ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Curso Académico 2009/2010

Proyecto de Fin de Carrera

ALGORITMO HEURÍSTICO PARA EL PROBLEMA DEL VERTEX SEPARATION

Autor: Samuel Álvarez Mangas

Tutores: Abraham Duarte Muñoz
Eduardo García Pardo

Listado de Acrónimos

- AM: *Agile Modeling*.
- BLE: Búsqueda Local Exhaustiva.
- BLF: Búsqueda Local Factorizada.
- BLI: Búsqueda Local Inteligente.
- COTS: *Commercial Of The Shelf*.
- CVMaA: Constructivo Voraz de Mayor Adyacencia.
- CVMD: Constructivo Voraz de Menor Diferencia.
- CVMeA: Constructivo Voraz de Menor Adyacencia.
- CVMG: Constructivo Voraz de Menor Grado.
- FO: Función Objetivo.
- GRASP: *Greedy Randomize Adaptative Search Procedure*.
- OMT: *Object Modeling Technique*.
- OOSE: *Object Oriented Software Engineering*.
- PFC: Proyecto Final de Carrera.
- POO: Programación Orientada a Objetos.
- PRD: Path Relinking Dinámico.
- PRE: Path Relinking Estático.
- PU: Proceso Unificado.
- RCL: Lista de Candidatos Restringida.

- UML: Lenguaje Unificado de Modelado, del inglés *Unified Modeling Language*.
- VLSI: *Very Large Scale Integration*.
- XP: Programación Extrema, del inglés *eXtreme Programming*.

Agradecimientos

Este proyecto ha sido el resultado de muchas horas de esfuerzo. Horas que una serie de personas han logrado hacer más llevaderas.

En primer lugar debo agradecer a mis tutores, Abraham y Eduardo, la confianza depositada en mí para desarrollar este PFC. Además debo resaltar el apoyo que me han dado a lo largo del curso, el tiempo que me han dedicado y toda la ayuda prestada en la elaboración de este proyecto.

A mi familia, que me apoyado durante todo este tiempo. En especial a mi padre, por sus grandes consejos, no sólo para este proyecto, sino para la vida en general, lo que me ha permitido seguir siempre hacia adelante y no arrojar nunca la toalla.

A mis amigos de toda la vida, que han hecho que el desarrollo de este proyecto se hiciera más ameno.

A mis compañeros de Universidad, Carlos, Héctor, Jorge, Roberto, por hacerme pasar buenos momentos durante estos cinco años de carrera. En especial destacar a Michel, gran compañero y amigo, que ha estado siempre ahí cuando lo he necesitado.

A todos ellos, gracias.

Resumen

La temática de este Proyecto Fin de Carrera se enmarca dentro del área de la optimización. Particularmente se centra en la resolución de un problema específico: el problema *Vertex Separation*. El objetivo del proyecto es encontrar heurísticas que permitan abordar el problema de forma eficiente, generando soluciones de buena calidad.

Este proyecto no esta orientado al desarrollo de una aplicación de usuario, sino que está más cercano a un proyecto de investigación. En él, se hará una introducción al mundo de la optimización, donde se hablará sobre los distintos tipos de problemas de optimización, su complejidad algorítmica, las metodologías para abordarlos y el *software* existente para optimización.

Se hará una incursión en los métodos exactos y se comprobará, a través de la experimentación, que este tipo de métodos resultan ineficientes para la resolución de este problema, cuando el tamaño de la entrada es grande.

También se documentarán algunos de los métodos aproximados más relevantes. Por un lado se introducirán los métodos heurísticos, explicando en qué consisten, cuáles son sus ventajas y qué inconvenientes presentan. Por otro lado, se introducirá el concepto de técnica metaheurística, se explicarán las ventajas e inconvenientes de este tipo de técnicas y, a partir de los resultados experimentales, se mostrará como, este tipo de métodos, consiguen soluciones de muy buena calidad en un tiempo razonable.

Por último, se analizarán detalladamente los resultados obtenidos para determinar cuál de las técnicas propuestas, es la más eficaz.

Índice general

Listado de Acrónimos I

Agradecimientos III

Resumen V

1. Introducción 1

1.1. Optimización . . . 1

1.2. Problemas de optimización . . . 1

1.3. Técnicas de resolución . . . 3

1.3.1. Exactas . . . 3

1.3.2. Aproximadas . . . 4

1.3.3. Software de optimización . . . 6

1.4. Problema Vertex Separation . . . 7

1.4.1. Definición del problema . . . 8

1.4.2. Aplicaciones . . . 12

1.5. Propuesta . . . 13

2. Objetivos 17

2.1. Objetivos del Proyecto Fin de Carrera . . . 17

2.2. Objetivos personales . . . 18

3. Descripción Algorítmica 19

3.1. Algoritmos exactos . . . 19

3.1.1. Backtracking . . . 19

3.1.2. Backtracking con poda . . . 21

3.2. Algoritmos heurísticos . . . 22

3.2.1. Constructivo voraz . . . 22

3.2.2. Búsquedas locales . . . 23

3.3. Algoritmos metaheurísticos . . . 28

3.3.1. GRASP 28
3.3.2. Path Relinking 30

4. Descripción Informática 33
4.1. Introducción 33
4.1.1. Modelo de proceso 33
4.1.2. Tipos de modelo de proceso 33
4.1.3. Elección de la metodología 38
4.2. Programación Extrema 38
4.3. Adaptación de la metodología al PFC 39
4.3.1. Hitos del proyecto 39
4.4. Tecnología empleada 57
4.4.1. Java 57
4.4.2. Eclipse 58
4.4.3. LaTeX 59

5. Resultados experimentales 61
5.1. Descripción de los conjuntos de instancias 61
5.1.1. Harwellboeing 62
5.1.2. Small_Instances 62
5.1.3. Grafos_Aleatorios 63
5.1.4. Grid 64
5.2. Experimentación con métodos exactos 65
5.2.1. Experimentación Backtracking 65
5.2.2. Experimentación Backtracking con poda 65
5.3. Experimentación con métodos heurísticos 66
5.3.1. Experimentación con constructivos voraces 67
5.3.2. Experimentación con búsquedas locales 67
5.4. Experimentación con métodos metaheurísticos 75
5.4.1. Experimentación GRASP 75
5.4.2. Experimentación Path Relinking 77
5.4.3. Evolución experimental 78
5.4.4. Experimentación final 78

6. Conclusiones y trabajos futuros 81
6.1. Conclusiones 81
6.2. Trabajos futuros 82

A. Contenido del CD **83**

B. Resultados por instancia **85**

Bibliografía **90**

Índice de figuras

1.1. Relación entre los problemas P, NP, NP-Completos y NP-Duros 3
1.2. Algoritmo heurístico atrapado en un óptimo local 5
1.3. Instancia G 9
1.4. Ordenación Lineal(φ) 10
1.5. Cálculo de la Función Objetivo para i=0 10
1.6. Cálculo de la Función Objetivo para i=1 11
1.7. Cálculo de la Función Objetivo para i=2 11
1.8. Cálculo de la Función Objetivo para i=3 11
1.9. Esquema de técnicas de resolución 15

3.1. Orden de expansión de los nodos de un árbol de exploración por el algoritmo de *backtracking* 20
3.2. Orden de expansión de los nodos de un árbol de exploración por el algoritmo de *backtracking* con poda 21
3.3. Procedimientos de Búsqueda Local (Inserción e intercambio) 24
3.4. Búsqueda Local Factorizada vs Búsqueda Local Exhaustiva 26
3.5. Ejemplo de vértices críticos 27
3.6. Lista de candidatos restringida para problemas de minimización 29
3.7. Ejemplo del algoritmo Path Relinking. 31

4.1. Modelo de Proceso en Cascada. 34
4.2. Modelo Evolutivo. 35
4.3. Modelo de Proceso Mixto. 35
4.4. Modelo de Proceso Basado en Reutilización. 36
4.5. Modelo de Espiral. 36
4.6. Proceso Unificado. 37
4.7. Diseño correspondiente al primer hito del proyecto 42
4.8. Diseño correspondiente al hito dos, donde se especifica el *backtracking* con poda 44

4.9. Diseño correspondiente al hito tres, referente a los constructivos voraces . . 46
4.10. Diseño correspondiente al hito cuarto, referente a la BLE 48
4.11. Diseño correspondiente al hito quinto, referente a la BLF 49
4.12. Diseño correspondiente al hito sexto, referente a la BLI 51
4.13. Diseño correspondiente al hito séptimo, referente a la modificación del esquema de constructivos 52
4.14. Diseño correspondiente al hito séptimo, referente al esquema del método GRASP 54
4.15. Diseño correspondiente al hito octavo, referente al esquema del método Path Relinking 57

5.1. Ejemplo de Grid cuadrado 64

Capítulo 1

Introducción

En este capítulo se van a introducir algunos conceptos que se tratarán a lo largo de este Proyecto Fin de Carrera (PFC).

En primer lugar se hablará sobre el concepto de optimización, los diferentes problemas de optimización que existen y sobre las técnicas de resolución que se pueden aplicar para resolver este tipo de problemas. Por otra parte, se definirá el problema del Vertex Separation y se realizará una propuesta algorítmica para resolverlo.

1.1. Optimización

La optimización intenta dar respuesta a un tipo general de problemas donde se desea elegir la mejor solución entre un determinado conjunto de soluciones en un tiempo limitado [4].

1.2. Problemas de optimización

Un problema de optimización es un problema en el que hay varias posibles soluciones y alguna forma clara de comparación entre ellas, de manera que éste existe si y sólo si se dispone un conjunto de soluciones candidatas diferentes que pueden ser comparadas [19]. Los problemas de optimización se definen con tres elementos básicos:

- Una **Función Objetivo (FO)**, la cual queremos maximizar o minimizar dependiendo del tipo de problema.
- Un conjunto de **variables** que determinan el valor de la FO.
- Un conjunto de **restricciones** que delimitan los valores que pueden tomar las variables.

En resumen, un problema de optimización consiste en encontrar valores de las variables que minimicen o maximicen la FO, cumpliendo con un conjunto de restricciones.

En este tipo de problemas también hay que conocer y tener claros los conceptos de mínimo/máximo global y local de una función:

- **Mínimo Local**: Valor de una función que es menor que los valores de la función en puntos cercanos, pero que no es el menor de todos los valores.
- **Mínimo Global**: Valor de una función que es menor o igual a cualquier valor de la función. El mínimo global es el menor de todos los valores.
- **Máximo Local**: Valor de una función que es mayor que los valores de la función en puntos cercanos, pero que no es el mayor de todos los valores.
- **Máximo Global**: Valor de una función que es mayor o igual a cualquier valor de la función. El máximo global es el mayor de todos los valores.

