe una lista de enteros.
Salida: El método devuelve un valor entero.
Descripción: Cuando se invoca el método se devuelve el número de conexiones que se crearían entre una lista de nodos.

4.6.3. Clase Instancia

La clase `Instancia` ha sido implementada con el fin de modelar la información de un fichero de entrada, de la que se dispondrá en memoria durante toda la ejecución del proyecto y no será corrompida ni modificada.

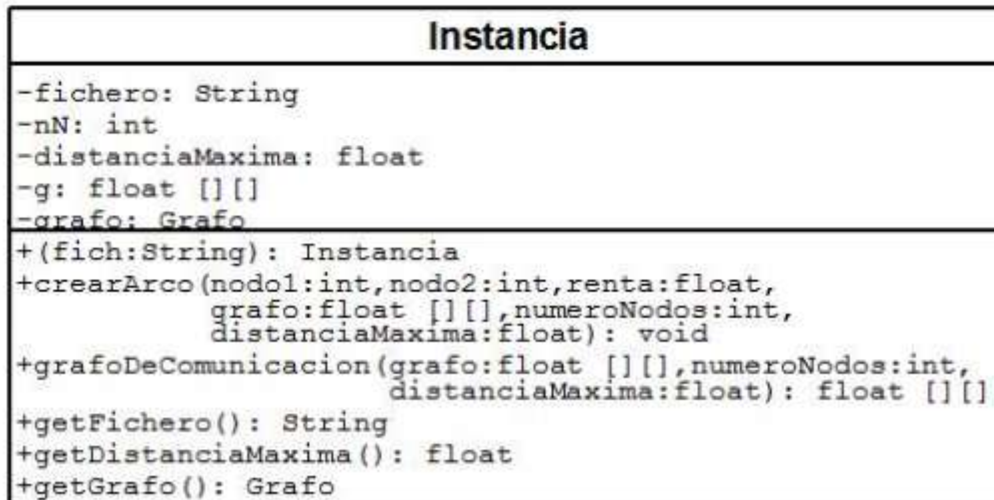

Figura 4.6. Clase `Instancia`.

Atributos o propiedades

La clase contiene un atributo `fichero` que representa la ruta del fichero del que se obtienen los datos de entrada. Un atributo `nN` (`numeroNodos`) que representa el número de nodos del grafo leído. `DistanciaMaxima` representa la distancia máxima que una señal puede recorrer antes de degradarse. El atributo `g` es una estructura matriz que contendrá los datos del grafo que se vayan leyendo de fichero. El atributo `grafo` se construye una vez leído `g` y `nN` y es un objeto que representa fielmente la información aportada por el fichero de entrada.

Constructor

Cabecera: `public Instancia(String fich)`
Objetivos: Construir un objeto de la clase `Instancia`.
Entrada: El método recibe un *`String`*.
Salida: El método devuelve un objeto `Instancia`.
Descripción: Cuando se invoca el método se devuelve un objeto de la clase `Instancia`.

CrearArco

Cabecera: `private static void crearArco(int i, int j, float renta,float [][] grafo, int numeroNodos,float distanciaMaxima)`
Objetivos: Crear una arista entre dos nodos.
Entrada: El método recibe tres enteros, dos reales y una estructura matriz de reales.
Salida: El método no devuelve nada.

Descripción: Este método se invoca en la creación del grafo de comunicación. Pretende crear las aristas nuevas sin necesidad de utilizar regeneradores (explicado en capítulos anteriores). Para ello recibe dos nodos, *i* y *j*, un parámetro renta que es la distancia que la señal puede recorrer sin deteriorarse habiendo ido de *i* a *j*. Junto con la información del grafo este método es todo lo que necesita para crear los nuevos arcos del grafo de comunicación.

GrafoDeComunicacion

Cabecera: `private float [][] grafoDeComunicacion(float [][] grafo, int numeroNodos,float distanciaMaxima)`
Objetivos: Crear el grafo de comunicación.
Entrada: El método recibe dos parámetros reales y uno entero.
Salida: El método devuelve un matriz de reales.
Descripción: Este método se invoca para la creación de grafo de comunicación. En primer lugar elimina los arcos mayores que `distanciaMaxima` y después apoyándose en `crearArco` construye los arcos factibles sin usar regeneradores.

4.6.4. Clase Solucion

La clase `Solucion` ha sido construida para modelar los datos que nos interesa obtener como solución del problema.

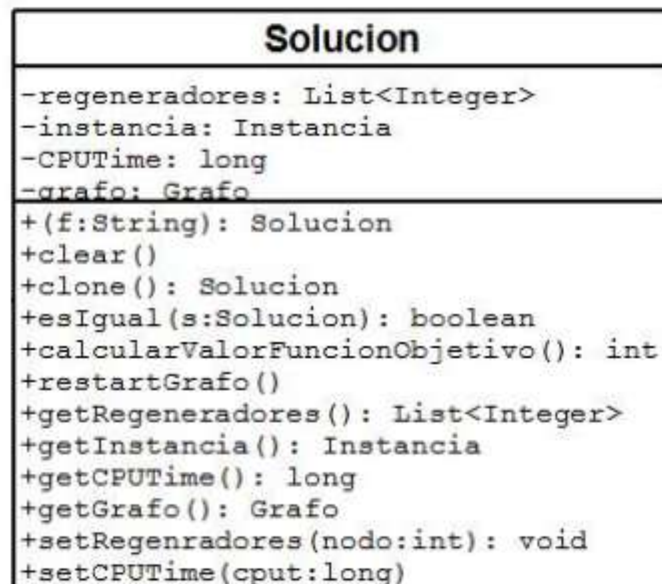

Figura 4.7. Clase `Solucion`.

Atributos o propiedades

La clase almacena en `regeneradores` una lista de los nodos que albergan un regenerador. Además también contiene un objeto de la clase `Instancia` dónde encontrar los datos de entrada originales. Una variable `CPUTime` para determinar cuánto tiempo tardó en construirse esa solución. Viene dada en milisegundos. Por último una instancia de la clase `Grafo` que representa el grafo solución, sobre el que se harán operaciones, actualizaciones, etcétera.

Constructor

Cabecera: `public Solucion(String f)`
Objetivos: Devolver un objeto de la clase `Solucion`.
Entrada: El método recibe un *String*.
Salida: El método devuelve un objeto `Solucion`.
Descripción: Este método se invoca para crear un objeto de la clase Solución. La cuál construye un objeto `Instancia` con la ruta de fichero que recibe y un objeto de la clase `Grafo` independiente que copia del grafo de la clase `Instancia`.

Clear

Cabecera: `public void clear()`
Objetivos: Restaurar los valores del objeto `Solucion` a los valores iniciales.
Entrada: El método no recibe parámetros.
Salida: El método no devuelve nada.
Descripción: Este método reinicia a vacío la lista de regeneradores, inicializa la variable de `CPUTime` y restaura el grafo sobre el que trabaja la solución copiando el original de la instancia que posee.

Clone

Cabecera: `public Solucion clone() throws CloneNotSupportedException`
Objetivos: Devolver un objeto exactamente idéntico al invocador pero en una posición de memoria nueva.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un objeto `Solucion`.
Descripción: Este método sobrecarga el método clone intrínseco de Java y devuelve un objeto `Solucion` en una posición de memoria nueva.

EsIgual

Cabecera: `public boolean esIgual(Solucion s2)`
Objetivos: Determinar si una solución dada es igual a la invocadora.
Entrada: El método recibe un objeto `Solucion`.
Salida: El método devuelve un valor booleano.
Descripción: Dada una solución se determina si es igual a la invocadora. Se considera que es igual cuando ambas tienen los mismos regeneradores ubicados en los mismos nodos.

CalcularValorFuncionObjetivo

Cabecera: `public int calcularValorFunciónObjetivo()`
Objetivos: Determinar como de buena es una solución.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un valor entero.
Descripción: El método devuelve un valor entero que indica como de buena es una solución, cuanto más alto este valor mejor será la solución. Para hacer una función general se resta al número de nodos del grafo el número de regeneradores de la solución, así las soluciones con menos regeneradores devolverán valores mayores.

RestartGrafo

Cabecera: `public void restartGrafo()`
Objetivos: Garantizar la coherencia del grafo de la solución.
Entrada: El método no recibe parámetros.
Salida: El método no devuelve ningún valor.
Descripción: El método inicializa el grafo acorde al de la instancia original y vuelve a colocar los regeneradores recalculando los arcos que se crean. Esta función se utiliza por ejemplo en el re encaminamiento de trayectorias, cuando se añaden y quitan regeneradores a una solución y se quiere tener el grafo coherente a esta lista de regeneradores.

4.6.5. Clase Constructivo

La clase `Constructivo` ha sido concebida como plantilla para algoritmos constructores de soluciones. Es una clase abstracta que reúne las características comunes que estos algoritmos deben cumplir.

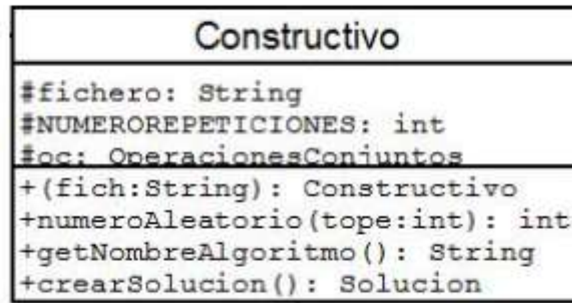

Figura 4.8. Clase `Constructivo`.

Atributos o propiedades

La clase `Constructivo` contiene un `fichero` que especificará la ruta del archivo que nos proporcionará la información del grafo. La constante `NUMEROREPETICIONES` indica el número de veces que el algoritmo itera para construir ese número de soluciones y quedarse con la mejor. La propiedad `oc` es un paquete implementado por el estudiante que contiene operaciones típicas de las estructuras de conjuntos.

Constructor

Cabecera: `public Constructivo(String fich)`
Objetivos: Definir cómo crear un objeto de las clases que hereden de `Constructivo`.
Entrada: El método recibe un *String*.
Salida: El método devuelve un objeto `Constructivo`.
Descripción: El método define cómo crear un objeto de las clases que hereden de `Constructivo`, es un método común.

NumeroAleatorio

Cabecera: `protected static int numeroAleatorio(int tope)`
Objetivos: Devolver un número aleatorio.
Entrada: El método recibe un entero.
Salida: El método devuelve un entero.
Descripción: El método devuelve un valor entero generado aleatoriamente entre 0 y el valor recibido `tope`.

GetNombreAlgoritmo

Cabecera: `public abstract String getNombreAlgoritmo()`
Objetivos: Definir un método abstracto que implementen las clases herederas de `Constructivo`.

Entrada: El método no recibe nada.
Salida: El método devuelve un *String*.
Descripción: Se define un método abstracto que implementen las clases herederas de `Constructivo` de manera que cada una devuelva su nombre.

<u>ConstruirSolucion</u>

Cabecera: `public abstract Solucion crearSolucion()`
Objetivos: Definir un método abstracto que implementen las clases herederas de `Constructivo`.
Entrada: El método no recibe nada.
Salida: El método devuelve un objeto de la clase `Solucion`.
Descripción: Se define un método abstracto que implementen las clases herederas de `Constructivo` para que cada una de ellas implemente su algoritmo.

4.6.6. Clase AlgoritmoConstructivo1Random

Es una de las clases herederas de `Constructivo`, lo que significa que es una de las encargadas de construir una solución a un problema dado.

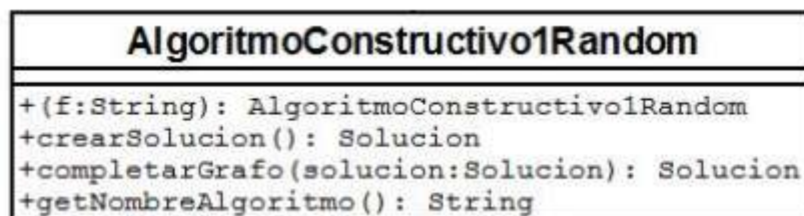

Figura 4.9. Clase `AlgoritmoConstructivo1Random`.

<u>Constructor</u>

Cabecera: `public AlgoritmoConstructivo1Random(String f)`
Objetivos: Crear un objeto de la clase `AlgoritmoConstructivo1Random`.
Entrada: El método recibe un *String*.
Salida: El método devuelve un objeto de la clase `AlgoritmoConstructivo1Random`.
Descripción: Se devuelve un objeto de la clase `AlgoritmoConstructivo1Random`. Para ello se utiliza el constructor de la clase padre.

CrearSolucion

Cabecera: `public Solucion crearSolucion()`
Objetivos: Implementar el código que define este algoritmo.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un objeto de la clase `Solucion`.
Descripción: Se devuelve un objeto de la clase `Solucion` creado al problema dado mediante la ejecución del algoritmo. Se recuerda que este algoritmo está explicado en el Capitulo 2 y a continuación se explica con un ejemplo.

Supóngase que se recibe el grafo de la Figura 4.10. El algoritmo evalúa todos los nodos del grafo para elaborar la lista de candidatos. La evaluación de un nodo en este algoritmo consiste en el número de conexiones nuevas que generarían si se ubicase un regenerador en ellos.