Dependiendo de la complejidad computacional, los problemas de optimización se pueden clasificar en las siguientes clases:

- **Problemas P**: Son todos aquellos problemas de decisión para los cuales se tiene un algoritmo de resolución que se ejecuta en tiempo polinomial en una máquina determinista [19].
- **Problemas NP**: Está formada por todos aquellos problemas en los que, a pesar de que no se haya encontrado ningún algoritmo polinómico que obtenga una solución óptima, sí se puede saber en tiempo polinómico si un valor corresponde a una solución del problema [19].
- **Problemas NP-Completos**: Hace referencia a todos aquellos problemas NP que no tienen un algoritmo en tiempo polinómico que los resuelva, pero que sin embargo no se ha podido demostrar formalmente que no exista, aunque los matemáticos creen que realmente no existe [19].
- **Problemas NP-Duros**: Son problemas de igual o incluso mayor dificultad que los problemas NP-completos. Este tipo de problemas no es un subconjunto de los problemas NP, es decir, no existe un algoritmo polinómico que nos permita verificar una solución [19].

La relación entre estas clases de problemas se muestra en la Figura 1.1.

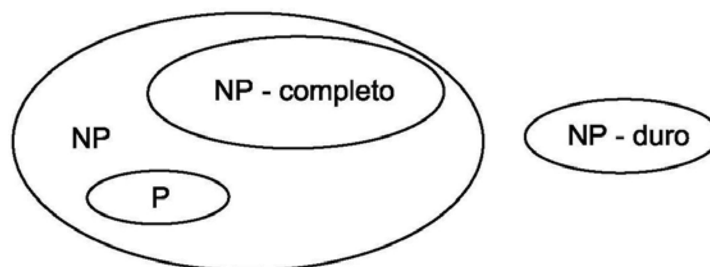

Figura 1.1: Relación entre los problemas P, NP, NP-Completos y NP-Duros

1.3. Técnicas de resolución

Existen dos tipos de técnicas para la resolución a problemas de optimización:

- **Exactas**
- **Aproximadas**

1.3.1. Exactas

Se utilizan para problemas cuya dificultad de cómputo es moderada o baja y, por lo tanto, es razonable exigir soluciones óptimas al problema [21]. Algunas de estas técnicas son:

- **Fuerza Bruta**: Consiste en generar todo el universo de candidatos y visitarlos uno por uno, para hallar la solución óptima al problema.

- **Backtracking**: Este algoritmo es una refinación del algoritmo de fuerza bruta. En el *backtracking* se explora todo el universo de candidatos para obtener la solución óptima. La diferencia principal con los algoritmos de fuerza bruta es que se explora de forma ordenada, lo cual permite, dado un conocimiento del problema, filtrar de antemano grandes porciones del espacio de candidatos. A este recorte se le denomina poda y puede mejorar mucho el rendimiento del algoritmo.

- **Branch&Bound**: Es una técnica muy similar al *backtracking*. Se encarga de generar un árbol de soluciones, pero introduce una estrategia de ramificación, que permite guiar la búsqueda por estimaciones de beneficio. Dichas estimaciones se calculan en cada nodo. Además, incluye una estrategia de poda, que permite eliminar nodos que no llevan a la solución óptima.

- **Divide y Vencerás**: Se basa en la idea de que es más fácil atacar un problema por partes, que hacerlo todo a la vez. Este algoritmo consta de tres etapas: dividir,

conquistar y combinar. Durante estas etapas lo que se hace es dividir una instancia del problema en dos o más instancias más pequeñas, que se resuelven (generalmente con una llamada recursiva) y luego se combinan sus soluciones para obtener la solución al problema original.

- **Programación Dinámica**: En muchos casos un problema no puede ser dividido en problemas disjuntos de forma útil para la utilización de un algoritmo de divide y vencerás. La técnica de programación dinámica se basa en el aprovechamiento del principio de optimalidad. Este principio dicta que la solución óptima de un problema contiene soluciones óptimas a problemas más pequeños.

- **Programación Lineal**: Es una técnica que consiste en expresar el problema como la maximización de una función lineal sujeta a una serie de restricciones lineales. El problema puede estar definido con variables continuas o enteras. Si el problema se logra expresar de forma concisa, existen algoritmos eficientes para resolver dichos problemas. Uno de los más conocidos es el algoritmo Simplex.

1.3.2. Aproximadas

Este tipo de técnicas se utilizan para resolver problemas en los cuales, encontrar la solución óptima en un tiempo razonable, es prácticamente imposible [19]. Sin embargo, consiguen soluciones de muy buena calidad en un tiempo razonable. Dentro de los métodos de resolución aproximados hay que distinguir dos grandes grupos:

- **Técnicas Heurísticas**

- **Técnicas Metaheurísticas**

Técnicas Heurísticas

Este tipo de técnicas están basadas en procedimientos conceptuales simples, que permiten encontrar soluciones de buena calidad (no necesariamente hallan la solución óptima) a problemas difíciles, de un modo sencillo y eficiente.

El término "Heurística" deriva del griego *heuriskein*, que significa encontrar o descubrir. Algunas de estas técnicas son:

- **Heurísticas constructivas**: Se encargan de agregar elementos a una solución con el objetivo de alcanzar una solución completa. El método constructivo finaliza cuando se ha completado la solución o se cumple con algún criterio de parada. Un ejemplo de

ello es el constructivo voraz (*greedy*) que trata de construir una solución seleccionando iterativamente aquellos elementos que tienen menor o mayor coste (dependiendo de si el problema propuesto es de maximización o de minimización).

- **Búsqueda local**: Obtiene mejores resultados que los algoritmos constructivos, sin embargo, requiere un mayor esfuerzo computacional. A partir de una solución inicial, iterativamente se reemplaza por una solución mejor, definida en una vecindad. Es decir, se encarga de buscar los mínimos o máximos locales, dependiendo del tipo de problema de optimización.

El principal problema que presentan los algoritmos heurísticos es su incapacidad para escapar de los óptimos locales. En la Figura 1.2 se muestra cómo, para una vecindad dada, el algoritmo heurístico basado en un método de búsqueda local se quedaría atrapado en un máximo local. Esto es debido a que los algoritmos heurísticos no poseen ningún mecanismo que le permita escapar de los óptimos locales.

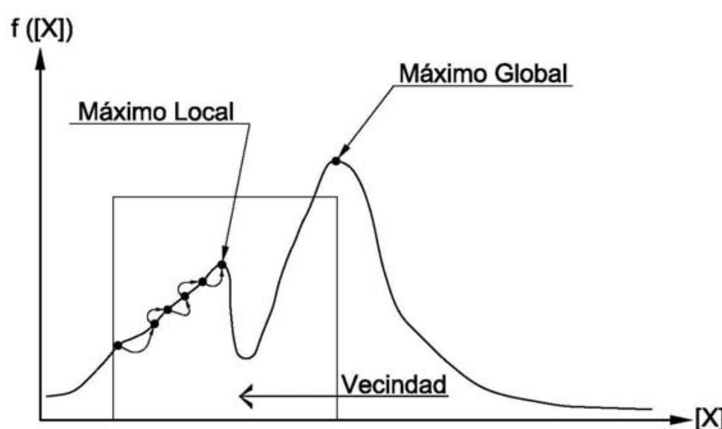

Figura 1.2: Algoritmo heurístico atrapado en un óptimo local

Técnicas Metaheurísticas

Consisten en heurísticas con un nivel de abstracción más alto. Las características que presentan son las siguientes:

- Guían el proceso de búsqueda incluyendo, en general, heurísticas subordinadas.
- Son de uso genérico (no son específicas para una clase de problema).
- Admiten descripciones a nivel abstracto.
- Deben instanciarse para cada clase de problema.

Los objetivos que persiguen son:

- Encontrar rápidamente soluciones factibles de buena calidad (cercanas al óptimo).
- Recorrer el espacio de soluciones sin quedar atrapados en una zona del espacio de soluciones.

A continuación se abordarán de manera genérica, algunas metaheurísticas bastante conocidas:

- **Greedy Randomized Adaptive Search Procedure (GRASP)**: Consiste en combinar un procedimiento constructivo *greedy* y una búsqueda local. Se divide en dos etapas: una de construcción de la solución y otra de mejora. Es una técnica aleatoria, que mantiene un único individuo, sin memoria y constructiva.
- **Path Relinking**: La idea del Path Relinking es buscar soluciones que compartan atributos con otras soluciones, con la esperanza de obtener mejores soluciones. Se parte de una solución inicial, se realizan los movimientos oportunos, generando soluciones intermedias hasta llegar a una solución final. Durante todo este trayecto se va almacenando aquella solución que tiene mejor FO. Es una técnica poblacional, sin memoria, y constructiva.
- **Búsqueda Tabú**: Consiste en una búsqueda local con memoria a corto plazo, y tiene como objetivo escapar de mínimos locales, evitando ciclos. Es una técnica determinista, que mantiene un único individuo, con memoria, y basada en trayectorias.
- **Algoritmos Genéticos**: Representan una metaheurística poblacional sencilla e intuitiva, que se basa en una población de soluciones candidatas, que evolucionan por medio de los mecanismos genéticos neodarwinistas de selección, cruce y mutación.

En el capítulo de descripción algorítmica, se detallará de forma más extensa las técnicas metaheurísticas utilizadas en este PFC, concretamente las técnicas GRASP y Path Relinking.

1.3.3. Software de optimización

El *software* de optimización consiste en programas que permiten la resolución de problemas de optimización [15]. Se presentan en variadas formas:

- Integrados en plantillas de calculo
- Precompiladores

- Lenguajes especializados

Una de sus principales ventajas es que son fáciles de integrar con otros sistemas. Algunos de estos *software* de optimización son:

- **KNITRO**: Basado en tres tipos de algoritmos de punto interior (directo, CG y *active set*) lo cual le permite afrontar casi cualquier tipo de problema no lineal [22].

- **CONOPT**: Especialmente ideado para problemas de gran dimensión y con restricciones que introducen una gran no linealidad. Basado en el algoritmo del gradiente reducido. Indicado cuando existe dificultad para hallar soluciones factibles [7].

- **MINOS**: Minimiza funciones sujetas a restricciones no lineales. Tiene su principal aplicación en problemas con pocos grados de libertad y con pocas restricciones no lineales [15].

- **BARON**: Basado en el algoritmo *Branch* & *Bound*. Permite obtener óptimos globales bajo ciertas condiciones formales del problema [1].

- **DICOPT**: Se basa en dividir un problema general en varios subproblemas [11].

- **CPLEX**: Es uno de los *software* de optimización más importantes y conocidos. Resuelve problemas de programación lineal, de redes y cuadráticos. Maneja un número ilimitado de restricciones y utiliza algoritmos como *Simplex* y *Branch* & *Bound* [12].

- **GUROBI**: Centrado en resolver problemas de programación lineal y entera [3].

- **XA**: Se centra también en la resolución de problemas de programación lineal y entera. Se basa en el algoritmo de *Simplex* y *Barrera* [2].

- **Excel Solver**: Es una herramienta que proporciona Excel, que sirve para resolver problemas de optimización lineal y no lineal. Es capaz de resolver problemas que tengas hasta 200 variables de decisión [18].

1.4. Problema Vertex Separation

En este PFC se abordará el problema del Vertex Separation, el cual se encuentra dentro de los problemas de ordenación lineal. Los problemas de ordenación lineal son una clase particular de los problemas de optimización combinatoria cuyo objetivo es encontrar una ordenación de los vértices de un grafo, de tal manera que la FO se optimiza.

1.4.1. Definición del problema

Para poder realizar la descripción detallada del problema, primero se va a introducir los conceptos básicos sobre la nomenclatura del problema [8]. La nomenclatura que se va a manejar es la siguiente:

- **G=(V,E)**: Representa una instancia del problema (Un grafo con un conjunto de vértices y aristas).
- **V(G)**: Representa el conjunto de vértices del grafo.
- **E(G)**: Representa el conjunto de aristas del grafo.
- φ: Representa una ordenación lineal del grafo, es decir, una solución específica al problema.
- φ*: Representa todas las ordenaciones lineales del grafo, es decir, el conjunto de todas las posibles soluciones al problema.
- **i**: Representa el punto de corte.
- $\mathbf{L(i,\varphi,G)=\{u \in V : \varphi(u) \leq i\}}$: Representa el conjunto izquierdo, es decir, todos aquellos vértices que quedan a la izquierda de un punto de corte i.
- $\mathbf{R(i,\varphi,G)=\{u \in V : \varphi(u) > i\}}$: Representa el conjunto derecho, es decir, todos aquellos vértices que quedan a la derecha de un punto de corte i.
- $\mathbf{\delta(i,\varphi,G)=\{u \in L(i,\varphi,G) : \exists v \in R(i,\varphi,G): uv \in E(G)\}}$: Representa el número de vértices del conjunto izquierdo que tienen como adyacente al menos un vértice del conjunto derecho. Esto representa a la FO del Vertex Separation de una ordenación lineal determinada (φ).

El objetivo que persigue el **Vertex Separation**, consiste en quedarse con la FO mínima de todas las posibles ordenaciones lineales del grafo (soluciones):

$$\text{vs}(\varphi^*,\text{G}) = \text{MINVS(G)}$$

La FO de cada solución se calcula como el número máximo de vértices del conjunto izquierdo que cumplen con la restricción de tener al menos un adyacente en el conjunto derecho para todos los posibles puntos de corte del grafo:

$$\text{vs}(\varphi,\text{G}) = \max_{i \in [|V|]} \delta(\text{i},\varphi,\text{G})$$

En cuanto a la solución del problema, se representará mediante un *array* unidimensional, donde cada posición del *array* corresponderá a un vértice de la instancia del problema. Se ha elegido este tipo de estructura, ya que facilita la representación de la solución como una ordenación lineal y es fácil de manipular.

Una vez descrito, de forma teórica, en qué consiste el problema del Vertex Separation y cómo se va a representar su solución, se va a proceder a explicarlo mediante un ejemplo práctico.

En primer lugar debemos elegir una instancia con la cual trabajar, como se trata de un ejemplo, la instancia elegida está formada únicamente por cinco vértices, con el objetivo de reducir la complejidad de la explicación.

Como se observa en la Figura 1.3 la instancia elegida representa un grafo finito, conexo, sin ciclos y no dirigido.

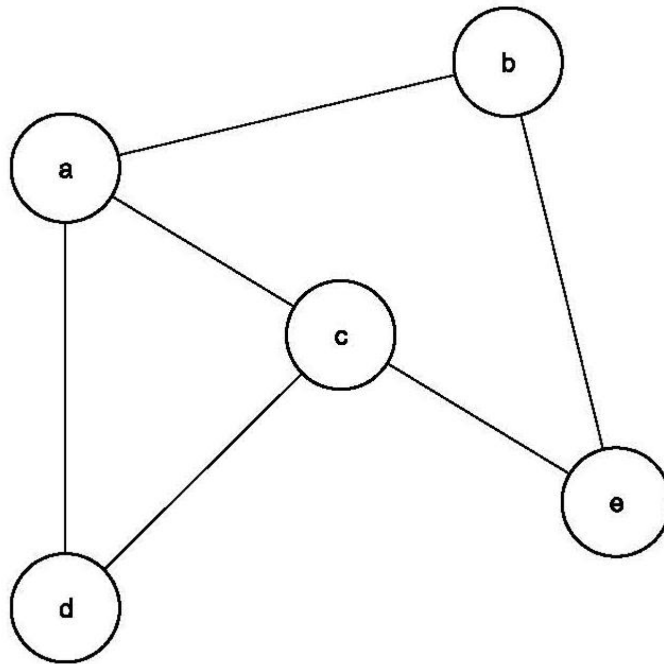

Figura 1.3: Instancia G

Una vez elegida la instancia, obtenemos una ordenación lineal (φ). Sobre esta ordenación lineal (φ) que observamos en la Figura 1.4 calculamos el valor de la FO para todos los puntos de corte de φ. Este proceso se compone de un número concreto de iteraciones, que viene determinado por todos los vértices de la instancia exceptuando el último vértice, ya que el corte de este último no se evalúa.

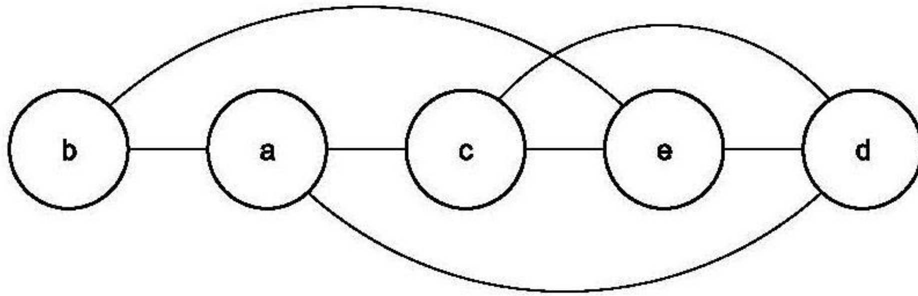

Figura 1.4: Ordenación Lineal(φ)

En el primer corte, como se observa en la Figura 1.5, la FO siempre va a tomar el valor de uno, ya que estamos ante un grafo conexo. En el segundo corte, mostrado en la Figura 1.6, se puede ver como en el conjunto izquierdo L(i,φ,G), formado por dos elementos {b,a}, ambos elementos tienen al menos un adyacente en el conjunto derecho R(i,φ,G), la FO en esta situación toma el valor de dos. En la Figura 1.7, referente al punto de corte dos, se puede ver como el valor de la FO toma el valor de tres, esto se debe a que L(i,φ,G) tiene tres elementos y todos ellos tienen un adyacente en R(i,φ,G). Por último, en la Figura 1.8, se muestra la última iteración, donde se observa que a pesar de que el conjunto izquierdo L(i,φ,G) contiene cuatro elementos, el valor de la FO es tres, ya que sólo tres elementos de L(i,φ,G) {a,c,e} tienen un adyacente en R(i,φ,G).

Una vez calculado δ(i,φ,G) para todos los puntos de corte de φ nos quedamos con el valor máximo que representará la FO de φ. En el ejemplo la FO de φ es tres. Sin embargo el problema no esta resuelto ya que únicamente hemos calculado la FO de una ordenación lineal φ. Lo que hay que hacer es realizar el proceso anterior para φ*, es decir, para todas las posibles ordenaciones lineales de la instancia G.