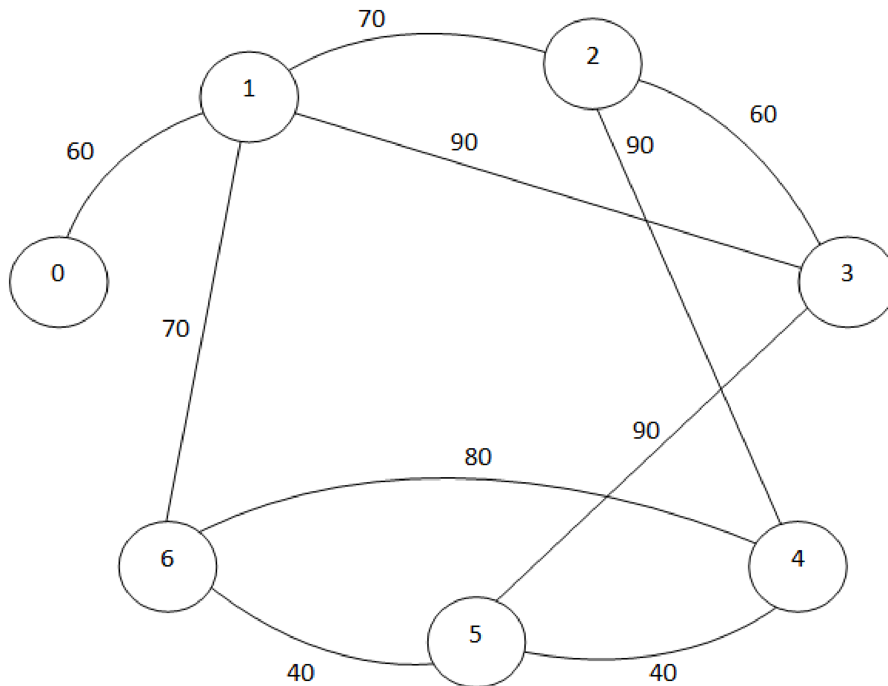

Figura 4.10. Grafo inicial.

En este caso la evaluación de los nodos sería:

Eval(nodo(0))=0
Eval(nodo(1))=5
Eval(nodo(2))=2
Eval(nodo(3))=2
Eval(nodo(4))=2
Eval(nodo(5))=2
Eval(nodo(6))=2.

Con estos datos se crea la lista de candidatos restringida RCL, de la que formarán parte todos los nodos que superen el valor del parámetro de corte.

$$th = \min(eval(e)) + \alpha\,(\max(eval(e)) - \min(eval(e)))$$

En el ejemplo *th*=2.5, de manera que la *RCL* = {1}, de esta lista se elige un nodo al azar. Al haber solo un elemento el nodo elegido será el 1, actualizando el grafo como se muestra en la Figura 4.11.

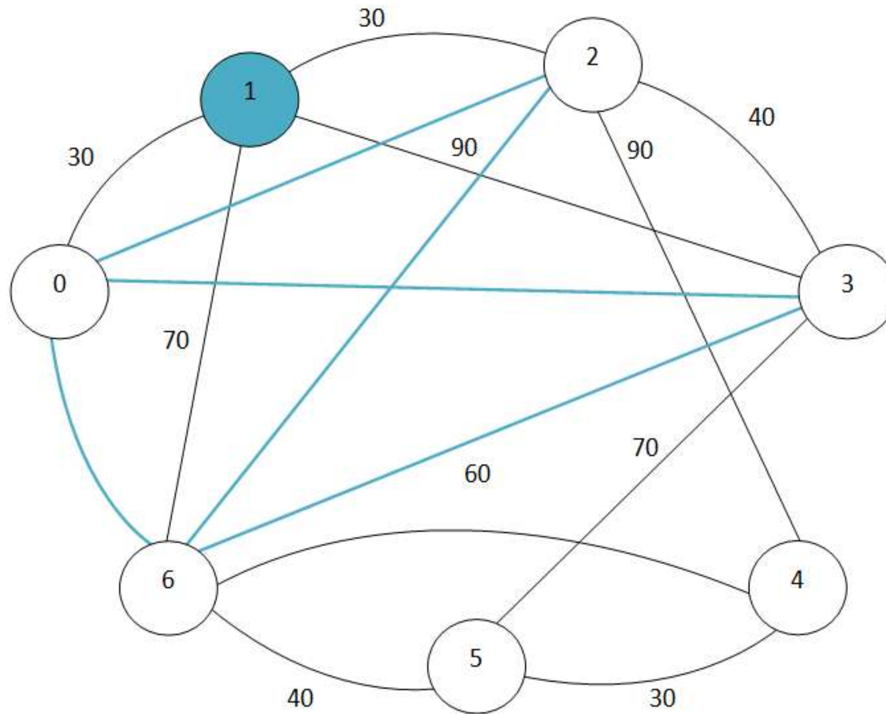

Figura 4.11. Grafo después de ubicar un regenerador en el nodo 1.

Este proceso se repite de manera cíclica hasta conseguir un grafo completo. Lo que en este caso supone una vuelta más consiguiendo una evaluación de nodos:

Eval(nodo(0))=0
Eval(nodo(1))=0
Eval(nodo(2))=3
Eval(nodo(3))=3
Eval(nodo(4))=1
Eval(nodo(5))=1
Eval(nodo(6))=6.

Donde *th*=3.0 y *RCL* = {6}. Finalmente y después de alojar un regenerador en el nodo 6 el grafo se actualizaría como se muestra en la figura 4.12.

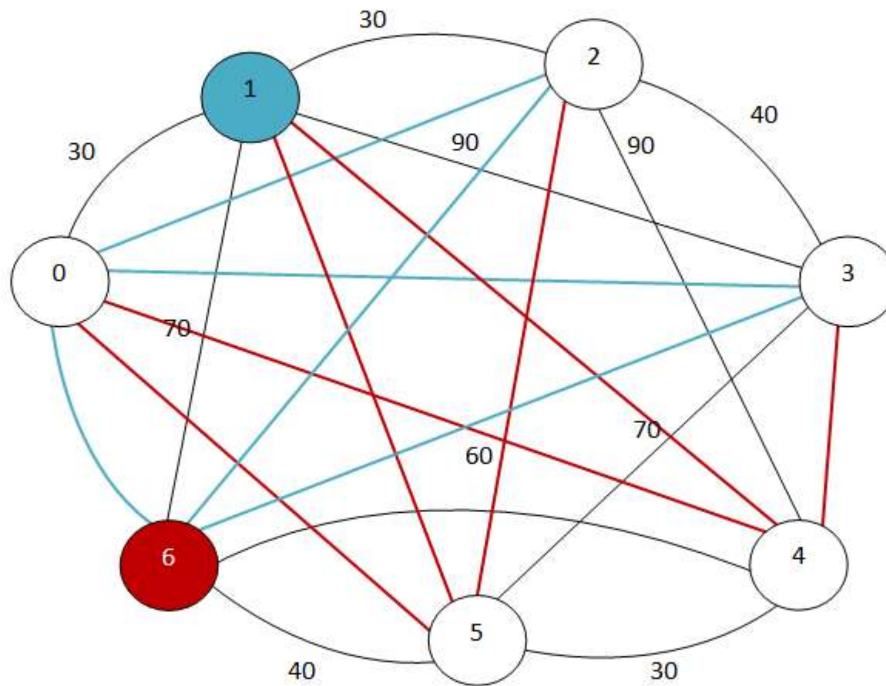

Figura 4.12. Grafo completo.

<u>CompletarSolucion</u>

Cabecera: `public Solucion completarGrafo(Solucion solucion)`
Objetivos: Completar soluciones incompletas.
Entrada: El método recibe un tipo `Solucion`.
Salida: El método devuelve un objeto de la clase `Solucion`.
Descripción: Se recibe una solución incompleta, y se completa su lista de nodos con regenerador hasta que es completo. Para ello se miran todos los nodos del grafo, y se elige el nodo que más arcos nuevos cree si se ubica un regenerador en él, esto se repite de manera cíclica hasta que la solución se completa.

<u>GetNombreAlgoritmo</u>

Cabecera: `public String getNombreAlgoritmo()`
Objetivos: Devolver el nombre del algoritmo.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un *String*.
Descripción: Se devuelve el nombre del algoritmo.

4.6.7. Clase AlgoritmoH1Random

Define otra manera de proporcionar soluciones a un problema en base a ideas de un algoritmo distinto.

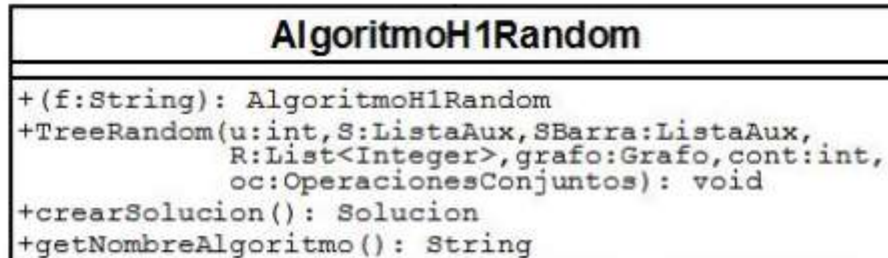

Figura 4.13. Clase `AlgoritmoH1Random`.

<u>Constructor</u>

Cabecera: `public AlgoritmoH1Random(String f)`
Objetivos: Crear un objeto de la clase `AlgoritmoH1Random`.
Entrada: El método recibe un *String*.
Salida: El método devuelve un objeto de la clase `AlgoritmoH1Random`.
Descripción: Se devuelve un objeto de la clase `AlgoritmoH1Random`. Para ello se utiliza el constructor de la clase padre.

<u>TreeRandom</u>

Cabecera: `public static void TreeRandom(int u,ListaAux S,ListaAux SBarra,List<Integer>R, Grafo grafo, int cont, OperacionesConjuntos oc)`
Objetivos: Ayudar a `crearSolucion` a generar una solución factible a un problema.
Entrada: El método recibe dos enteros, dos objetos de la clase `ListaAux`, una lista de enteros, un objeto de la clase `Grafo` y un objeto de la clase `OperacionesConjuntos`.
Salida: No devuelve parámetros.
Descripción: Es un método recursivo, que actualizando los conjuntos de nodos visitados y no visitados, *`S`* y *`SBarra`* respectivamente consigue crear el árbol de expansión y finalmente actualizar el conjunto solución *`R`*.

<u>CrearSolucion</u>

Cabecera: `public Solucion crearSolucion()`
Objetivos: Crear una solución siguiendo el algoritmo que se describe en esta clase.
Entrada: El método no recibe ningún parámetro.

Salida: Devuelve una solución al problema.
Descripción: El método construye una solución siguiendo el algoritmo descrito en el Capítulo 2.

Si se sigue con el ejemplo del grafo de la Figura 4.10. El algoritmo comienza seleccionando el nodo perteneciente al grafo con menor número de vecinos, es decir el nodo 0. Después se apoya en el método `TreeRandom` pasando como parámetro el nodo seleccionado, el conjunto de nodos visitados o estudiados, el conjunto de nodos no visitados, la información relativa al grafo que se estudia, un contador y el objeto para realizar operaciones sobre conjuntos:

```
TreeRandom( 0, {0}, {1,2,3,4,5,6}, grafo, 0, oc)
```

El método estudia los vecinos del nodo recibido, en este caso el 0, como en este caso el nodo 0 sólo tiene un vecino se selecciona como nodo donde alojar un regenerador este vecino, el nodo 1. Dejando el grafo como se muestra en la Figura 4.11.

El procedimiento recursivo se llama a sí mismo después de ubicar el primer regenerador con los siguientes valores:

```
TreeRandom( 1, {0}, {1,2,3,4,5,6}, grafo, 1, oc)
```

En esta iteración se estudian los vecinos no visitados de 1, es decir, 2,3 y 6. La evaluación de los nodos es:

Eval(nodo(2))=1 Eval(nodo(3))=1 Eval(nodo(6))=2.

De donde se deduce que *th*=1.5 y la *RCL* = {6}, de esta lista se elije aleatoriamente un elemento, en este caso 6 y se actualiza el grafo dejándolo como el de la Figura 4.12.

En este momento el grafo ya es completo y todos los nodos han sido visitados, volviendo así los métodos recursivos sus llamadas y terminando, devolviendo como solución *R*={1,6}.

<u>GetNombreAlgoritmo</u>

Cabecera: public String getNombreAlgoritmo()
Objetivos: Devolver el nombre del algoritmo.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un String.
Descripción: Se devuelve el nombre del algoritmo.

4.6.8. Clase AlgoritmoH2Random

Hereda de Constructivo, lo que significa que es otra de las clases que construyen una solución factible a un problema dado.