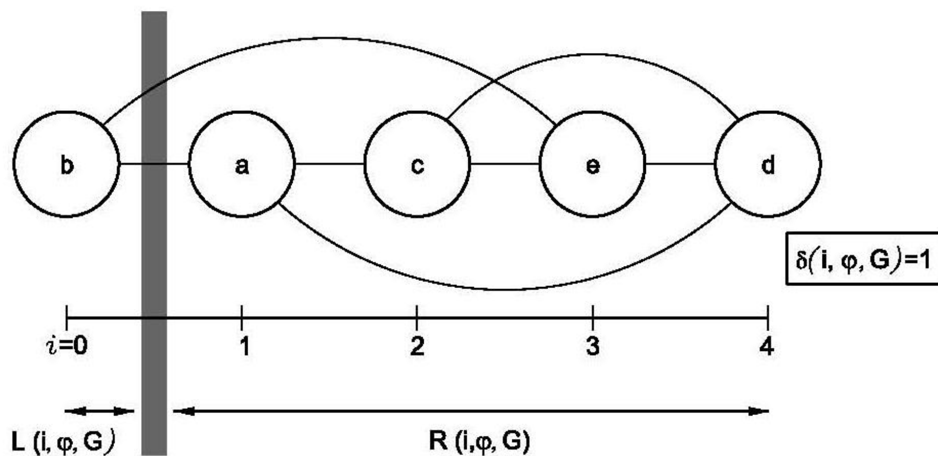

Figura 1.5: Cálculo de la Función Objetivo para i=0

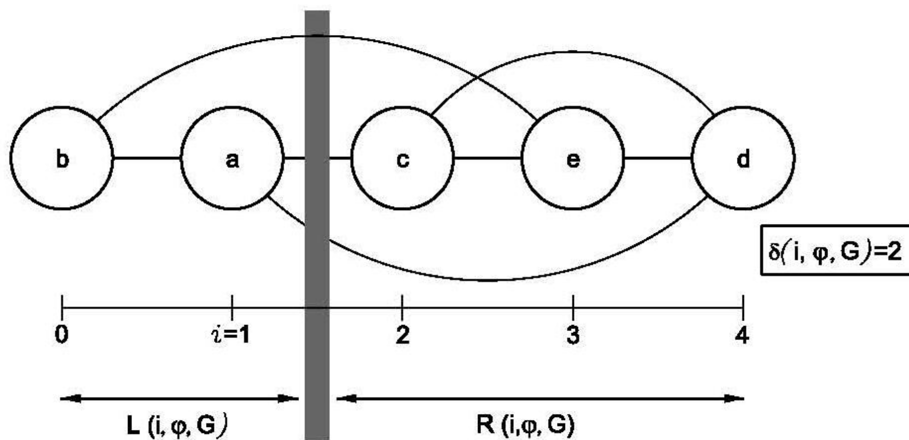

Figura 1.6: Cálculo de la Función Objetivo para i=1

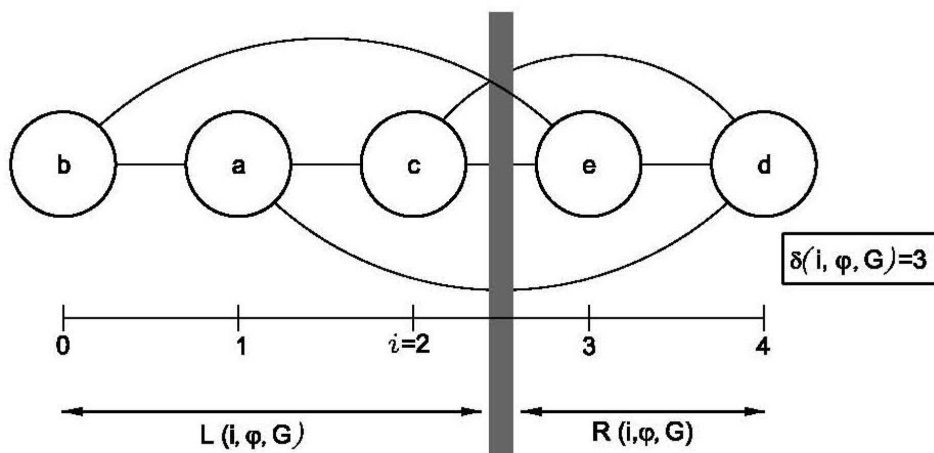

Figura 1.7: Cálculo de la Función Objetivo para i=2

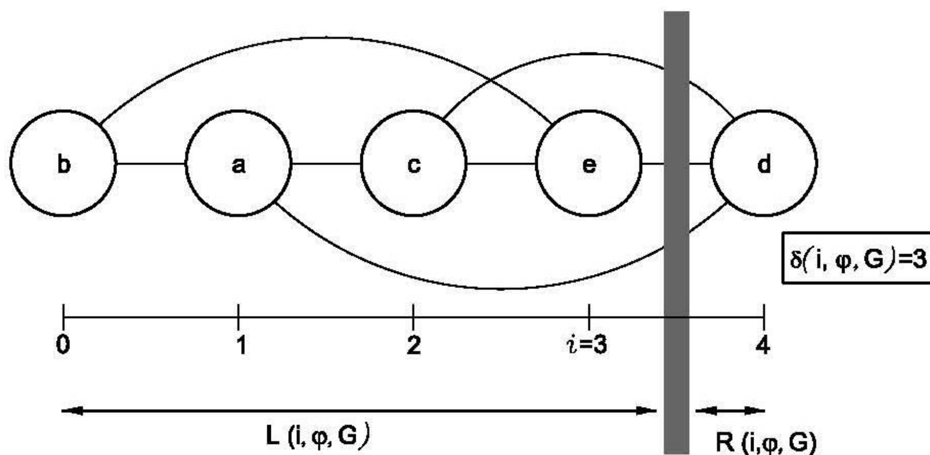

Figura 1.8: Cálculo de la Función Objetivo para i=3

Una vez calculado todas las FO de φ*, la solución al problema del Vertex Separation será aquella ordenación lineal φ que tenga la menor FO.

Recapitulando, los pasos para resolver el problema del Vertex Separation son los siguientes:

- La FO de cada uno de los punto de corte de φ se ha calculado como el número máximo de vértices del conjunto izquierdo L(i,φ,G) que cumplen con la restricción de tener, al menos, un adyacente en el conjunto derecho R(i,φ,G).

- Una vez calculados los distintos valores que puede tomar δ(i,φ,G) para todos los puntos de corte de φ, lo que se debe hacer es quedarse con el máximo de ellos. Ese valor corresponderá al valor de la FO de φ. En nuestro ejemplo la FO de φ es 3.

- Este proceso habría que repetirlo para φ*, es decir, para todas las posibles soluciones de la instancia G.

- Una vez que se ha realizado este proceso para φ*, la solución al problema del Vertex Separation vendrá dada por la ordenación lineal φ que posea la mínima FO.

El Vertex Separation se trata de un problema NP-Completo de minimización, donde no se puede garantizar encontrar la mejor solución en un tiempo razonable [8].

Más adelante se explicarán las distintas técnicas tanto exactas como aproximadas que se han empleado para intentar resolver este problema complejo.

1.4.2. Aplicaciones

El Vertex Separation está relacionado con muchos problemas del área de las Ciencias de la Computación. Cabe destacar también su uso en el diseño de circuitos integrados VLSI [8].

VSLI corresponde a la sigla en ingles de *Very Large Scale Integration*, integración a escala muy grande.

La introducción de la tecnología VLSI ha tenido un gran impacto en el diseño de circuitos integrados. Las ventajas introducidas son:

- **Incremento en la densidad de integración**, con reducción de tamaño y peso de circuitos. Esto permite emplear más circuitería para implementar redundancia en menos espacio de manera más fiable y barata.

- **Reducción del consumo** debido a que la mayoría de las conexiones se hacen en un solo chip con lo que se reducen las señales empleadas.

- **Reducción de coste** en los circuitos integrados.
- **Mayor fiabilidad**, ya que al hacerse las conexiones internas es más difícil que se rompan o deterioren, por lo que el ratio de averías es más bajo. Los fallos se detectan antes y se evita la propagación de éstos.
- **Formalización del proceso de diseño**.

En los diseños VSLI, el problema del Vertex Separation se aplica como una forma de partición de circuitos VLSI en subsistemas más pequeños, con el objetivo de tener el menor número de componentes que conecten dichos subsistemas [8].

1.5. Propuesta

En esta sección se van a explicar los pasos que se van a seguir para intentar buscar un algoritmo de resolución eficiente, capaz de obtener soluciones de calidad.

En primer lugar se intentará abordar el problema mediante técnicas exactas:

- **Backtracking**
- **Backtracking con poda**

Una vez que se evalúe mediante experimentación si la aplicación de estas técnicas es eficiente, se empezará a trabajar sobre distintos métodos aproximados basados en heurísticas y metaheurísticas. El proceso que se seguirá es el siguiente:

- **Constructivo voraz**: Se estudiará el problema y se determinarán diversas técnicas que se encargarán de seleccionar, con distintos criterios, aquellos elementos que se crea que puedan construir una solución con el menor valor de la FO posible.
- **Búsqueda local**: Una vez realizados los distintos constructivos, se procederá a implementar algoritmos de búsqueda local, cuyo objetivo es mejorar la solución obtenida por los constructivos. Sin embargo, esta mejora estará ligada al mínimo local y no al global, que es el que realmente interesa. Una vez realizadas las distintas búsquedas locales, se intentará factorizar los cálculos que en ella se incluyan, para reducir los tiempos de cómputo mediante el estudio exhaustivo del problema, para dar con la clave de la factorización.

Una vez implementadas las distintas heurísticas y realizada la experimentación adecuada, se avanzará un grado más de abstracción, introduciendo técnicas metaheurísticas con las cuales se intentará obtener soluciones de mejor calidad. Estas metaheurísticas

van a estar basadas en las heurísticas que se han comentado anteriormente. Las técnicas metaheurísticas que se van a implementar son:

- **GRASP**: Este método metaheurístico combina un constructivo *greedy* con una búsqueda local. En este caso, una vez que se hayan realizado todas las experimentaciones anteriores, se seleccionará el mejor constructivo *greedy* y la mejor búsqueda local y con ello se implementará el algoritmo GRASP.

- **Path Relinking**: Tras la implementación de GRASP, se implementará la técnica Path Relinking. Esta metaheurística construirá el *Elite Set* mediante la técnica GRASP. Se implementarán dos tipos distintos de Path Relinking:

  - **Path Relinking Estático**: El *Elite Set* se construye con las diez mejores soluciones obtenidas de ejecutar cien veces la técnica de GRASP. Las soluciones del *Elite Set* se comparan entre sí, generando las soluciones intermedias necesarias, hasta llegar de una solución a otra.
  - **Path Relinking Dinámico**: El *Elite Set* se construye con las diez primeras soluciones obtenidas mediante el método GRASP y, de forma aleatoria, se van escogiendo estas soluciones, comparándolas con las noventa soluciones restantes obtenidas mediante el algoritmo GRASP, generando las soluciones intermedias necesarias, hasta llegar de una solución a otra.

En ambos casos si en el trascurso de una solución inicial a una final, se obtiene una solución con una FO mejor que alguna de las soluciones del *Elite Set*, esta se incluye en el conjunto y se repite de nuevo todo el proceso anterior. Es importante recordar que el tamaño del *Elite Set* es constante, por lo que cada vez que se introduce una solución hay que eliminar del conjunto la peor de ellas.

Finalmente, una vez que se hayan probado todas las técnicas y se hayan realizado todas las experimentaciones necesarias, se obtendrá la conclusión de cuál de ellas obtiene mejores resultados para el problema del Vertex Separation. En la Figura 1.9 se intenta resumir todo lo que se ha comentado anteriormente en esta sección.

En lo que resta de memoria se abordarán los siguientes aspectos:

- En el Capítulo 2 se explicarán los objetivos que se persiguen con el desarrollo de este PFC.

- En el Capítulo 3 se detallará la descripción algorítmica, donde se explicarán los algoritmos implementados.

- En el Capítulo 4 se hablará sobre la descripción informática, en la cual, entre otras cosas, se detallará la metodología empleada para el desarrollo de este PFC.
- En el Capítulo 5 (Resultados experimentales) se recogerán y se compararán los resultados de los experimentos llevados a cabo durante todo el desarrollo del PFC.
- Por último, en el Capítulo 6 se comentarán las conclusiones acerca del PFC y los trabajos futuros.

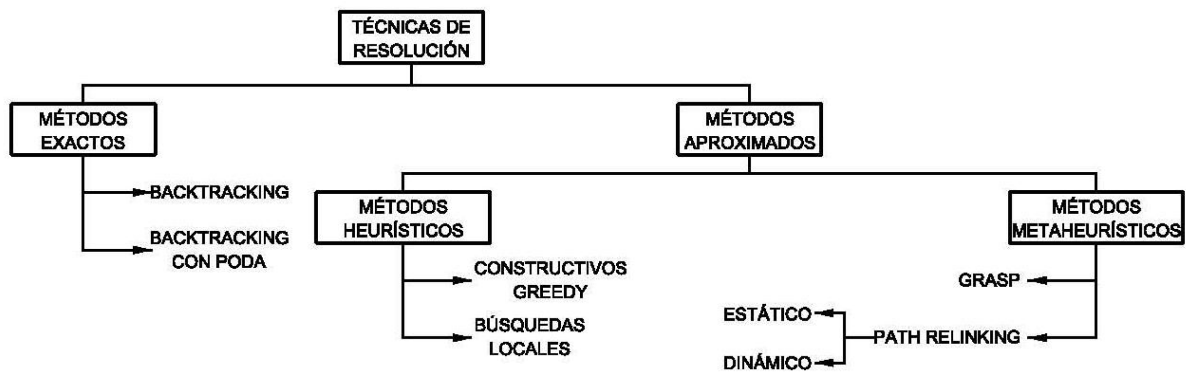

Figura 1.9: Esquema de técnicas de resolución

Capítulo 2

Objetivos

En este capítulo se van a describir los objetivos que se persiguen en este PFC y también los objetivos personales que se desean alcanzar con el desarrollo del mismo.

2.1. Objetivos del Proyecto Fin de Carrera

El objetivo de este PFC es desarrollar un algoritmo metaheurístico capaz de resolver el problema del Vertex Separation de una forma eficiente.

Para reducir la complejidad de alcanzar el objetivo final, la mejor solución es dividir el objetivo final en subobjetivos menos complejos y finalmente combinar todos ellos para alcanzar el objetivo deseado.

Inicialmente se demostrará la dificultad de la resolución del problema mediante técnicas exactas de optimización, como por ejemplo el *backtracking*, con el objetivo de comprobar que estas técnicas no son eficientes para este tipo de problemas. A continuación se orientará la resolución del problema hacia técnicas aproximadas de optimización.

Una vez cumplido con el objetivo anterior, se procederá a explicar el campo de las técnicas aproximadas de optimización, donde se empezará a investigar sobre las distintas heurísticas que se podrían aplicar a este problema, con el objetivo de conseguir soluciones aceptables en un tiempo de cómputo razonable.

Se utilizarán, a continuación, distintas metaheurísticas con las cuales se intentará mejorar las soluciones que se obtengan con las otras técnicas, alcanzando así el objetivo final que persigue este PFC.

De manera más detallada, este PFC se podría dividir en los siguientes subobjetivos:

- Implementación de algoritmos exactos.
- Desarrollo de algoritmos constructivos.
- Implementación de búsquedas locales.

- Desarrollo de técnicas metaheurísticas.

2.2. Objetivos personales

Con el desarrollo de este PFC espero alcanzar los siguientes objetivos personales que, sin duda, en un futuro serán de gran utilidad:

- Profundizar y reforzar mis conocimientos en la tecnología Java.
- Realizar la memoria del PFC mediante LaTeX, con el objetivo de aprender a utilizar este sistema de documentación formal, que nunca había utilizado a lo largo de la carrera. Esta elección viene ligada a que sin duda LaTeX, aporta a los textos un aspecto profesional.
- Realizar mi primer documento de carácter científico.
- Ampliar mis conocimientos sobre las distintas técnicas de optimización.
- Aprender una nueva metodología de desarrollo *software* distinta a la del Proceso Unificado (PU), que es la única metodología que se estudia a lo largo de la carrera.
- Iniciarme en el campo de la investigación.

Aparte de todos estos objetivos personales que me propongo conseguir, sin duda mi objetivo principal con el desarrollo del PFC es acabar mi carrera como Ingeniero Informático, que supondrá acabar una etapa en mi vida y comenzar otra nueva etapa donde me esperan nuevas experiencias e inquietudes.

Capítulo 3

Descripción Algorítmica

En esta sección se van a describir de forma detallada los diferentes algoritmos que se han implementado durante el desarrollo de este PFC. Estos algoritmos se pueden clasificar en tres grandes grupos:

- Exactos
- Heurísticos
- Metaheurísticos

3.1. Algoritmos exactos

Los algoritmos exactos se utilizan para problemas cuya dificultad de cómputo es moderada o baja. Estos algoritmos proporcionan la solución óptima del problema. Los algoritmos exactos implementados durante este PFC han sido:

- *Backtracking*
- *Backtracking* con poda

3.1.1. Backtracking

El algoritmo de *backtracking* que se ha implementado busca la solución óptima del problema, es decir, genera todas las permutaciones de forma organizada, de manera que prueba todas las posibles ordenaciones lineales del problema, quedándose con la mejor solución de todas. En general, la forma de actuar consiste en elegir una alternativa del conjunto de opciones en cada etapa del proceso de resolución y, si esta elección no conduce a ninguna solución, la búsqueda vuelve al punto donde se realizó esa elección y lo intenta con otro valor. Cuando se han agotado todos los posibles valores en ese punto, la búsqueda vuelve a la anterior fase en la que se hizo otra elección entre valores. Si no hay más puntos

de elección, la búsqueda finaliza. En la Figura 3.1 se muestra un ejemplo gráfico del funcionamiento del *backtracking*, y su algoritmo está reflejado en el Pseudocódigo 3.1.

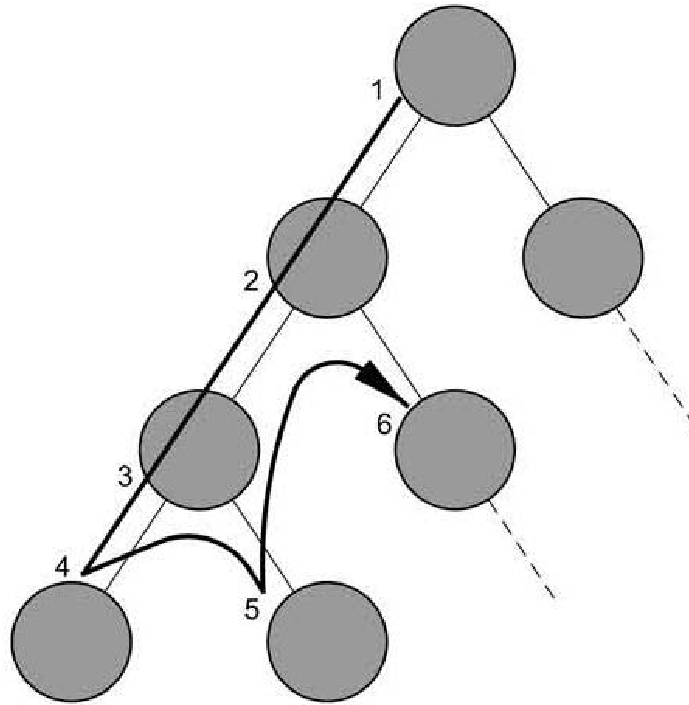

Figura 3.1: Orden de expansión de los nodos de un árbol de exploración por el algoritmo de *backtracking*

Pseudocódigo Backtracking

```
Función Backtracking(etapa):
   Repetir ∀v ∈ V(G).
           Si solucionCompleta(solucionActual) entonces
              Si esMejorSolucion(solucionActual,solucionMejor) entonces
                solucionMejor ← solucionActual
           Si no
              Si esFactibleSolucion(solucionActual) entonces
                 añadirVerticeSolucion(v,solucionActual)
                 Backtracking(etapaSiguiente)
```

Pseudocódigo 3.1: Backtracking

3.1.2. Backtracking con poda

Es un refinamiento del *backtracking* con el que se consigue reducir el número de candidatos y se mejora, de forma considerable, el tiempo de cómputo. Se encarga de generar un árbol de soluciones, pero introduce una estrategia de ramificación, que permite guiar la búsqueda por estimaciones de beneficio. Dichas estimaciones se calculan en cada nodo. Además, incluye una estrategia de poda, que permite eliminar nodos que no llevan a la solución óptima.

La poda es un mecanismo que permite descartar el recorrido de ciertas zonas del espacio de búsqueda. El *backtracking* que se ha implementado, realiza su expansión en profundidad. La cota que indica si se poda o no una rama, se inicializa al máximo valor entero posible (*MAX_INT*). Según se va expandiendo el árbol en profundidad, en los nodos intermedios, se calcula el valor de la FO y se compara con la cota de poda. Si la FO en ese nodo es superior a la cota de poda, entonces se deja de expandir esa rama y se retrocede al nivel anterior. En los nodos hoja, es donde se actualiza el valor de la cota de poda. En la Figura 3.2 se muestra un ejemplo de la estrategia de poda aplicada al *backtracking* y su pseudocódigo está reflejado en el Pseudocódigo 3.2.

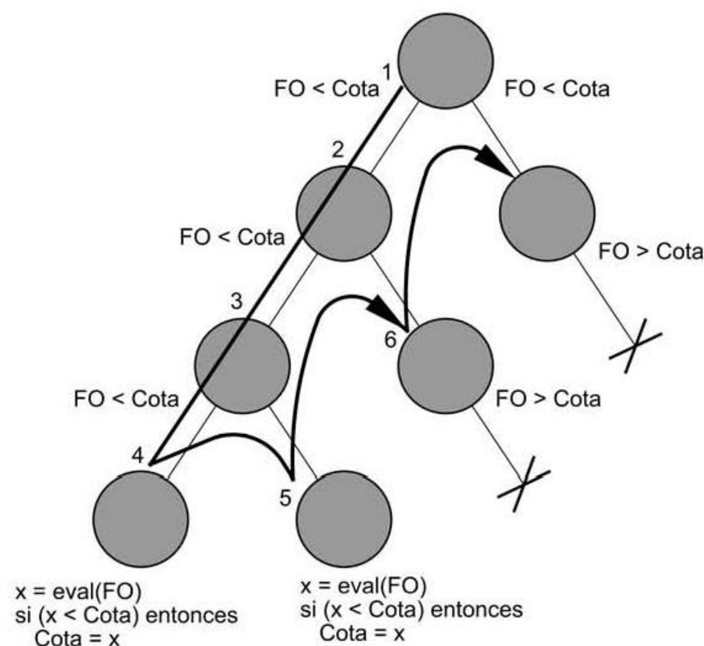

Figura 3.2: Orden de expansión de los nodos de un árbol de exploración por el algoritmo de *backtracking* con poda

Pseudocódigo Backtracking con Poda

```
Función BacktrackingPoda(etapa):
   mejorCota ← ∞
   Repetir ∀v ∈ V(G).
           Si solucionCompleta(solucionActual) entonces
              Si esMejorSolucion(solucionActual,solucionMejor) entonces
                solucionMejor ← solucionActual
                mejorCota ← evaluarSolucion(solucionMejor)
           Si no
              Si esFactibleSolucion(solucionActual) entonces
                 Si evaluarSolucion(solucionActual) < mejorCota entonces
                    añadirVerticeSolucion(v,solucionActual)
                    BacktrackingPoda(etapaSiguiente)
```

Pseudocódigo 3.2: Backtracking con poda

3.2. Algoritmos heurísticos

Consisten en procedimientos simples, a menudo basados en el sentido común, que se supone que obtendrán una buena solución (no necesariamente la óptima) a problemas difíciles, de un modo sencillo y rápido.

3.2.1. Constructivo voraz

El constructivo voraz (*greedy*) trata de construir una solución, seleccionando iterativamente aquellos elementos que van hacer posible construir una buena solución del problema de forma rápida y sencilla. En este PFC se han implementado cuatro tipos distintos de constructivos, que tienen la misma estructura, la cuál puede verse en el Pseudocódigo 3.3, y únicamente se diferencian en el criterio de selección del siguiente vértice a etiquetar:

- **Constructivo Voraz de Menor Grado (CVMG)**: Los candidatos son todos aquellos vértices que no han sido etiquetados. De todo el conjunto de candidatos se selecciona el vértice que menor grado de adyacencia tiene.

- **Constructivo Voraz de Menor Adyacencia (CVMeA)**: Los candidatos son todos los adyacentes de los vértices ya etiquetados. De todo el conjunto de candidatos se selecciona el vértice que menor grado de adyacencia tiene.