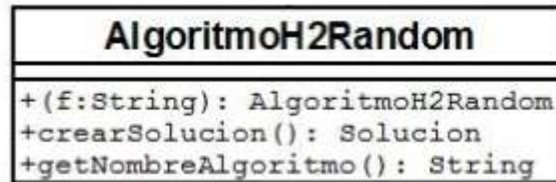

Figura 4.14. Clase AlgoritmoH2Random.

Constructor

Cabecera: `public AlgoritmoH2Random(String f)`
Objetivos: Devolver un objeto de la clase `AlgoritmoH2Random`.
Entrada: El método recibe un *String*.
Salida: El método devuelve objeto de esta clase.
Descripción: Se devuelve un objeto de la clase `AlgoritmoH2Random` cuando es invocado.

CrearSolucion

Cabecera: `public Solucion crearSolucion()`
Objetivos: Crear una solución a un problema dado.
Entrada: El método no recibe parámetros.
Salida: El método devuelve objeto de la clase `Solucion`.
Descripción: Se devuelve un objeto `Solucion` que modela una solución a una instancia dada siguiendo las ideas que defiende este algoritmo.

El grafo de la Figura 4.10. sirve para explicar el método mediante un ejemplo. Es una mezcla de los dos anteriores. En primer lugar se recorren todos los nodos buscando el que tenga un menor número de vecinos, en este caso el nodo 0. Después se estudian sus vecinos, pero como solo tiene un vecino que es el nodo 1, es el seleccionado para alojar un regenerador actualizando el grafo y dejándolo como en la Figura 4.11.

El procedimiento es iterativo de manera que en la segunda iteración se vuelve a mirar en todo el grafo cuál es el nodo con un menor número de vecinos, en este caso es el 4. Se evalúan los vecinos del nodo 4, que son 2,5 y 6 siendo sus valores:

Eval(nodo(2))=4 Eval(nodo(5))=3 Eval(nodo(6))=5.

El valor del parámetro de corte *th* será 4, de manera que la *RCL* = {2,6}. Entre los que se elije uno al azar, supongamos que se selecciona el nodo 2 y se actualiza el grafo como se aprecia en la Figura 4.15.

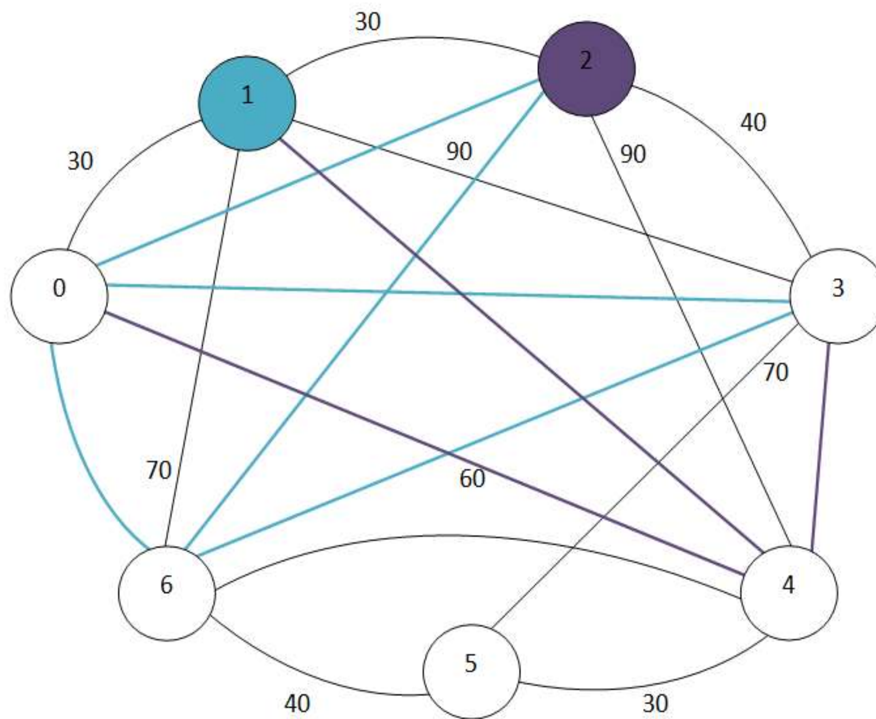

Figura 4.15. Grafo con aristas actualizadas después de ubicar regeneradores en los nodos 1 y 2.

El algoritmo itera otra vez porque faltan nodos por comunicar. En la siguiente iteración el nodo con menor número de vecinos y por tanto el que se somete a estudio es el nodo 5. La evaluación de sus vecinos es:

Eval(nodo(3))=4 Eval(nodo(4))=4 Eval(nodo(6))=4.

y el parámetro *th* = 4. Lo que implica que la *RCL* = {3,4,6}. De entre todos los candidatos se elije al azar, supongamos que se elije el nodo 3, el grafo se completaría tal y como se muestra en la Figura 4.16.

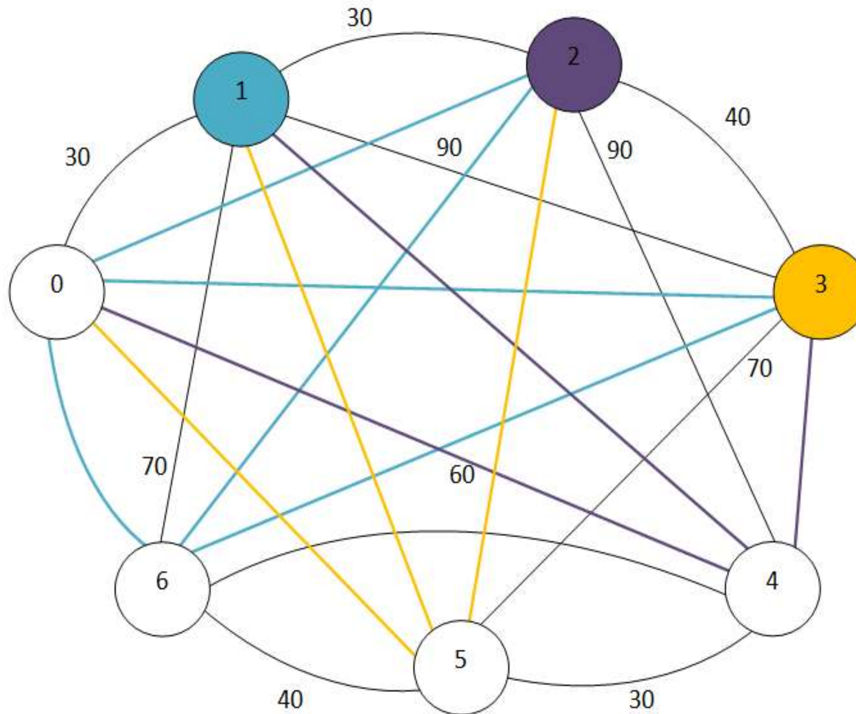

Figura 4.16. Grafo completo con regeneradores en los nodos 1,2 y 3.

Finalmente el algoritmo devolvería como solución un conjunto de regeneradores R= {1, 2, 3}.

4.6.9. Clase ListaAux

Define una estructura auxiliar para las listas. Ha sido creada para que la utilice la clase `AlgoritmoH1Random`, una clase que implementa métodos recursivos. Había un problema cuando se realizaban operaciones de conjuntos sobre las listas en estos métodos recursivos, y es que se creaban objetos nuevos, lo que producía que al volver de la recursividad los conjuntos no estuviesen actualizados como debían.
Con esta clase el objeto siempre es único, es decir, siempre es un objeto de `ListaAux`. Contiene una estructura de lista primitiva de Java y aunque esta cambie en operaciones, la clase contenedora `ListaAux` no cambiará en todo el proceso.

Figura 4.17. Clase `ListaAux`.

Atributos o propiedades

Contiene un objeto típico de listas en Java.

GetLista

Cabecera: `public int getLista(int n)`
Objetivos: Devuelve el elemento especificado de la lista.
Entrada: El método recibe un entero.
Salida: El método devuelve un entero.
Descripción: Se devuelve el nodo de la posición *n*.

GetLista

Cabecera: `public List<Integer> getLista()`
Objetivos: Devuelve el objeto lista.
Entrada: El método no recibe parámetros.
Salida: El método devuelve un tipo lista.
Descripción: Cuando se invoca se devuelve la lista de enteros.

SetLista

Cabecera: `public void setLista(int n)`
Objetivos: Añade el elemento *n* a la lista.
Entrada: El método recibe un entero.
Salida: No se devuelve nada.
Descripción: Se añade a la lista el elemento se especifique en la llamada.

SetLista

Cabecera: `public void setLista(List<Integer> l)`
Objetivos: Asigna a la lista de la clase la lista recibida.
Entrada: El método recibe una lista.
Salida: No se devuelve nada.
Descripción: Cuando se invoca se asigna a la lista del objeto invocador la lista que se pasa como parámetro.

Clear

Cabecera: `public void clear()`
Objetivos: Vaciar la lista.
Entrada: El método no recibe parámetros.
Salida: No se devuelve nada.

Descripción: Se vacía la lista cuando es invocado.

4.6.10. Clase OperacionesConjuntos

Esta clase reúne los métodos para operar con listas como si se tratasen de conjuntos.

OperacionesConjuntos
+unionConjuntos(a:List<Integer>,b:List<Integer>): List<Integer>
+restaConjuntos(a:List<Integer>,b:List<Integer>): List<Integer>
+interseccionConjuntos(a:List<Integer>,b:List<Integer>): List<Integer>
+complementario(a:List<Integer>,n:int): List<Integer>

Figura 4.18. Clase `OperacionesConjuntos`.

UnionConjuntos

Cabecera: `public List<Integer> unionConjuntos(List<Integer> A, List<Integer> B)`
Objetivos: Hacer la unión del conjunto *A* y *B*.
Entrada: El método recibe dos listas de enteros.
Salida: Se devuelve una lista de enteros.
Descripción: En la lista devuelta estarán los elementos comunes a la lista *A* y la *B*.

RestaConjuntos

Cabecera: `public List<Integer> restaConjuntos(List<Integer> A, List<Integer> B)`
Objetivos: Hacer la resta del conjunto *A* y *B*.
Entrada: El método recibe dos listas de enteros.
Salida: Se devuelve una lista de enteros.
Descripción: En la lista devuelta estarán los elementos comunes de *A* sin los elementos que pudiesen coincidir con la lista *B*.

InterseccionConjuntos

Cabecera: `public List<Integer> interseccionConjuntos(List<Integer> A, List<Integer> B)`
Objetivos: Hacer la intersección del conjunto *A* y *B*.
Entrada: El método recibe dos listas de enteros.
Salida: Se devuelve una lista de enteros.
Descripción: En la lista devuelta estarán los elementos comunes de *A* y *B*.

Complementario

Cabecera: public List<Integer> complementario(List<Integer> A, int n)
Objetivos: Hacer el complementario del conjunto *A*.
Entrada: El método recibe una lista de enteros y un entero.
Salida: Se devuelve una lista de enteros.
Descripción: En la lista devuelta estarán los elementos complementarios a *A*. El entero que recibe es para saber el tope, el número de elementos máximo.

4.6.11. Clase BusquedaLocal

La clase BusquedaLocal es una clase abstracta que sirve de plantilla para todos los algoritmos de optimización basados en ideas de búsqueda local.

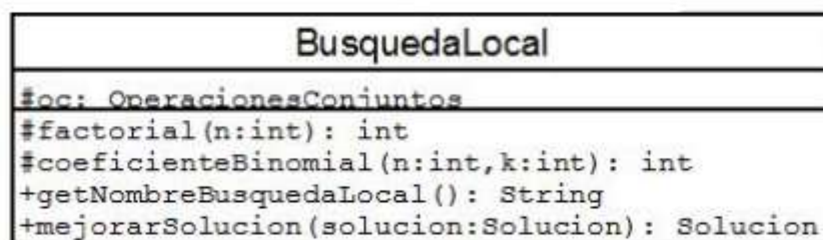

Figura 4.19 Clase BusquedaLocal.

Atributos o propiedades

Esta clase cuenta con un atributo protegido que define las operaciones sobre conjuntos, se ubica aquí para que todas las clases que hereden de esta cuenten con este atributo.

Factorial

Cabecera: protected int Factorial(int n)
Objetivos: Devolver el factorial de un número dado.
Entrada: El método recibe un entero.
Salida: Se devuelve un entero.
Descripción: Cuando se invoca el método se devuelve un número entero que es el factorial del entero recibido.

CoeficienteBinomial

Cabecera: protected int CoeficienteBinomial(int n,int k)
Objetivos: Devolver el coeficiente binomial de dos valores dados.

Entrada: El método recibe dos enteros.
Salida: Se devuelve un entero.
Descripción: Cuando se invoca el método se devuelve un número entero que es el coeficiente binomial de los dos enteros recibidos. Se utiliza para saber el número de posibles combinaciones de k en k de un conjunto de nodos n. Por ejemplo combinaciones de 2 en 2 elementos de un conjunto de 5 elementos.

GetNombreBusquedaLocal

Cabecera: `public abstract String getNombreBusquedaLocal()`
Objetivos: Definir un método plantilla que implementarán las clases herederas.
Entrada: El método no recibe nada.
Salida: Se devuelve un *String*.
Descripción: Definir un método plantilla que implementarán las clases herederas con el fin de que devuelvan el nombre del algoritmo.