- **Constructivo Voraz de Mayor Adyacencia (CVMaA)**: Los candidatos son todos los adyacentes de los vértices ya etiquetados. De todo el conjunto de candidatos se selecciona el vértice que mayor grado de adyacencia tiene.
- **Constructivo Voraz de Menor Diferencia (CVMD)**: Los candidatos son todos aquellos vértices que no han sido etiquetados. De todo el conjunto de candidatos se selecciona el vértice que tenga menor diferencia entre sus adyacentes no etiquetados y sus adyacentes etiquetados.

Pseudocódigo constructivo voraz

```
Procedimiento Constructivo:
   Sean S y N los conjuntos de vértices etiquetados y no etiquetados
   respectivamente
   Inicialmente S = ϕ y N = V(G).
   Seleccionamos el vértice v ∈ N que tiene menor grado de
   adyacencia
   Asignación de la etiqueta k = 1 a v
   S = {v}, N = N\{v}
   Mientras N ≠ ϕ
      k = k + 1
      Seleccionamos un vértice v ∈ N en base al criterio de
      selección elegido.
      Asignación de la etiqueta k al vértice v
      S = S ∪ {v}, N = N\{v}
```

Pseudocódigo 3.3: Constructivo voraz

3.2.2. Búsquedas locales

Parten de una solución factible dada y a partir de ella intentan mejorarla. Para ello examinan su vecindad y todos los posibles movimientos, seleccionando el mejor movimiento de todos. El tiempo computacional es mayor que el de las heurísticas constructivas, pero obtienen mejores soluciones. El inconveniente de las búsquedas locales es su incapacidad para escapar de óptimos locales.

Durante el desarrollo de este PFC se han implementado tres tipos distintos de búsqueda local:

- Búsqueda Local Exhaustiva (BLE).

- Búsqueda Local Factorizada (BLF).
- Búsqueda Local Inteligente (BLI).

Las búsquedas locales mencionadas anteriormente construyen las distintas soluciones pertenecientes a la vecindad mediante dos tipos de procedimientos:

- Basado en Intercambios: Consiste únicamente en intercambiar los elementos de la posición a mover y la posición destino. Se muestra un ejemplo en la Figura 3.3
- Basado en Inserciones: Consiste en mover el elemento de la posición a mover a la posición destino y desplazar el resto de elementos comprendidos entre ese intervalo. Hay que tener en cuenta que dichos elementos se desplazarán hacia la izquierda si la posición a mover es menor que la posición destino y a la derecha si ocurre lo contrario. Se muestra un ejemplo en la Figura 3.3

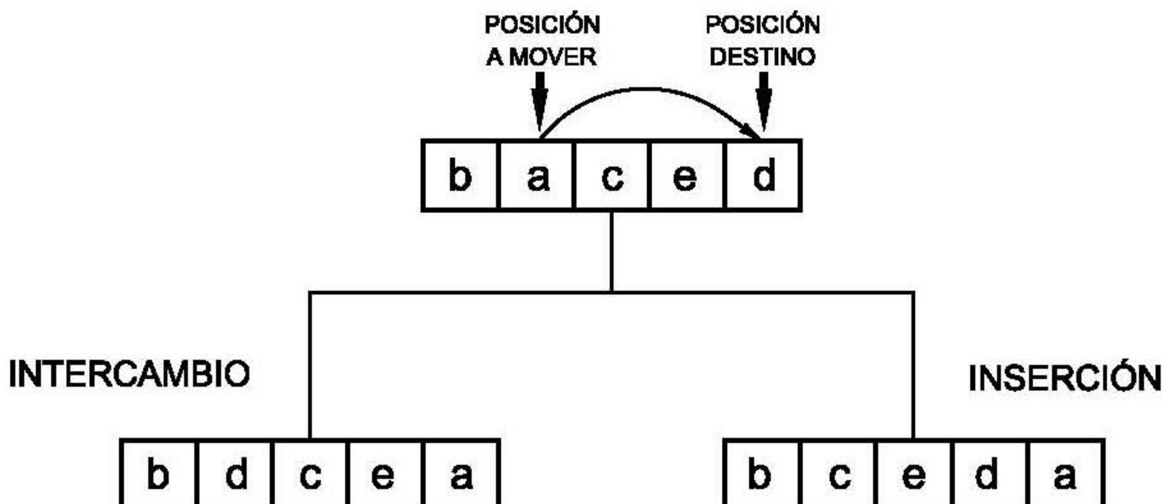

Figura 3.3: Procedimientos de Búsqueda Local (Inserción e intercambio)

Descripción del algoritmo Búsqueda Local Exhaustiva

En este tipo de búsqueda se mueven todos los elementos de la solución, y las nuevas soluciones obtenidas se evalúan por completo.

Lo primero que se hace es calcular, de forma aleatoria, la posición de inicio con la que se va a empezar. Una vez calculada, se recorren todas las posiciones a las que se puede mover el elemento, empezando por dicha posición. En cada iteración se calcula, de forma aleatoria, la posición destino inicial a la que se va a mover el elemento y, partiendo de esta posición, se recorren el resto de posiciones a las que se puede mover.

En cuanto se produzca un cambio entre la posición a mover y la posición destino que mejore la FO, se actualiza la solución. Esto provoca que se salga del bucle encargado de recorrer todas las posiciones destino a las que se puede mover el elemento y que se pruebe con la siguiente posición a mover. Si a lo largo de todo el proceso se produce una mejora de la solución, éste se vuelve a repetir. El pseudocódigo de esta búsqueda local puede verse en el Pseudocódigo 3.4.

Pseudocódigo Búsqueda Local Exhaustiva

```
Procedimiento Búsqueda Local Exhaustiva(S):
   Sea S una solución al problema y
   K el valor de la Función Objetivo(FO) de S.
   mejora ← cierto
   Mientras mejora = cierto
      mejora ← falso
      LPM ← lista de posiciones a mover de S (todas las posiciones).
      ∀i ∈ LPM
         LPD ← lista de posiciones destino de S (todas las posiciones).
         ∀j ∈ LPD
            S’ = procedimientoBusqueda(i,j)
            K’ ← evaluarSolucionCompleta(S’)
            Si K’ < K entonces
               S ← S’
               K ← K’
               mejora ← cierto
               break
```

Pseudocódigo 3.4: Búsqueda Local Exhaustiva

Descripción del algoritmo Búsqueda Local Factorizada

Este tipo de búsqueda consiste en un refinamiento de la Búsqueda Local Exhaustiva. La única diferencia que existe se encuentra en la evaluación de la solución obtenida tras realizar un movimiento.

En la BLE se evalúa por completo la solución obtenida al aplicar la búsqueda local, mientras que en la BLF la solución no es evaluada por completo, sino que únicamente se evalúan los cortes comprendidos entre la posición a mover y la posición destino, ya que el resto no se ven afectados por el cambio. En la Figura 3.4 se puede ver la diferencia entre

la BLE y la BLF. Con este factorización se reduce el tiempo computacional del algoritmo. Su pseudocódigo puede observarse en el Pseudocódigo 3.5.

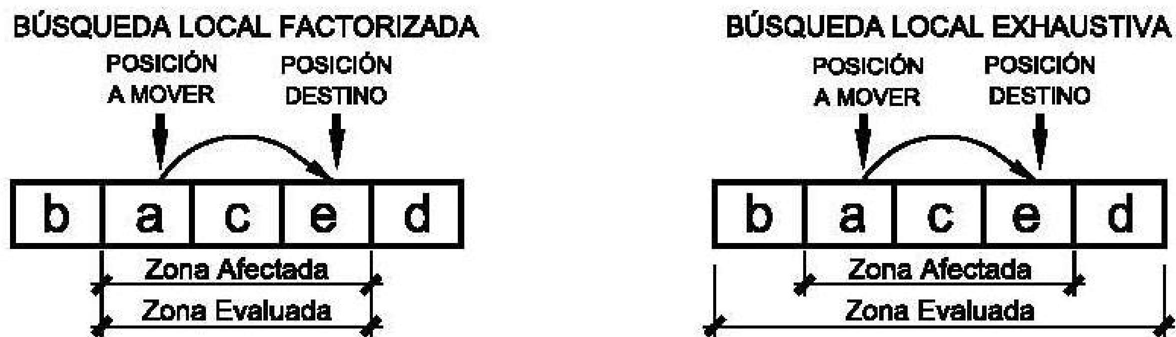

Figura 3.4: Búsqueda Local Factorizada vs Búsqueda Local Exhaustiva

Pseudocódigo Búsqueda Local Factorizada

```
Procedimiento Búsqueda Local Factorizada(S):
   Sea S una solución al problema y
   K el valor de la Función Objetivo(FO) de S.
   mejora ← cierto
   Mientras mejora = cierto
      mejora ← falso
      LPM ← lista de posiciones a mover de S (todas las posiciones).
      ∀i ∈ LPM
         LPD ← lista de posiciones destino de S (todas las posiciones).
         ∀j ∈ LPD
            S’ = procedimientoBusqueda(i,j)
            K’ ← evaluarSolucionParcial(S’,i,j)
            Si K’ < K entonces
               S ← S’
               K ← K’
               mejora ← cierto
               break
```

Pseudocódigo 3.5: Búsqueda Local Factorizada

Descripción del algoritmo Búsqueda Local Inteligente

La Búsqueda Local Inteligente, se basa en la Búsqueda Local Factorizada, ya que el procedimiento algorítmico es el mismo. La única diferencia es que los candidatos a ser

movidos no son todos los elementos de la solución, como ocurría en el caso de la BLF, sino que estos candidatos han de cumplir con la restricción de encontrarse en las posiciones de corte donde se produzca el máximo valor de la FO. Los vértices que se encuentran en estas posiciones, se denominan vértices críticos, y son los candidatos a ser movidos, ya que se parte de la idea de que si se mueven los vértices que se encuentran en las posiciones críticas, la FO en dicha posiciones disminuirá, mejorándose así la calidad de la solución. Puede verse un ejemplo gráfico en la Figura 3.5. El pseudocódigo de la Búsqueda Local Inteligente puede verse en el Pseudocódigo 3.6.