MejorarSolucion

Cabecera: `public abstract Solucion mejorarSolucion(Solucion solucion)`
Objetivos: Definir un método plantilla que implementarán las clases herederas.
Entrada: El método recibe un objeto de la clase `Solucion`.
Salida: Se devuelve un objeto de la clase `Solucion`.
Descripción: Definir un método plantilla que implementarán las clases herederas con el fin de plasmar los algoritmos que describe cada una.

4.6.12. Clase BL2x1

Con esta clase se define el primer algoritmo de optimización de soluciones, dónde se intentará sustituir dos regeneradores de una solución dada por un solo regenerador manteniendo la correcta construcción de la solución.

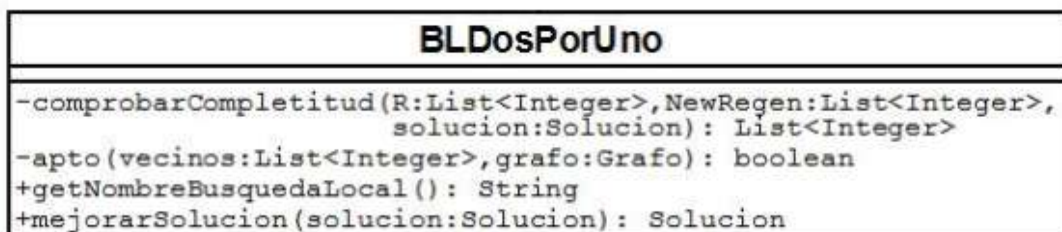

Figura 4.20. Clase `BLDosPorUno`.

ComprobarCompletitud

Cabecera: `private List<Integer> comprobarCompletitud(List<Integer> R, List<Integer> NewRegen, Solucion solucion)`
Objetivos: Cerciorarse de que el conjunto de candidatos a sustituir los dos nodos estudiados es factible.
Entrada: El método recibe el conjunto de regeneradores de la solución dada, el conjunto de candidatos y la solución.
Salida: Lista de regeneradores candidatos.
Descripción: Definir un método que compruebe si todos los candidatos de la lista son correctos.

Apto

Cabecera: `private boolean apto(List<Integer> vecinos, Grafo grafo)`
Objetivos: Cerciorarse de que no se quitan nodos imprescindibles en la solución.
Entrada: El método recibe una lista de nodos y el grafo original.
Salida: Un valor booleano.
Descripción: Se define un método que no permita eliminar de la solución los nodos que tengan un vecino que solo sea accesible mediante estos nodos.

GetNombreBusquedaLocal

Cabecera: `public String getNombreBusquedaLocal()`
Objetivos: Devolver el nombre del algoritmo de optimización.
Entrada: El método no recibe parámetros.
Salida: Un valor *String*.
Descripción: Se devuelve un valor *String* con el nombre del algoritmo de optimización.

MejorarSolucion

Cabecera: `public Solucion mejorarSolucion(Solucion solucion)`
Objetivos: Optimizar la solución recibida si es posible.
Entrada: El método recibe un objeto `Solucion`.
Salida: Devuelve un objeto `Solucion`.
Descripción: La idea del algoritmo es estudiar los vecinos de los dos nodos en los que hay regenerador y se quiere sustituir por uno solo. Para este conjunto de vecinos se comprueba si existe algún vecino de alguno de estos, que alojando un regenerador comunique a todos los que quedan por comunicar como consecuencia de quitar los dos regeneradores.

Se toma como ejemplo el grafo de la Figura 4.16. Se va a intentar sustituir los regeneradores de los nodos 2 y 3 por uno solo que deje la solución factible, que debería ser el nodo 6.

Del conjunto solución R={1,2,3} se eliminan los nodos seleccionados i=2 y j=3, dando origen a *RPrima*={1}. El conjunto de candidatos se inicializa a todos los nodos, salvo los contenidos en *RPrima*, C={0,2,3,4,5,6}.

Se estudian los vecinos de los nodos i=2 y j=3, es decir N={1,2,3,4,5} y para cada uno de ellos se comprueba si alguno de sus vecinos coincide con el conjunto *RPrima* ({1} en este caso). Si entre estos vecinos se encuentra el nodo 1 no se hace nada, pero si los vecinos no coinciden con el nodo 1 se actualiza la solución de candidatos. Dejando como candidatos los nodos que ya eran candidatos y además coinciden con los vecinos del nodo n.

En primer lugar el nodo n=1 tiene como vecinos $N(n)$={0,2,3,6} como entre ellos no se encuentra el nodo {1} se actualiza el conjunto $C=C \cap N(n)$, C={0,2,3,6}.

El nodo n=2 tiene como vecinos $N(n)$={1,3,4} y como contiene el nodo 1 no se hace nada.

Los vecinos de n=3 son $N(n)$={1,2,5} y tampoco se actualiza el conjunto de candidatos.

Los vecinos de n=4 son $N(n)$={2,5,6}. Se actualiza el conjunto solución dejándolo C={2,6}.

Por último, los vecinos de n=5 son $N(n)$={3,4,6} y se actualiza el conjunto solución, quedando este con el valor final C={6}.

De manera que el grafo finalmente pasa a tener dos regeneradores en los nodos 1 y 6 como se muestra en la figura 4.13.

4.6.13. Clase BL1x0

Esta clase implementa una solución de optimización que debido a la aleatoriedad de los métodos constructivos tiene mucho sentido. Intenta dejar una solución con N regeneradores en una solución factible con N-1 regeneradores sin sustituir ninguno de estos, simplemente eliminando alguno que sobre porque con los restantes se completa el grafo.

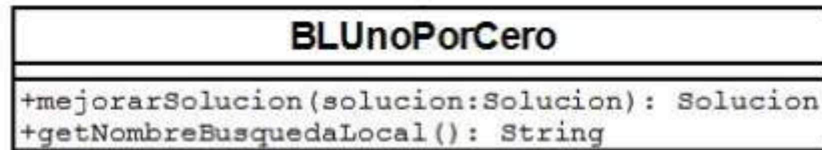

Figura 4.21. Clase `BLUnoPorCero`.

GetNombreBusquedaLocal

Cabecera: `public String getNombreBusquedaLocal()`
Objetivos: Devolver el nombre del algoritmo de optimización.
Entrada: El método no recibe parámetros.
Salida: Un valor *String*.
Descripción: Se devuelve un valor *String* con el nombre del algoritmo de optimización.

MejorarSolucion

Cabecera: `public Solucion mejorarSolucion(Solucion solucion)`
Objetivos: Optimizar la solución recibida si es posible.
Entrada: El método recibe un objeto `Solucion`.
Salida: Devuelve un objeto `Solucion`.
Descripción: La idea del algoritmo es intentar eliminar un nodo y probar con los nodos restantes si son una solución completa, si es solución completa la devuelve como una solución optimizada y sino prueba con otro nodo.

En el ejemplo del grafo de la Figura 4.10. tiene sentido si suponemos una solución R={1,2,6}. El algoritmo de optimización uno por cero probaría a quitar el nodo 1 y comprobar si R={2,6} dejan como solución factible el grafo. Ante la negativa a esta cuestión el algoritmo prueba ahora a quitar el nodo 2, y comprueba que la solución R={1,6} es una solución óptima devolviéndola, dejando el grafo como se muestra en la Figura 4.12.

4.6.14. Clase BL2x1Manual

Esta clase se construyó con el mismo objetivo que la clase `BLDosPorUno` pero con un algoritmo de fuerza bruta.

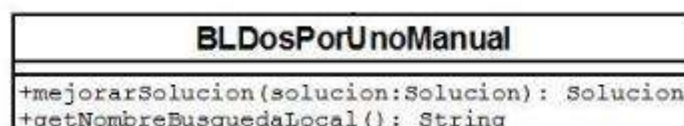

Figura 4.22. Clase `BLDosPorUnoManual`.

GetNombreBusquedaLocal

Cabecera: `public String getNombreBusquedaLocal()`
Objetivos: Devolver el nombre del algoritmo de optimización.
Entrada: El método no recibe parámetros.
Salida: Un valor *String*.
Descripción: Se devuelve un valor *String* con el nombre del algoritmo de optimización.

MejorarSolucion

Cabecera: `public Solucion mejorarSolucion(Solucion solucion)`
Objetivos: Optimizar la solución recibida si es posible.
Entrada: El método recibe un objeto `Solucion`.
Salida: Devuelve un objeto `Solucion`.
Descripción: La idea del algoritmo es intentar reemplazar dos nodos con regenerador por solo uno. La diferencia es que este algoritmo es completamente manual, es decir, elije dos nodos con regenerador e intenta buscar uno entre todos los nodos del grafo que pueda reemplazar a estos dos. Si lo consigue devuelve la solución optimizada y si no lo consigue prueba a elegir otros dos nodos diferentes de la solución y a repetir la operación.

4.6.15. Clase PR

Es otra clase abstracta que sigue el patrón de diseño *Template*. En esta ocasión sirve de plantilla para las clases que quieran implementar algoritmos de optimización basados en las ideas del re-encadenamiento de trayectorias (*Path Relinking*).

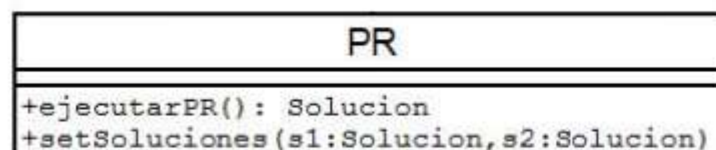

Figura 4.23. Clase `PR`.

ejecutarPR

Cabecera: `public abstract Solucion ejecutarPR()`
Objetivos: Definir un método general donde las clases herederas implementen sus algoritmos.
Entrada: El método no recibe parámetros.
Salida: Devuelve un objeto `Solucion`.

Descripción: Se define un método general donde las clases herederas implementen sus algoritmos.

SetSoluciones

Cabecera: `protected abstract void setSoluciones(Solucion s1,Solucion s2)`
Objetivos: Asignar las soluciones al objeto de las clases que hereden de PR para trabajar con ellas.
Entrada: El método recibe dos objetos `Solucion`.
Salida: No devuelve valores.
Descripción: Es un método abstracto con lo que las clases herederas tendrán que implementarlo para que cada una asigne a sus soluciones iniciales los variables recibidos.

4.6.16. Clase PR1

Hereda de la clase *PR* anteriormente explicada. Es el primer algoritmo de re encadenamiento de trayectorias implementado.

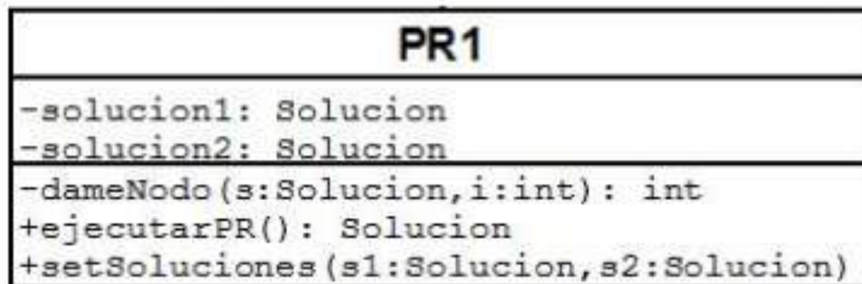

Figura 4.24. Clase `PR1`.

Atributos o propiedades

La clase tiene dos atributos de ámbito privado, son dos objetos de la clase `Solucion` con los que debe trabajar el algoritmo implementado.

DameNodo

Cabecera: `private int dameNodo(Solucion s,int i)`
Objetivos: Devolver el nodo de la posición *i* de la solución *s*.
Entrada: Recibe un objeto de la clase `Solucion` y otro entero.
Salida: Devuelve un entero.

Descripción: El método mira en la lista de nodos con regenerador de la solución recibida y devuelve el nodo que se encuentra en la posición *i*.

SetSoluciones

Cabecera: `protected void setSoluciones(Solucion s1, Solucion s2)`
Objetivos: Asignar las soluciones recibidas a las propiedades `Solucion` del objeto.
Entrada: Recibe dos objetos de la clase `Solucion`.
Salida: No devuelve valores.
Descripción: El método asigna a su propiedad `solucion1` la solución `s1` recibida y a `solucion2` la `s2`.

EjecutarPR

Cabecera: `public Solucion ejecutarPR()`
Objetivos: Intentar encontrar una solución mejor que las dos recibidas en el constructor.
Entrada: No se reciben parámetros de entrada.
Salida: Devuelve un objeto Solucion.
Descripción: El método parte de la `solucion1` recibida en el constructor e intenta transformarla en la `solucion2`. En este camino de transformación estudia si cada nueva solución generada podría devolver una solución óptima mediante el aplique de técnicas de optimización basadas en búsqueda local.

Supóngase que el algoritmo recibe como `solucion1` *R1*={1,5,6} y como `solucion2` *R2*={1,2,4}. El algoritmo de re-encadenamiento de trayectorias elije un nodo a eliminar de *R1* que no esté en *R2*, por ejemplo *nE*=6 y un nodo a insertar que esté en *R2* y no en *R1*, por ejemplo *nI*=2. Con lo que tenemos una solución auxiliar *RAux*={1,5,2}. Esta solución no es una solución factible (deja el grafo incompleto) a la instancia de Figura 4.10. Sobre un grafo incompleto no se pueden aplicar algoritmos de optimización por contener información inconsistente, de manera que se completa dejando la solución en *RAux*={1,5,2,3}. Esta solución deja el grafo consistente pero evidentemente no es la más óptima, incluso a empeorado las dos recibidas. Aunque si aplicamos la búsqueda local de dos por uno el algoritmo sustituye los nodos *i*=5 y *j*=2 por el nodo 2, lo que deja la solución auxiliar con los regeneradores *RAux*={1,2,3}.Como se ha mejorado la solución se repite el algoritmo de búsqueda local, cambiando en esta iteración los nodos *i*=2 y *j*=3 por el nodo 6. Dejando una solución final de *RAux*={1,6}mejor que las dos recibidas y por tanto devuelta como solución óptima encontrada.

4.6.17. Clase PR2

Hereda de la clase `PR` anteriormente explicada. Es el segundo algoritmo de re encadenamiento de trayectorias implementado.

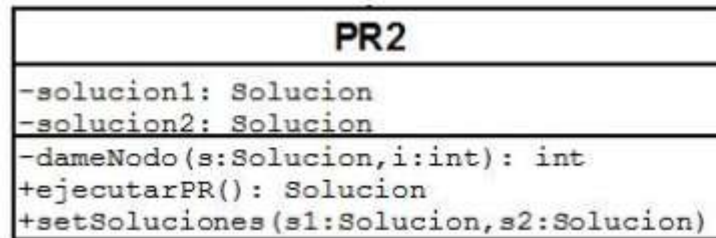

Figura 4.25. Clase `PR2`.

Atributos o propiedades

La clase tiene dos atributos de ámbito privado, son dos objetos de la clase `Solucion` con los que debe trabajar el algoritmo implementado.

DameNodo

Cabecera: `private int dameNodo(Solucion s,int i)`
Objetivos: Devolver el nodo de la posición *i* de la solución *s*.
Entrada: Recibe un objeto de la clase `Solucion` y otro entero.
Salida: Devuelve un entero.
Descripción: El método mira en la lista de nodos con regenerador de la solución recibida y devuelve el nodo que se encuentra en la posición *i*.

SetSoluciones

Cabecera: `protected void setSoluciones(Solucion s1, Solucion s2)`
Objetivos: Asignar las soluciones recibidas a las propiedades `Solucion` del objeto.
Entrada: Recibe dos objetos de la clase `Solucion`.
Salida: No devuelve valores.
Descripción: El método asigna a su propiedad `solucion1` la solución `s1` recibida y a `solucion2` la `s2`.

EjecutarPR

Cabecera: public Solucion ejecutarPR()
Objetivos: Intentar encontrar una solución mejor que las dos recibidas en el constructor.
Entrada: No se reciben parámetros de entrada.

Salida: Devuelve un objeto Solucion.
Descripción: El método parte de la `solucion2` recibida en el constructor e intenta transformarla en la `solucion1`, es el proceso inverso al descrito en la clase `PR1`. En este camino de transformación estudia si cada nueva solución generada podría devolver una solución óptima mediante el aplique de técnicas de optimización basadas en búsqueda local.

4.6.18. Clase Algoritmo

Es una clase abstracta que funciona de plantilla para los algoritmos metaheurísticos que se implementen.

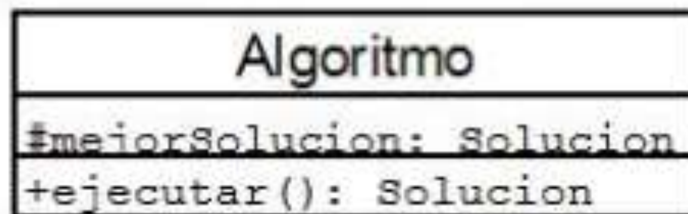

Figura 4.26. Clase `Algoritmo`.

Atributos o propiedades

Esta clase alberga un objeto de la clase `Solucion`, es la mejor solución que se encontrará como resultado de ejecutar los algoritmos de las clases herederas de esta.

Ejecutar

Cabecera: `public abstract Solucion ejecutar()`
Objetivos: Definir un método donde todas las clases herederas de esta clase implementen sus algoritmos metaheurísticos.
Entrada: No se reciben parámetros de entrada.
Salida: Devuelve un objeto `Solucion`.
Descripción: Se define un método donde todas las clases herederas de esta clase implementen sus algoritmos metaheurísticos.

4.6.19. Clase GRASP

La clase `GRASP` implementa el algoritmo metaheurístico *GRASP* que comentamos en capítulos anteriores.

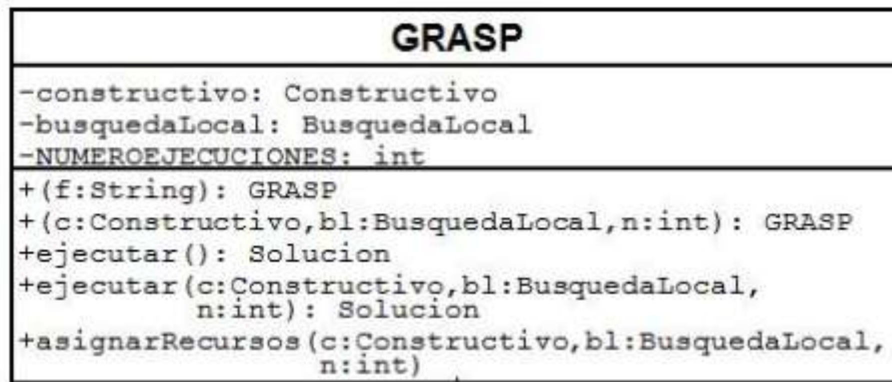

Figura 4.27. Clase `GRASP`.

Atributos o propiedades

`GRASP` tiene una propiedad privada que hace referencia a un objeto de la clase Constructivo, este objeto es el que utiliza el algoritmo para construir soluciones. Tiene un objeto que implementa un algoritmo de optimización de búsqueda local que utiliza la clase en su algoritmo.

Por último, le indicamos un número de ejecuciones, que son las veces que el algoritmo iterará sobre una instancia construyendo e intentando mejorar la solución obtenida en cada ocasión.

Constructor1

Cabecera: `public GRASP(String f)`
Objetivos: Crear un objeto de esta clase.
Entrada: Se recibe un parámetro *String*.
Salida: Devuelve un objeto `GRASP`.
Descripción: Este constructor crea objetos de la clase `GRASP` recibiendo un *String* que contiene la ruta de un fichero. Con este constructor a las propiedades del método, necesarias para ejecutar el código del algoritmo, se les asigna valores manualmente en código, valores por defecto.

Constructor2

Cabecera: `public GRASP(Constructivo c,BusquedaLocal bl, int n)`
Objetivos: Crear un objeto de esta clase.
Entrada: Se recibe un parámetro `Contructivo`, otro de `BusquedaLocal` y un *`int`*.
Salida: Devuelve un objeto `GRASP`.

Descripción: Este constructor crea objetos de la clase `GRASP` recibiendo los valores para sus atributos, un objeto que construya soluciones, otro que intente mejorarlas y el número de iteraciones.

Ejecutar

Cabecera: `public Solucion ejecutar()`
Objetivos: Crear un objeto `Solucion` utilizando un algoritmo `GRASP`.
Entrada: No se reciben parámetros.
Salida: Devuelve un objeto `Solucion`.
Descripción: Este método devuelve un objeto Solucion de un problema dado siguiendo los pasos definidos por los algoritmos `GRASP`. Se recuerda que este algoritmo itera un número de veces que viene dado por el atributo `NUMEROEJECUCIONES` y en cada una de esas iteraciones se construye una solución a la instancia dada y se intenta optimizar mediante un método de búsqueda local. Finalmente se devuelve la mejor solución hallada.

Ejecutar

Cabecera: `public Solucion ejecutar(Constructivo c, BusquedaLocal bl, int n)`
Objetivos: Crear un objeto `Solucion` utilizando un algoritmo `GRASP`.
Entrada: Se recibe un objeto de la clase `Constructivo`, otro de la clase `BusquedaLocal` y un `int`.
Salida: Devuelve un objeto `Solucion`.
Descripción: Este método devuelve un objeto `Solucion` de un problema dado siguiendo los pasos definidos por los algoritmos *GRASP*. Es un método sobrecargado, y es así con la finalidad de que se pueda ejecutar el algoritmo no solo con las propiedades con las que fue construido, sino con otras distintas que se definan en otro punto del código del proyecto sin necesidad de crear otro objeto `GRASP`.

AsignarRecursos

Cabecera: `public void asignarRecursos(Constructivo c, BusquedaLocal bl, int n)`
Objetivos: Cambiar el valor de las propiedades del objeto `GRASP`.
Entrada: Se recibe un objeto de la clase `Constructivo`, otro de la clase `BusquedaLocal` y un `int`.
Salida: No devuelve parámetros.
Descripción: Este método pretende dar la posibilidad al programador de cambiar el valor de las propiedades del objeto `GRASP`. Asigna a las propiedades del objeto los valores recibidos en el método.

4.6.20. Clase GRASP-PR

La clase GRASPPR define otro algoritmo metaheurístico, basado en el algoritmo *GRASP* pero utilizando además el re-encaminamiento de trayectorias para intentar encontrar soluciones optimas en un espacio de vecindad mayor.

La clase ejecuta un código *GRASP* basado en las mismas ideas que el anterior, pero construye un conjunto de soluciones llamado eliteSet que contiene las *n* mejores soluciones halladas en el algoritmo GRASP.

Finalmente de este conjunto de soluciones se elijen dos al azar con las que se realiza el proceso de re encadenamiento de trayectorias.
El algoritmo devuelve la mejor solución encontrada.

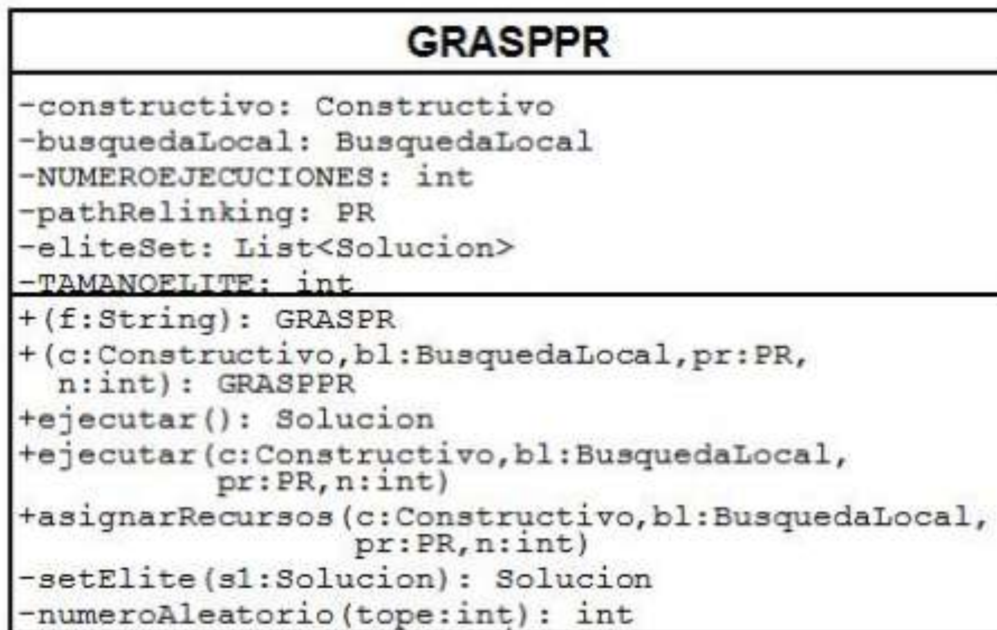

Figura 4.28. Clase GRASPPR.

Atributos o propiedades

Los objetos de esta clase tendrán atributos para realizar el algoritmo GRASP, es decir, un objeto de la clase Constructivo, un objeto de la clase BusquedaLocal y un número de repeticiones. Además incorpora un objeto de la clase PR que será el encargado de realizar la segunda parte del algoritmo. Junto con un conjunto de soluciones que albergará las *n* mejores soluciones. Donde *n* viene determinada por el atributo TAMANOELITE.

Constructor1

Cabecera: `public GRASPPR(String f)`
Objetivos: Crear un objeto de esta clase.
Entrada: Se recibe un parámetro *String*.
Salida: Devuelve un objeto `GRASP-PR`.
Descripción: Este constructor crea objetos de la clase `GRASPPR` recibiendo un *String* que contiene la ruta de un fichero. Con este constructor las propiedades del método, necesarias para ejecutar el código del algoritmo, se les asigna valores manualmente en código, valores por defecto.