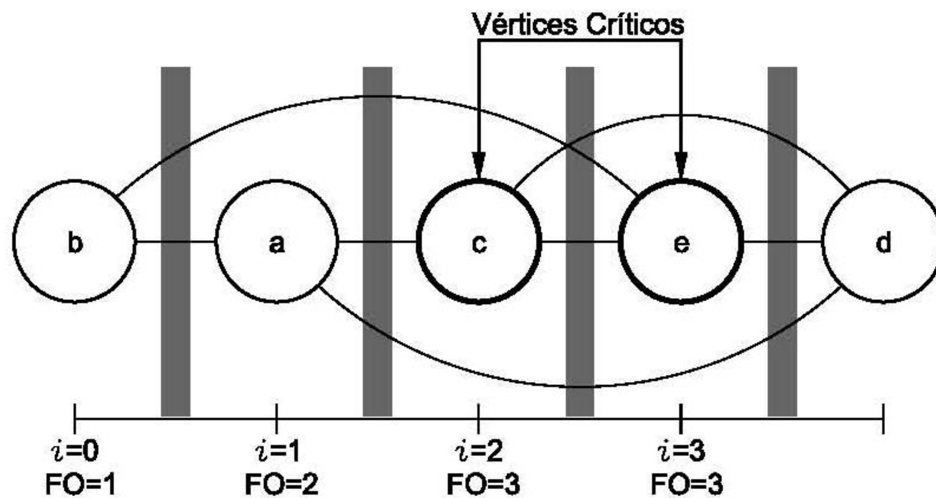

Figura 3.5: Ejemplo de vértices críticos

Pseudocódigo Búsqueda Local Inteligente

```
Procedimiento Búsqueda Local Inteligente(S):
   Sea S una solución al problema y
   K el valor de la Función Objetivo(FO) de S.
   mejora ← cierto
   Mientras mejora = cierto
      mejora ← falso
      LPC ← lista de posiciones críticas a mover de S.
         ∀i ∈ LPC
         LPD ← lista de posiciones destino de S (todas las posiciones).
         ∀j ∈ LPD
            S’ = procedimientoBusqueda(i,j)
            K’ ← evaluarSolucionParcial(S’,i,j)
            Si K’ < K entonces
```

```
S ← S'
K ← K'
mejora ← cierto
```

Pseudocódigo 3.6: Búsqueda Local Inteligente

3.3. Algoritmos metaheurísticos

Son procedimientos con un mayor grado de abstracción, que guían y modifican otras heurísticas para explorar soluciones más allá de la simple optimalidad local [19].

3.3.1. GRASP

El nombre de esta metaheurística viene de su acrónimo en inglés GRASP (*Greedy Randomized Adaptive Search Procedure*), que en castellano se podría traducir como procedimiento de búsqueda voraz, aleatorizado y adaptativo.

GRASP es un proceso multiarranque en el que cada arranque se corresponde con una iteración. Cada iteración tiene dos fases bien diferenciadas: la fase de construcción, que se encargará 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 mediante una búsqueda local. Su pseudocódigo se muestra en el Pseudocódigo 3.7.

La fase constructiva es un procedimiento iterativo encargado de construir una solución elemento a elemento. Su implementación se muestra en el Pseudocódigo 3.8. Inicialmente, se parte de un conjunto vacío, y los elementos seleccionables, suponiendo que c_{min} y c_{max} son respectivamente los valores más bajo y más alto del coste de la solución, serían todos aquellos elementos cuyo coste fuese inferior al umbral dado por la siguiente expresión:

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

Esto es debido a que en el contexto de GRASP no se selecciona el mejor candidato posible, sino que se elige aleatoriamente un candidato de un conjunto de buenos candidatos. Este conjunto de candidatos recibe el nombre *Restricted Candidate List (RCL)*, que en castellano significa Lista de Candidatos Restringida. Puede verse un ejemplo en la Figura 3.6.

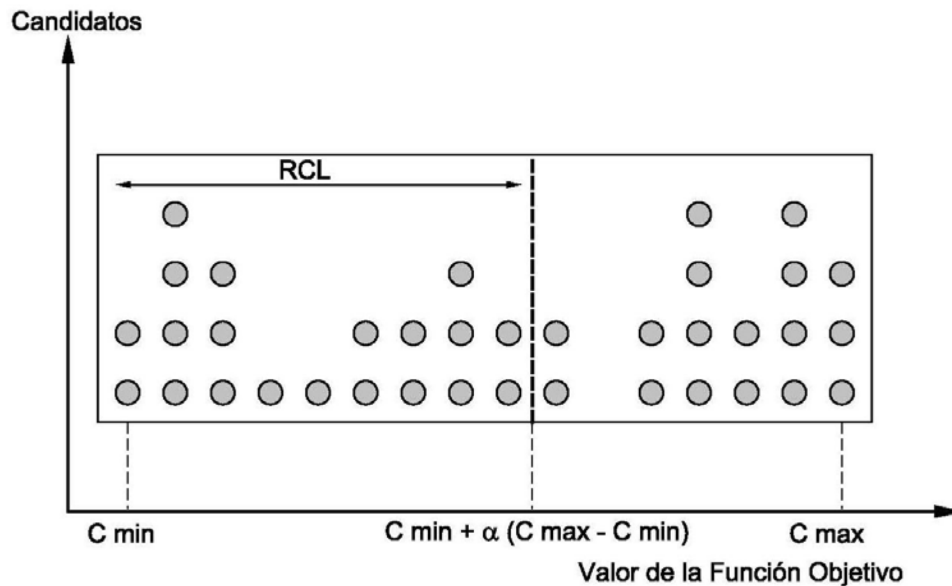

Figura 3.6: Lista de candidatos restringida para problemas de minimización

Pseudocódigo GRASP

```
Procedimiento GRASP(M):
   Sea M el número de soluciones a construir.
   K ← ∞ //Función Objetivo inicial
   for i:= 1 to M do
      α = rand()
      S ← ConstructivoVorazAleatorizado(α)
      S' ← MejorarSolucion(S)
      K' ← evaluarSolucion(S')
      si K' < K entonces
        K ← K'
        ActualizarMejorSolucion(S')
```

Pseudocódigo 3.7: Esquema algorítmico GRASP

Pseudocódigo constructivo aleatorizado GRASP

```
Procedimiento constructivo aleatorizado GRASP(α):
   Sea α el grado de aleatoriedad
   Sean S y N los conjuntos de vértices etiquetados y no etiquetados
   respectivamente
   Inicialmente S = φ y N = V(G).
```

```
Seleccionamos el vértice v ∈ N que tiene menor grado de
adyacencia
Asignación de la etiqueta k = 1 a v
S = {v}, N = N\{v}
Mientras N ≠ ϕ
   k = k + 1
   RLC ← ConstruirRLC(N,α) //Lista de candidatos
   v ← Seleccionar(RLC) //Seleccionar un candidato de forma aleatoria
   Asignación de la etiqueta k al vértice v
   S = S ∪ {v}, N = N\{v}
```

Pseudocódigo 3.8: Constructivo GRASP

3.3.2. Path Relinking

Path Relinking es otra metaheurística cuya idea es buscar soluciones que compartan atributos con otras soluciones, con la esperanza de obtener mejores soluciones. Se parte de una solución inicial, se realizan los movimientos oportunos, generando soluciones intermedias hasta llegar a una solución final. Durante todo este trayecto se va almacenando aquella solución que tiene mejor valor de la FO. Es una técnica poblacional (emplea un conjunto de soluciones en cada iteración del algoritmo) sin memoria y constructiva. Se muestra un ejemplo de la técnica Path Relinking en la Figura 3.7. Al realizar el recorrido entre la solución inicial y la solución final, hay dos alternativas:

- Realizar el recorrido desde la posición inicial hasta la posición final generando todas las soluciones intermedias necesarias y eligiendo la mejor solución.
- Similar al procedimiento anterior, únicamente que cuando se llega al nivel veinte de profundidad en el trascurso de generación de las soluciones intermedias, se obtiene la solución correspondiente a ese nivel y se aplica sobre ella una búsqueda local. El nivel en el que se aplica la búsqueda local puede variar según la implementación.

Se han implementado dos versiones distintas de Path Relinking que únicamente se diferencian entre sí en el cálculo del *Elite Set* y en los elementos que comparan.

- Path Relinking Estático (PRE): En esta versión, el *Elite Set* se construye con las diez mejores soluciones obtenidas de ejecutar cien veces el algoritmo de GRASP. Se realiza el recorrido mostrado en la Figura 3.7 entre todas las soluciones del *Elite Set*. Su pseudocódigo se muestra en el Pseudocódigo 3.9.

- Path Relinking Dinámico (PRD): El *Elite Set* se construye con las diez primeras soluciones obtenidas mediante el algoritmo GRASP y, de forma aleatoria, se van escogiendo estas soluciones y realizando el recorrido mostrado en la Figura 3.7 con las noventa soluciones restantes obtenidas mediante el método GRASP. Su pseudocódigo se muestra en el Pseudocódigo 3.9.

En ambos casos, si en el trascurso de una solución inicial a una final se obtiene una solución con una FO mejor que alguna de las soluciones del *Elite Set*, ésta se incluye en el conjunto y se repite de nuevo todo el proceso anterior. Es importante recordar que el tamaño del *Elite Set* es constante por lo que cada vez que se introduce una solución hay que eliminar del conjunto la peor de ellas.

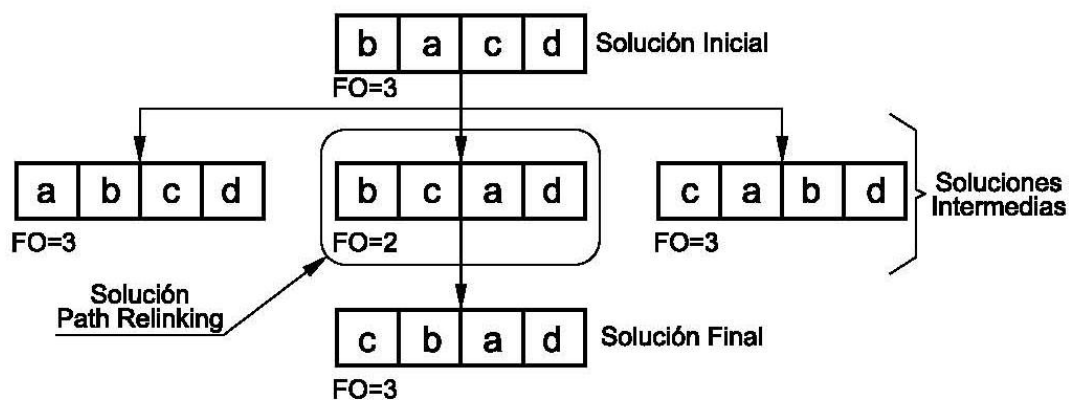

Figura 3.7: Ejemplo del algoritmo Path Relinking.

Pseudocódigo del algoritmo Path Relinking Estático

```
Procedimiento Path Relinking Estático:
   //Construir el EliteSet con las 10 mejores soluciones de la técnica GRASP
   EliteSet ← generarEliteSet()
   K ← ∞
   ∀inicio ∈ EliteSet
      ∀destino ∈ EliteSet
         Si inicio ≠ destino entonces
            S' ← realizarRecorrido(inicio,destino)
            K' ← evaluarSolucion(S')
            Si K' < K entonces
               K ← K'
               actualizarMejorSolucion(S')
```

Pseudocódigo 3.9: Path Relinking Estático

Pseudocódigo del algoritmo Path Relinking Dinámico

```
Procedimiento Path Relinking Dinámico:
   //Construir el EliteSet con las 10 primeras soluciones de la técnica GRASP
   //Devuelve una Lista de Candidatos (LC) con las 90 soluciones restantes
   (EliteSet,LC) ← generarEliteSet()
   K ← ∞
   ∀inicio ∈ LC
      destino ← obtenerCandidatoAleatorio(EliteSet)
      S’ ← realizarRecorrido(inicio,destino)
      K’ ← evaluarSolucion(S’)
      Si K’ < K entonces
         K ← K’
         actualizarMejorSolucion(S’)
```

Pseudocódigo 3.10: Path Relinking Dinámico

Capítulo 4

Descripción Informática

4.1. Introducción

En esta sección, en primer lugar se tratará el concepto de modelo de proceso, qué tipos de modelo de proceso existen y finalmente se elegirá el modelo de proceso más conveniente para el desarrollo de este PFC.