Constructor2

Cabecera: `public GRASPPR(Constructivo c, BusquedaLocal bl, PR pr, int n)`
Objetivos: Crear un objeto de esta clase.
Entrada: Se recibe un parámetro de la clase `Contructivo`, otro de `BusquedaLocal`, otro de `PR` y un *`int`*.
Salida: Devuelve un objeto `GRASPPR`.
Descripción: Este constructor crea objetos de la clase `GRASPPR` recibiendo los valores para sus atributos, un objeto que construya soluciones, otro que intente mejorarlas y el número de iteraciones. Además de un objeto que se base en los algoritmos de re encaminamiento de trayectorias.

Ejecutar

Cabecera: `public Solucion ejecutar()`
Objetivos: Crear un objeto `Solucion` utilizando un algoritmo `GRASPPR`.
Entrada: No se reciben parámetros.
Salida: Devuelve un objeto `Solucion`.
Descripción: Este método devuelve un objeto `Solucion` de un problema dado siguiendo los pasos definidos por el algoritmo `GRASPPR`. Se recuerda que este algoritmo itera un número de veces que viene dado por el atributo `NUMEROEJECUCIONES` y en cada una de esas iteraciones se construye una solución a la instancia dada y se intenta optimizar mediante un método de búsqueda local. De esas soluciones se queda con las *n* mejores, donde *n* viene dado por `TAMANOELITE` y entre todas ellas selecciona dos al azar, a las que aplica el re-encadenamiento de trayectorias.

Ejecutar

Cabecera: `public Solucion ejecutar(Constructivo c, BusquedaLocal bl, PR pr, int n)`
Objetivos: Crear un objeto `Solucion` utilizando un algoritmo `GRASPPR`.

Entrada: Se recibe un objeto de la clase `Constructivo`, otro de la clase `BusquedaLocal`, otro de la clase `PR` y un *`int`*.
Salida: Devuelve un objeto `Solucion`.
Descripción: Este método devuelve un objeto `Solucion` de un problema dado siguiendo los pasos definidos por el algoritmo `GRASPPR`. Es un método sobrecargado, con la finalidad de que se pueda ejecutar el algoritmo no solo con las propiedades con las que fue construido, sino con otras distintas que se definan en otro punto del código del proyecto sin necesidad de crear otro objeto `GRASPPR`.

<u>AsignarRecursos</u>

Cabecera: `public void asignarRecursos(Constructivo c, BusquedaLocal bl, PR pr, int n)`
Objetivos: Cambiar el valor de las propiedades del objeto `GRASPPR`.
Entrada: Se recibe un objeto de la clase `Constructivo`, otro de la clase `BusquedaLocal`, otro de la clase `PR` y un `int`.
Salida: No devuelve parámetros.
Descripción: Este método pretende dar la posibilidad al programador de cambiar el valor de las propiedades del objeto `GRASPPR`. Asigna a las propiedades del objeto los valores recibidos en el método.

<u>SetElite</u>

Cabecera: `private void setElite(Solucion s1)`
Objetivos: Estudiar si una solución debe formar parte del conjunto elite o no.
Entrada: Se recibe un parámetro `Solucion`.
Salida: No devuelve valores.
Descripción: El método recibe una solución y comprueba si el conjunto elite no está lleno, si hay hueco la inserta y si no estudia si la solución recibida es mejor que alguna de las que forma parte del conjunto, en cuyo caso la reemplaza.

<u>NumeroAleatorio</u>

Cabecera: `private static int numeroAleatorio(int tope)`
Objetivos: Devolver un número aleatorio.
Entrada: El método recibe un entero.
Salida: El método devuelve un entero.
Descripción: El método devuelve un valor entero generado aleatoriamente entre 0 y el valor recibido `tope`.

4.6.21. CrearEstadisticas

La clase `CrearEstadisticas` ha sido modelada para escribir en una tabla de un fichero Excel los resultados de ejecutar las principales y diferentes combinaciones de algoritmos sobre las instancias disponibles.

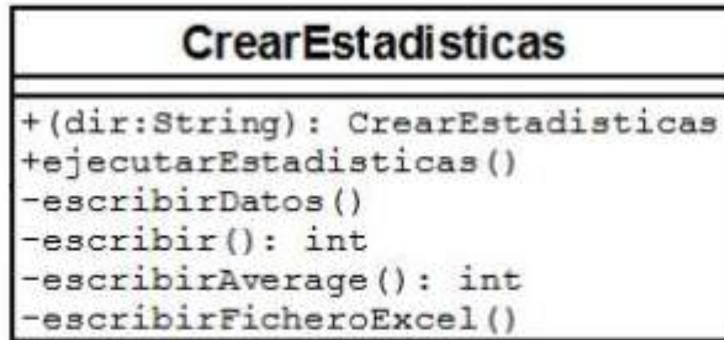

Figura 4.29. Clase `CrearEstadisticas`.

Constructor

Cabecera: `public CrearEstadisticas(String dir)`
Objetivos: Devolver un objeto de esta clase.
Entrada: El método recibe *`String`*.
Salida: El método devuelve un objeto de la clase `CrearEstadisticas`.
Descripción: El método recibe un *`String`* que contiene la ruta de un directorio, y se crea un objeto de esta clase con ese dato.

EjecutarEstadisticas

Cabecera: `public void ejecutarEstadisticas()`
Objetivos: Ejecutar las combinaciones algorítmicas sobre los ficheros del directorio dado y escribir los resultados en un fichero Excel.
Entrada: El método no recibe parámetros.
Salida: El método no devuelve valores.
Descripción: El método ejecuta sobre todas las instancias del directorio los algoritmos constructivos, los algoritmos constructivos con la búsqueda local dos por uno, los algoritmos constructivos con la búsqueda local uno por cero y por último los algoritmos constructivos con la búsqueda dos por uno manual. Estos datos los almacena en variables que finalmente escribe en el fichero Excel.

EscribirDatos

Cabecera: `private void escribirDatos(Solucion solucionAC1R,Solucion solucionAC1RBL1,Solucion solucionAC1RBL2,Solucion solucionAC1RBL3, Solucion solucionAH1R,Solucion solucionAH1RBL1,Solucion solucionAH1RBL2,Solucion solucionAH1RBL3, Solucion solucionAH2R,Solucion solucionAH2RBL1,Solucion solucionAH2RBL2,Solucion solucionAH2RBL3)`
Objetivos: Escribir los resultados de las soluciones en las variables que posteriormente se escribirán en el fichero Excel.
Entrada: El método recibe las doce soluciones de las principales combinaciones algorítmicas.
Salida: El método no devuelve valores.
Descripción: El método recibe las principales soluciones para una instancia y escribe la información importante (número de regeneradores y el tiempo de ejecución) en las variables que en otros métodos auxiliares de la clase escribirán en el fichero Excel.

Escribir

Cabecera: `private int escribir(int indiceFila,String tituloAlgoritmo,WritableSheet hoja, List<Integer> numReg,List<Long> CPUTime)`
Objetivos: Escribir los resultados de cada una de las combinaciones algorítmicas.
Entrada: El método un entero, un *String*, un objeto de la clase `WritableSheet` y dos listas, una de enteros y otra de *longs*.
Salida: El método devuelve un entero.
Descripción: El método recibe un entero que indica la fila o la posición de la hoja Excel (objeto `WritableSheet`) en la que debe escribir. Recibe el titulo del algoritmo, es decir, de la combinación algorítmica (AlgoritmoConstructivo1Random, AlgoritmoConstructivo1Random+BL1,…) de la que se van a escribir los resultados, que se reciben en las dos listas, una de regeneradores y otra de tiempos de ejecución.

EscribirAverage

Cabecera: `private int escribirAverage(int indiceFila,String tituloAlgoritmo,List<Integer> numReg,List<Long> CPUTime,WritableSheet pagina1)`
Objetivos: Escribir un cuadro mostrando una media de los resultados obtenidos para cada configuración.
Entrada: El método un entero, un String, un objeto de la clase WritableSheet y dos listas, una de enteros y otra de *longs*.
Salida: El método devuelve un entero.

Descripción: El método recibe un entero que indica la fila o la posición de la hoja Excel (objeto WritableSheet) en la que debe escribir. Recibe el titulo del algoritmo, es decir, de la combinación algorítmica (AlgoritmoConstructivo1Random, AlgoritmoConstructivo1Random+BL1,…) de la que se van a escribir los resultados medios, que se calcularán gracias a los datos recibidos en las dos listas, una de regeneradores y otra de tiempos de ejecución.

EscribirFicheroExcel

Cabecera: `private void escribirFicheroExcel()`
Objetivos: Escribir el fichero Excel una vez que se tienen los datos de las ejecuciones.
Entrada: El método no recibe parámetros.
Salida: El método no devuelve valores.
Descripción: Este método es invocado por `ejecutarEstaditicas()` una vez se han calculado los datos de los algoritmos y usa los métodos `escribir()` y `escribirAverage()` para conseguir su propósito. Que no es otro que escribir de manera formateada los resultados obtenidos en un fichero Excel.

4.6.22. Clase Main

La clase `Main` es la directora de todas las demás clases, es la que realiza las pruebas utilizando `crearEstadisticas()` o la que utiliza las clases que describen algoritmos. Es la clase que gobierna los experimentos.

Figura 4.30. Clase `Main`.

Capítulo 5

Experimentos

En este Capítulo se exponen los resultados más relevantes obtenidos sobre las instancias con las que se ha contado para el estudio.

Se ha utilizado el lenguaje de paradigma orientado a objetos Java. Para implementar las clases y módulos del proyecto se ha utilizado la plataforma de libre distribución NetBeans IDE 6.0.1.

El proyecto ha sido ejecutado en un computador que utiliza el Sistema Operativo Windows Vista Home Premium con Service Pack 2. Algunas especificaciones del sistema:

- Procesador: Intel Core 2 Duo P8600 a 2.40GHz.
- Memoria RAM: 4 GB.
- Disco duro 286 GB.
- Sistema operativo de 32 bits.

Las instancias tienen nombres del tipo dataN40P1Inst1, dataN40P1Inst2,… dataN40P1Inst10. Significa, utilizando este ejemplo, que la instancia tiene 40 nodos (N40), a mayor número de planta (P1, P2,…, P9) hay más número de nodos desconectados en el grafo y las instancias (Inst1, Inst2,…, Inst10) varían los nodos que están conectados o desconectados.

5.1. Resultados

Dado el número de algoritmos desarrollados, las posibles combinaciones entre ellos y el hecho de poder aplicar cada una de estas combinaciones a cada instancia se podrían realizar infinidad de pueblas, se ha elegido una configuración de algoritmos ligera y representativa para aportar datos en este apartado.

Las tablas resumen muestran en su primera columna la combinación de algoritmos utilizada. La segunda columna muestra la media de los regeneradores que se utilizan en

las instancias. La tercera columna muestra el mejor resultado encontrado. La cuarta la desviación, es un parámetro que indica la variabilidad en las soluciones obtenidas, a más valor mayor variabilidad en las soluciones se calcula con la siguiente fórmula:

$$Desv = \frac{(media - mejor)}{mejor} * 100$$

La última columna es el *CPUTime* medio obtenido de todas las ejecuciones expresado en milisegundos.

Los algoritmos constructivos se han ejecutado 10 veces sobre cada instancia en las instancias de 40, 60 y 80 nodos, y 6 veces en las instancias de 100 nodos. El parámetro α de los algoritmos constructivos siempre tiene el valor 0.5. Los algoritmos metaheurísticos GRASP y GRASP-PR siempre se ejecutan 3 veces sobre cada instancia.

En primer lugar se van a mostrar las tablas resumen de los algoritmos constructivos sobre todas las instancias

La Tabla 5.1. muestra la tabla resumen de los resultados obtenidos por los algoritmos constructivos en las instancias de 40 nodos.

N40	Media	Mejor	Desviación	Tiempo(ms)
C1R	4,457143	1	345,7143	455
H1R	4,628571	1	362,8571	41
H2R	4,371429	1	337,1429	28

Tabla 5.1. Tabla resumen de los algoritmos constructivos en las instancias N40.

La Tabla 5.2. muestra la tabla resumen de los resultados obtenidos por los algoritmos constructivos en las instancias de 60 nodos.

N60	Media	Mejor	Desviación	CPUTime
C1R	5,142857	2	157,1429	3874
H1R	5,414286	1	441,4286	212
H2R	5,214286	2	160,7143	112

Tabla 5.2. Tabla resumen de los algoritmos constructivos en las instancias N60.

La Tabla 5.3. muestra la tabla resumen de los resultados obtenidos por los algoritmos constructivos en las instancias de 80 nodos.

N80	Media	Mejor	Desviación	CPUTime
C1R	5,685714	2	184,2857	15455
H1R	5,985714	2	199,2857	645
H2R	5,914286	1	491,4286	373

Tabla 5.3. Tabla resumen de los algoritmos constructivos en las instancias N80.

La Tabla 5.4. muestra la tabla resumen de los resultados obtenidos por los algoritmos constructivos en las instancias de 100 nodos.

N100	Media	Mejor	Desviación	CPUTime
C1R	6,471429	2	223,5714	25863
H1R	6,742857	2	237,1429	636
H2R	6,657143	2	232,8571	402

Tabla 5.4. Tabla resumen de los algoritmos constructivos en las instancias N100.

El algoritmo constructivo que por lo general obtiene una mejor media de resultados es el *C1R*, esto es debido a que la idea del algoritmo es una idea intuitiva pero muy eficaz. Colocar los regeneradores en el nodo que más vecinos tenga, porque de esta manera se generará un mayor número de nuevas conexiones entre estos vecinos. Sin embargo, no es tan eficiente, se puede apreciar en las tablas que el tiempo medio se dispara en este algoritmo en comparación con los demás. Si se tiene en cuenta este parámetro el algoritmo *H2R* tarda mucho menos en ejecutarse y consigue resultados casi tan buenos como el algoritmo *C1R*.

El próximo algoritmo es el metaheurístico GRASP. Combina los algoritmos constructivos con el algoritmo de búsqueda local *LS2x1* para todas las instancias.

La Tabla 5.5. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS2x1* en las instancias de 40 nodos.

N40	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS2x1)	3,828571	1	282,8571	1451
GRASP2(H1R+LS2x1)	3,842857	1	284,2857	148
GRASP3(H2R+LS2x1)	3,885714	1	288,5714	95

Tabla 5.5. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS2x1*) en las instancias N40.

La Tabla 5.6. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS2x1* en las instancias de 60 nodos.

N60	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS2x1)	4,4	1	340	12167
GRASP2(H1R+LS2x1)	4,685714	1	368,5714	736
GRASP3(H2R+LS2x1)	4,428571	2	121,4286	463

Tabla 5.6. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS2x1*) en las instancias N60.

La Tabla 5.7. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS2x1* en las instancias de 80 nodos.

N80	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS2x1)	4,957143	1	395,7143	48892
GRASP2(H1R+LS2x1)	5,357143	1	435,7143	2024
GRASP3(H2R+LS2x1)	5,142857	1	414,2857	1291

Tabla 5.7. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS2x1*) en las instancias N80.

La Tabla 5.8. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS2x1* en las instancias de 100 nodos.

N100	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS2x1)	5,428571	2	171,4286	78884
GRASP2(H1R+LS2x1)	5,828571	2	191,4286	2286
GRASP3(H2R+LS2x1)	5,742857	2	187,1429	1508

Tabla 5.8. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS2x1*) en las instancias N100.

En el momento en el que utilizan algoritmos metaheurísticos se observa que la media de las soluciones obtenidas mejora y el mejor resultado encontrado se halla en una diversidad mayor de algoritmos. Por ejemplo, en la Tabla 5.2 el algoritmo *C1R* obtenía como mejor solución el valor 2, y si lo combinamos con el algoritmo de optimización *LS2x1*, se encuentra como mejor valor el 1, tal y como se muestra en la Tabla 5.6.

El próximo algoritmo es GRASP combina los algoritmos constructivos con el algoritmo de búsqueda local *LS1x0* para todas las instancias.

La Tabla 5.9. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS1x0* en las instancias de 40 nodos.

N40	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS1x0)	3,957143	1	295,7143	1533
GRASP2(H1R+LS1x0)	4,114286	1	311,4286	151
GRASP3(H2R+LS1x0)	3,957143	1	295,7143	94

Tabla 5.9. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS1x0*) en las instancias N40.

La Tabla 5.10. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS1x0* en las instancias de 60 nodos.

N60	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS1x0)	4,857143	1	385,7143	12459
GRASP2(H1R+LS1x0)	5,1	1	410	707
GRASP3(H2R+LS1x0)	4,757143	1	375,7143	538

Tabla 5.10. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS1x0*) en las instancias N60.

La Tabla 5.11. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS1x0* en las instancias de 80 nodos.

N80	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS1x0)	5,342857	2	167,1429	48638
GRASP2(H1R+LS1x0)	5,614286	2	180,7143	2232
GRASP3(H2R+LS1x0)	5,385714	1	438,5714	1504

Tabla 5.11. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS1x0*) en las instancias N80.

La Tabla 5.12. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP con los algoritmos constructivos y la búsqueda local *LS1x0* en las instancias de 100 nodos.

N100	Media	Mejor	Desviación	Tiempo(ms)
GRASP1(C1R+LS1x0)	5,9	2	195	80066
GRASP2(H1R+LS1x0)	6,342857	2	217,1429	3502
GRASP3(H2R+LS1x0)	5,971429	2	198,5714	3959

Tabla 5.12. Tabla resumen del GRASP (Constructivos + Búsqueda local *LS1x0*) en las instancias N100.

En este caso las conclusiones son parecidas al caso anterior, se obtienen mejores resultados utilizando el algoritmo de optimización *LS1x0* que únicamente algoritmos constructivos. Sin embargo, se puede apreciar que en general, el algoritmo *LS2x1* obtiene mejores resultados que el algoritmo *LS1x0*, ya que busca nuevos nodos candidatos a albergar un regenerador en detrimento de dos regeneradores del conjunto solución y el algoritmo *LS1x0* simplemente comprueba si quitando un regenerador de un nodo del conjunto solución la solución se mantiene factible.

El próximo algoritmo es el metaheurístico GRASP-PR. Combina los algoritmos constructivos con el algoritmo de búsqueda local *LS2x1* y después entre dos de las mejores soluciones halladas se ejecuta el algoritmo de optimización *PR*.

La Tabla 5.13. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP-PR con los algoritmos constructivos, la búsqueda local *LS2x1* y el re-encaminamiento *PR1* en las instancias de 40 nodos.

N40	Media	Mejor	Desviación	Tiempo(ms)
GRASP-PR(AC1R+LS2x1+PR1)	3,928571	1	292,8571	1435
GRASP-PR(AH1R+ LS2x1+PR1)	3,885714	1	288,5714	203
GRASP-PR(AH2R+ LS2x1+PR1)	3,814286	1	281,4286	144

Tabla 5.13. Tabla resumen del GRASP-PR (Constructivos + Búsqueda local *LS2x1* + Re-encaminamiento *PR1*) en las instancias N40.

La Tabla 5.14. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP-PR con los algoritmos constructivos, la búsqueda local *LS2x1* y el re-encaminamiento *PR1* en las instancias de 60 nodos.

N60	Media	Mejor	Desviación	Tiempo(ms)
GRASP-PR(AC1R+ LS2x1+PR1)	4,542857	2	127,1429	12653
GRASP-PR(AH1R+ LS2x1+PR1)	4,428571	1	342,8571	1080
GRASP-PR(AH2R+ LS2x1+PR1)	4,442857	1	344,2857	978

Tabla 5.14. Tabla resumen del GRASP-PR (Constructivos + Búsqueda local *LS2x1* + Re-encaminamiento *PR1*) en las instancias N60.

La Tabla 5.15. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP-PR con los algoritmos constructivos, la búsqueda local *LS2x1* y el re-encaminamiento *PR1* en las instancias de 80 nodos.

N80	Media	Mejor	Desviación	Tiempo(ms)
GRASP-PR(AC1R+ LS2x1+PR1)	5,142857	2	157,1429	49594
GRASP-PR(AH1R+ LS2x1+PR1)	4,957143	1	395,7143	3137
GRASP-PR(AH2R+ LS2x1+PR1)	4,985714	1	398,5714	2917

Tabla 5.15. Tabla resumen del GRASP-PR (Constructivos + Búsqueda local *LS2x1* + Re-encaminamiento *PR1*) en las instancias N80.

La Tabla 5.16. muestra la tabla resumen de los resultados obtenidos por el algoritmo GRASP-PR con los algoritmos constructivos, la búsqueda local *LS2x1* y el re-encaminamiento *PR1* en las instancias de 100 nodos.

N100	Media	Mejor	Desviación	Tiempo(ms)
GRASP-PR(AC1R+ LS2x1+PR1)	5,5	2	175	83957
GRASP-PR(AH1R+ LS2x1+PR1)	5,557143	2	177,8571	6505
GRASP-PR(AH2R+ LS2x1+PR1)	5,485714	2	174,2857	6623

Tabla 5.16. Tabla resumen del GRASP-PR (Constructivos + Búsqueda local *LS2x1* + Re-encaminamiento *PR1*) en las instancias N100.

Después de ejecutar los algoritmos GRASP-PR sobre todas las instancias se puede concluir que es una metaheurística que consigue de los mejores tiempos encontrados. De hecho, supera en media al algoritmo GRASP (algoritmos constructivos + algoritmo de búsqueda local *LS2x1*) y GRASP (algoritmos constructivos + algoritmo de búsqueda local *LS1x0*) como se muestra comparando la Tabla 5.16 (GRASP-PR) con las Tablas 5.8 y 5.12 respectivamente. Esto es debido a que el algoritmo GRASP-PR utiliza las dos técnicas de optimización y en instancias con muchos nodos es más probable que se mejoren las soluciones construidas utilizando los dos algoritmos de optimización en lugar de uno solo.

En las plantas de 100 nodos se puede apreciar algo nuevo, el mejor no varía en ningún caso, esto puede ser debido a que se debería iterar más veces en los algoritmos constructivos, GRASP y GRASP-PR para encontrar un mayor rango de soluciones en dónde posiblemente se encontrasen soluciones mejores, o simplemente que las soluciones que se han obtenido no eran mejorables.

También se ha observado que utilizando el algoritmo de búsqueda local *LS 2x1 manual* en lugar del algoritmo heurístico de búsqueda local *LS 2x1* se tarda de media un 35% más, lo que hace que se pueda valorar mejorar la eficiencia del algoritmo de búsqueda local *LS 2x1*.

Capitulo 6

Conclusiones y trabajos futuros

Este Capítulo presenta las conclusiones obtenidas durante todo el proceso de realización del estudio y en qué áreas se puede seguir investigando.

6.1. Conclusiones

En primer lugar y hablando desde un punto académico para la formación del autor del proyecto se puede decir que ha sido muy beneficioso. Se han adquirido nuevos conocimientos y mayor destreza en el lenguaje de programación Java. Se han adquirido conocimientos sobre diferentes familias de algoritmos como son los algoritmos heurísticos y metaheurísticos, aplicándolos a un problema real, estudiando las posibilidades de éxito de los mismos y los resultados devueltos. Sin lugar a dudas se considera un complemento adecuado a la formación recibida previamente.

En cuanto a la elaboración del proyecto, creo que se han cumplido los objetivos marcados. Se han desarrollado diferentes algoritmos heurísticos que construyen soluciones realmente válidas y en muchas ocasiones óptimas.

Para intentar mejorar ese porcentaje de soluciones no óptimas que se consiguen de los algoritmos heurísticos constructivos, se han implementado diferentes algoritmos heurísticos de mejora. Gracias a los resultados obtenidos se ha comprobado que son realmente útiles y un complemento ideal a los algoritmos constructivos.

Por último, se han conseguido crear algoritmos más inteligentes que utilizan a los explicados anteriormente. Los algoritmos metaheurísticos consiguen explorar un mayor ámbito de posibles soluciones y devolver soluciones de mayor calidad a los problemas dados, sobre todo en las instancias de mayor número de nodos.

En cuanto a los resultados, también se han cumplido las expectativas. Dado el inmenso número de instancias de grafos con las que se ha podido trabajar, y las diferentes situaciones que planteaban entre sí, se ha conseguido obtener la solución óptima o soluciones muy buenas en todas las instancias estudiadas, independientemente de las

características de éstas. Además se ha logrado presentar estos resultados en un fichero Excel con un formato adecuado, que facilita la lectura, comprensión y análisis de los resultados.

6.2. Trabajos futuros

Algunos apartados y posibles líneas de investigación futuras son las siguientes:

- Creación de nuevos algoritmos heurísticos constructivos.
- Creación de nuevos algoritmos heurísticos de optimización.
- Creación de nuevos algoritmos metaheurísticos para obtener soluciones.
- Diseñar una interfaz gráfica que muestre los resultados obtenidos visualizando el grafo, los nodos con regeneradores…

Bibliografía

- "Metaheurísticas", Abraham Duarte Muñoz, Juan José Pantrigo Fernández, Micael Gallego Carrillo.

- "El lenguaje de programación Java", 3ª Edición, Arnold Gosling, Addison Wesley, 2001.

- "Piensa en Java", Eckel, 2ª Edición, Addison Wesley, 2002.

- "Introducción a la programación orientada a objetos con Java", C. Thomas Wu, Mc Graw Hill, 2001.

- "Técnicas avanzadas de diseño de software: Orientación a objetos, UML, patrones de diseño y Java", José F. Vélez Serrano, Ángel Sánchez Calle, Alfredo Casado Bernárdez, Santiago Doblas Álvarez.

- "Ingeniería del Software: un enfoque práctico", Pressman, McGraw-Hill, 2002 5ª Ed.

- "Ingeniería del Software", Ian Sommermille, Addison Wesley, 2004ª Ed.

- "Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión", Piattini et al., RA-MA, 1996.

- Página Web: http://www.netbeans.org/community/releases/60/.

- Página Web: http://www.wikipedia.org/.

- Página Web: http://www.gnome.org/projects/dia/.

Anexo

Se adjunta este anexo para que se pueda apreciar de una manera resumida los mejores resultados obtenidos para cada una de las instancias que se han estudiado.

Nombre fichero	Nº de Regeneradores	CPUTime (ms)
agrafo.txt	2	0
agrafo2.txt	2	0
agrafo3.txt	2	0
dataN40P1Inst1.txt	1	265
dataN40P1Inst10.txt	2	156
dataN40P1Inst2.txt	1	156
dataN40P1Inst3.txt	2	156
dataN40P1Inst4.txt	2	141
dataN40P1Inst5.txt	1	140
dataN40P1Inst6.txt	1	141
dataN40P1Inst7.txt	1	125
dataN40P1Inst8.txt	2	125
dataN40P1Inst9.txt	1	0
dataN40P2Inst1.txt	2	47
dataN40P2Inst2.txt	2	46
dataN40P2Inst3.txt	2	46
dataN40P2Inst4.txt	2	31
dataN40P2Inst5.txt	2	31
dataN40P3Inst1.txt	2	1014
dataN40P3Inst10.txt	2	1014
dataN40P3Inst2.txt	2	983
dataN40P3Inst3.txt	2	983
dataN40P3Inst4.txt	2	983
dataN40P3Inst5.txt	2	967
dataN40P3Inst6.txt	2	998
dataN40P3Inst7.txt	2	1030
dataN40P3Inst8.txt	2	1108
dataN40P3Inst9.txt	2	983
dataN40P4Inst1.txt	2	94
dataN40P4Inst2.txt	2	1124
dataN40P4Inst3.txt	2	94
dataN40P4Inst4.txt	3	78
dataN40P4Inst5.txt	3	94
dataN40P5Inst1.txt	3	94
dataN40P5Inst10.txt	3	94

dataN40P5Inst2.txt	3	94
dataN40P5Inst3.txt	3	93
dataN40P5Inst4.txt	3	94
dataN40P5Inst5.txt	3	94
dataN40P5Inst6.txt	3	78
dataN40P5Inst7.txt	3	78
dataN40P5Inst8.txt	3	94
dataN40P5Inst9.txt	3	94
dataN40P6Inst1.txt	4	78
dataN40P6Inst2.txt	3	1248
dataN40P6Inst3.txt	4	78
dataN40P6Inst4.txt	3	78
dataN40P6Inst5.txt	3	1248
dataN40P7Inst1.txt	4	94
dataN40P7Inst10.txt	4	1497
dataN40P7Inst2.txt	4	93
dataN40P7Inst3.txt	4	78
dataN40P7Inst4.txt	4	110
dataN40P7Inst5.txt	4	94
dataN40P7Inst6.txt	5	93
dataN40P7Inst7.txt	5	93
dataN40P7Inst8.txt	4	171
dataN40P7Inst9.txt	5	125
dataN40P8Inst1.txt	6	125
dataN40P8Inst2.txt	6	203
dataN40P8Inst3.txt	5	219
dataN40P8Inst4.txt	6	203
dataN40P8Inst5.txt	5	125
dataN40P9Inst1.txt	8	452
dataN40P9Inst10.txt	9	109
dataN40P9Inst2.txt	9	811
dataN40P9Inst3.txt	9	125
dataN40P9Inst4.txt	9	187
dataN40P9Inst5.txt	9	297
dataN40P9Inst6.txt	8	110
dataN40P9Inst7.txt	8	109
dataN40P9Inst8.txt	9	125
dataN40P9Inst9.txt	9	1170
dataN60P1Inst1.txt	2	421
dataN60P1Inst10.txt	2	437
dataN60P1Inst2.txt	2	437
dataN60P1Inst3.txt	2	452
dataN60P1Inst4.txt	1	452
dataN60P1Inst5.txt	2	468
dataN60P1Inst6.txt	2	515
dataN60P1Inst7.txt	2	437
dataN60P1Inst8.txt	2	765

dataN60P1Inst9.txt	2	468
dataN60P2Inst1.txt	2	1216
dataN60P2Inst2.txt	2	374
dataN60P2Inst3.txt	2	1934
dataN60P2Inst4.txt	2	436
dataN60P2Inst5.txt	2	375
dataN60P3Inst1.txt	2	390
dataN60P3Inst10.txt	2	344
dataN60P3Inst2.txt	2	327
dataN60P3Inst3.txt	2	343
dataN60P3Inst4.txt	2	343
dataN60P3Inst5.txt	2	5179
dataN60P3Inst6.txt	2	328
dataN60P3Inst7.txt	2	343
dataN60P3Inst8.txt	2	328
dataN60P3Inst9.txt	2	344
dataN60P4Inst1.txt	3	312
dataN60P4Inst2.txt	3	281
dataN60P4Inst3.txt	3	297
dataN60P4Inst4.txt	3	1529
dataN60P4Inst5.txt	3	1232
dataN60P5Inst1.txt	3	3932
dataN60P5Inst10.txt	3	1466
dataN60P5Inst2.txt	3	280
dataN60P5Inst3.txt	3	265
dataN60P5Inst4.txt	3	266
dataN60P5Inst5.txt	3	281
dataN60P5Inst6.txt	3	296
dataN60P5Inst7.txt	3	266
dataN60P5Inst8.txt	3	281
dataN60P5Inst9.txt	3	281
dataN60P6Inst1.txt	4	280
dataN60P6Inst2.txt	4	281
dataN60P6Inst3.txt	4	281
dataN60P6Inst4.txt	4	297
dataN60P6Inst5.txt	4	2356
dataN60P7Inst1.txt	5	327
dataN60P7Inst10.txt	5	297
dataN60P7Inst2.txt	5	1840
dataN60P7Inst3.txt	5	327
dataN60P7Inst4.txt	5	296
dataN60P7Inst5.txt	4	296
dataN60P7Inst6.txt	5	296
dataN60P7Inst7.txt	5	296
dataN60P7Inst8.txt	5	328
dataN60P7Inst9.txt	5	312
dataN60P8Inst1.txt	7	327

dataN60P8Inst2.txt	6	9032
dataN60P8Inst3.txt	6	13057
dataN60P8Inst4.txt	7	4899
dataN60P8Inst5.txt	7	405
dataN60P9Inst1.txt	10	468
dataN60P9Inst10.txt	11	406
dataN60P9Inst2.txt	11	312
dataN60P9Inst3.txt	10	515
dataN60P9Inst4.txt	11	390
dataN60P9Inst5.txt	10	359
dataN60P9Inst6.txt	10	406
dataN60P9Inst7.txt	10	1107
dataN60P9Inst8.txt	11	546
dataN60P9Inst9.txt	11	421
dataN80P1Inst1.txt	2	1092
dataN80P1Inst10.txt	2	1061
dataN80P1Inst2.txt	1	1108
dataN80P1Inst3.txt	2	1077
dataN80P1Inst4.txt	2	1092
dataN80P1Inst5.txt	2	2418
dataN80P1Inst6.txt	2	3401
dataN80P1Inst7.txt	2	3713
dataN80P1Inst8.txt	2	1076
dataN80P1Inst9.txt	2	1092
dataN80P2Inst1.txt	2	951
dataN80P2Inst2.txt	2	936
dataN80P2Inst3.txt	2	1217
dataN80P2Inst4.txt	2	874
dataN80P2Inst5.txt	2	858
dataN80P3Inst1.txt	2	6193
dataN80P3Inst10.txt	2	36770
dataN80P3Inst2.txt	2	952
dataN80P3Inst3.txt	2	38032
dataN80P3Inst4.txt	3	780
dataN80P3Inst5.txt	3	795
dataN80P3Inst6.txt	3	765
dataN80P3Inst7.txt	2	27300
dataN80P3Inst8.txt	2	4415
dataN80P3Inst9.txt	2	7878
dataN80P4Inst1.txt	3	905
dataN80P4Inst2.txt	3	874
dataN80P4Inst3.txt	3	702
dataN80P4Inst4.txt	3	733
dataN80P4Inst5.txt	3	687
dataN80P5Inst1.txt	3	5397
dataN80P5Inst10.txt	3	499
dataN80P5Inst2.txt	3	2699

dataN80P5Inst3.txt	3	1451
dataN80P5Inst4.txt	3	54413
dataN80P5Inst5.txt	3	780
dataN80P5Inst6.txt	4	780
dataN80P5Inst7.txt	4	624
dataN80P5Inst8.txt	3	56441
dataN80P5Inst9.txt	4	686
dataN80P6Inst1.txt	4	827
dataN80P6Inst2.txt	4	42292
dataN80P6Inst3.txt	4	52385
dataN80P6Inst4.txt	4	4820
dataN80P6Inst5.txt	4	58532
dataN80P7Inst1.txt	5	733
dataN80P7Inst10.txt	5	2403
dataN80P7Inst2.txt	5	858
dataN80P7Inst3.txt	6	889
dataN80P7Inst4.txt	6	936
dataN80P7Inst5.txt	5	2199
dataN80P7Inst6.txt	5	1497
dataN80P7Inst7.txt	6	3915
dataN80P7Inst8.txt	5	45786
dataN80P7Inst9.txt	5	795
dataN80P8Inst1.txt	7	998
dataN80P8Inst2.txt	7	14742
dataN80P8Inst3.txt	7	1794
dataN80P8Inst4.txt	7	76003
dataN80P8Inst5.txt	8	1435
dataN80P9Inst1.txt	13	95972
dataN80P9Inst10.txt	12	80762
dataN80P9Inst2.txt	10	79419
dataN80P9Inst3.txt	12	94286
dataN80P9Inst4.txt	12	67829
dataN80P9Inst5.txt	11	86564
dataN80P9Inst6.txt	13	83024
dataN80P9Inst7.txt	12	87282
dataN80P9Inst8.txt	11	100074
dataN80P9Inst9.txt	12	2231
dataN100P1Inst1.txt	2	1201
dataN100P1Inst10.txt	2	1185
dataN100P1Inst2.txt	2	1170
dataN100P1Inst3.txt	2	1154
dataN100P1Inst4.txt	2	1326
dataN100P1Inst5.txt	2	1248
dataN100P1Inst6.txt	2	1217
dataN100P1Inst7.txt	2	1280
dataN100P1Inst8.txt	2	1241

dataN100P1Inst9.txt	2	1258
dataN100P2Inst1.txt	2	1560
dataN100P2Inst2.txt	2	2543
dataN100P2Inst3.txt	2	1060
dataN100P2Inst4.txt	2	1092
dataN100P2Inst5.txt	2	1027
dataN100P3Inst1.txt	3	889
dataN100P3Inst10.txt	3	1373
dataN100P3Inst2.txt	3	1466
dataN100P3Inst3.txt	3	1467
dataN100P3Inst4.txt	3	1029
dataN100P3Inst5.txt	2	43087
dataN100P3Inst6.txt	3	1654
dataN100P3Inst7.txt	3	2886
dataN100P3Inst8.txt	2	998
dataN100P3Inst9.txt	2	48204
dataN100P4Inst1.txt	3	10390
dataN100P4Inst2.txt	3	6318
dataN100P4Inst3.txt	3	858
dataN100P4Inst4.txt	3	10613
dataN100P4Inst5.txt	3	8331
dataN100P5Inst1.txt	4	918
dataN100P5Inst10.txt	4	837
dataN100P5Inst2.txt	4	877
dataN100P5Inst3.txt	4	952
dataN100P5Inst4.txt	4	10437
dataN100P5Inst5.txt	4	982
dataN100P5Inst6.txt	4	998
dataN100P5Inst7.txt	4	1233
dataN100P5Inst8.txt	4	936
dataN100P5Inst9.txt	4	1045
dataN100P6Inst1.txt	5	1014
dataN100P6Inst2.txt	4	12168
dataN100P6Inst3.txt	5	1264
dataN100P6Inst4.txt	5	874
dataN100P6Inst5.txt	4	80324
dataN100P7Inst1.txt	6	2168
dataN100P7Inst10.txt	5	85145
dataN100P7Inst2.txt	6	1575
dataN100P7Inst3.txt	6	1358
dataN100P7Inst4.txt	5	122912
dataN100P7Inst5.txt	6	135000
dataN100P7Inst6.txt	6	1373
dataN100P7Inst7.txt	6	1107
dataN100P7Inst8.txt	6	1685
dataN100P7Inst9.txt	5	16146
dataN100P8Inst1.txt	8	2543

dataN100P8Inst2.txt	8	1700
dataN100P8Inst3.txt	7	125642
dataN100P8Inst4.txt	8	10733
dataN100P8Inst5.txt	8	2340
dataN100P9Inst1.txt	13	162708
dataN100P9Inst10.txt	15	141913
dataN100P9Inst2.txt	14	42869
dataN100P9Inst3.txt	14	158465
dataN100P9Inst4.txt	13	138482
dataN100P9Inst5.txt	14	138715
dataN100P9Inst6.txt	14	154970
dataN100P9Inst7.txt	15	161382
dataN100P9Inst8.txt	14	174782
dataN100P9Inst9.txt	13	159182