4.1.1. Modelo de proceso

Los modelo de proceso son estrategias de desarrollo que ayudan a organizar los diferentes procesos y actividades del ciclo de vida del *software*. Estos modelos ayudan al control y a la coordinación del proyecto. El modelo a utilizar depende del tipo de proyecto [10].

4.1.2. Tipos de modelo de proceso

Existen una gran variedad de modelos de proceso. Algunos de los más conocidos son:

- Modelo en Cascada
- Desarrollo Evolutivo
- Proceso Mixto
- Desarrollo con Reutilización
- Proceso Unificado
- Espiral
- Procesos Ágiles

Modelo en Cascada

El Modelo en Cascada es un modelo de proceso sin iteración, que se caracteriza por dividir el proceso de desarrollo en un conjunto de etapas secuenciales, donde una etapa no puede empezar hasta que no haya terminado la anterior. Al final de cada fase, se revisa el progreso del proyecto. El principal inconveniente que presenta este tipo de modelo es que ofrece poca flexibilidad ante los cambios de requisitos. Este tipo de modelo es práctico de utilizar en proyectos en los cuales los requisitos son muy estables, es decir, hay poca probabilidad de que cambien a lo largo de todo el proceso. Las fases de este modelo se muestran en la Figura 4.1 [10].

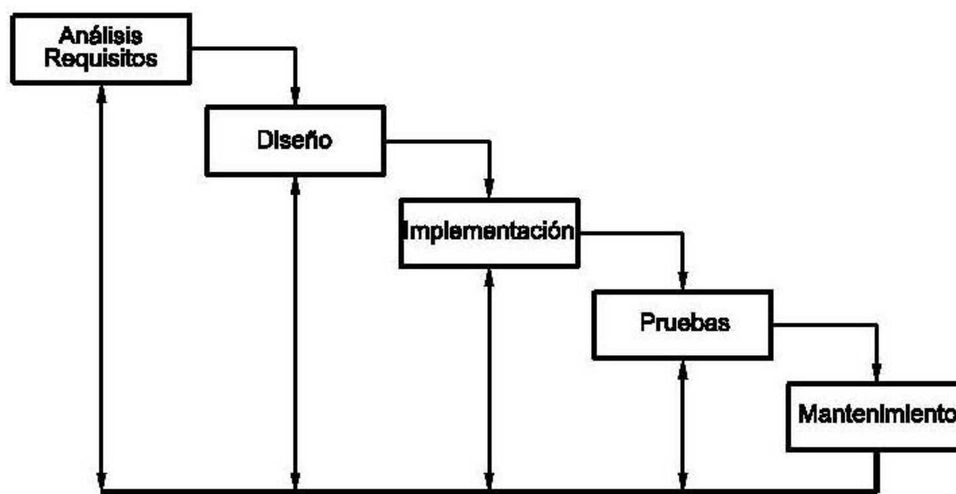

Figura 4.1: Modelo de Proceso en Cascada.

Desarrollo Evolutivo

El modelo de Desarrollo Evolutivo consiste en desarrollar una implementación inicial e ir refinándola hasta conseguir el sistema adecuado. Durante el desarrollo, las actividades se realizan de forma concurrente como se muestra en la Figura 4.2 [13].

La principal desventaja que presenta este modelo es la construcción de sistemas con estructuras deficientes, ya que se corre el riesgo de que el prototipo se convierta en el producto final. Este tipo de modelo es práctico de utilizar para sistemas interactivos pequeños, para partes de sistemas grandes y para sistemas con vida corta.

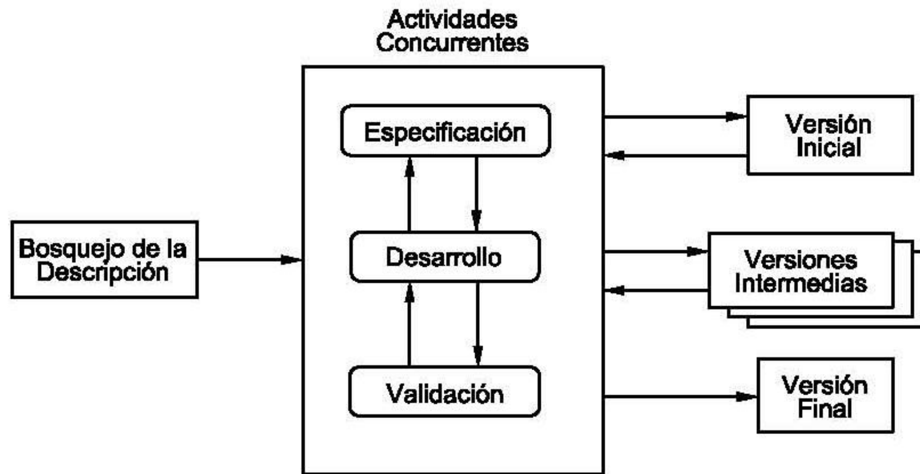

Figura 4.2: Modelo Evolutivo.

Proceso Mixto

El Proceso Mixto combina el modelo en Cascada y el modelo Evolutivo como se muestra en la Figura 4.3. Se caracteriza por el desarrollo de un prototipo desechable (con enfoque evolutivo) para resolver incertidumbres en la especificación inicial y, posteriormente, se reimplementa con un enfoque más estructurado, basado en el modelo en Cascada [13].

Figura 4.3: Modelo de Proceso Mixto.

Desarrollo con Reutilización

Este modelo esta basado en la reutilización sistemática, los sistemas se integran con componentes existentes o con sistemas COTS (*Commercial Of The Shelf*). Las etapas de este proceso están reflejadas en la Figura 4.4.

Este enfoque se está convirtiendo en el más importante, el inconveniente es que todavía no hay mucha experiencia sobre él. Se puede decir que es muy aplicado a nivel práctico, pero poco comprendido a nivel teórico [10].

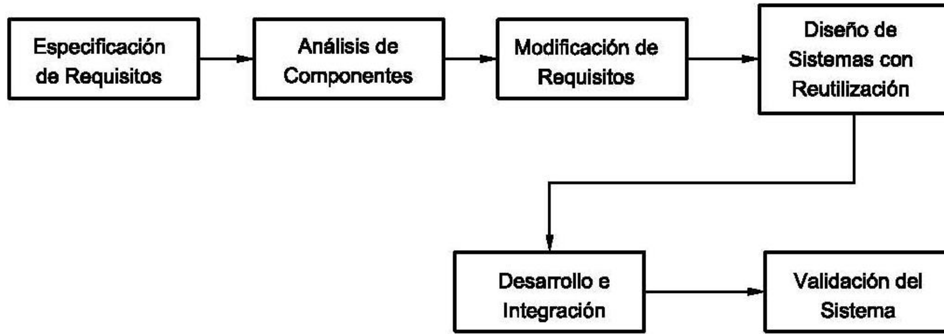

Figura 4.4: Modelo de Proceso Basado en Reutilización.

Espiral

Este modelo fue propuesto por Boehm en 1988. Se caracteriza porque es un modelo incremental, basado en prototipos donde la complejidad de las actividades se incrementan notablemente a medida que nos alejamos del centro de la espiral. Es un modelo orientado a riesgos. La espiral se divide en cuatro sectores, como se muestra en la Figura 4.5. El principal inconveniente que presenta es que es difícil establecer los hitos para determinar si se puede pasar a la siguiente vuelta de la espiral. Este modelo es aconsejable para proyectos complejos con gran incertidumbre [20].

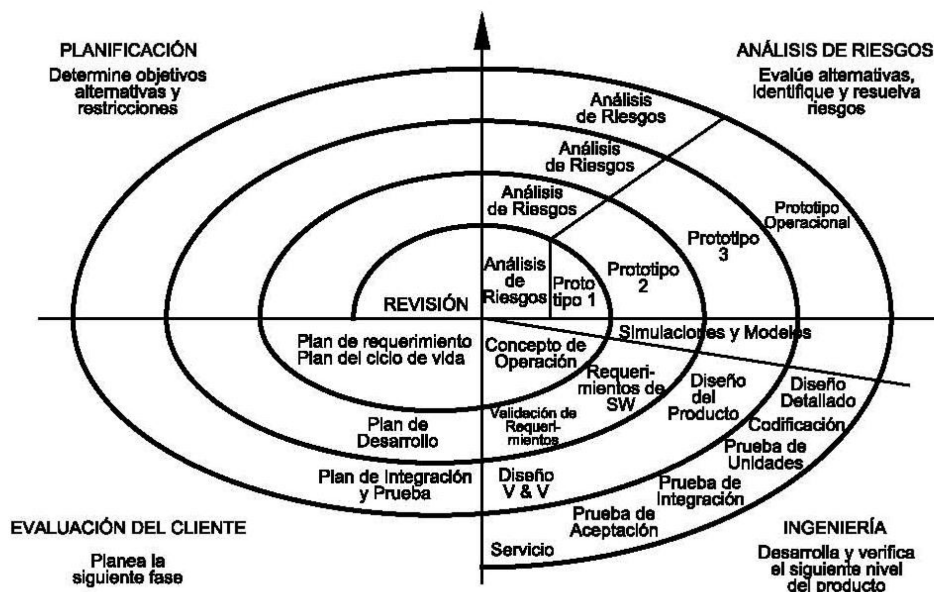

Figura 4.5: Modelo de Espiral.

Proceso Unificado

El Proceso Unificado surge de la unificación de tres metodologías de desarrollo basadas en el paradigma Orientado a Objetos.

- *Object Oriented Software Engineering* (OOSE): Casos de Uso.
- *Booch*: Diseño.
- *Object Modeling Technique* (OMT): Análisis.

Este modelo de proceso se caracteriza por la utilización de UML, está dirigido por casos de uso, se centra en la arquitectura y es iterativo e incremental.

Este proceso se divide en cuatro fases:

- Inicio: Consiste en poner en marcha el proyecto y desarrollar el análisis de negocio.
- Elaboración: Se recopila la mayor parte de los requisitos y se establece una arquitectura sólida.
- Construcción: Tiene como objetivo construir una versión beta.
- Transición: Se realizan las pruebas de aceptación y se obtiene una versión ejecutable.

En cada una de estas fases se realiza un determinado flujo de trabajo, mostrado en la Figura 4.6 [20].

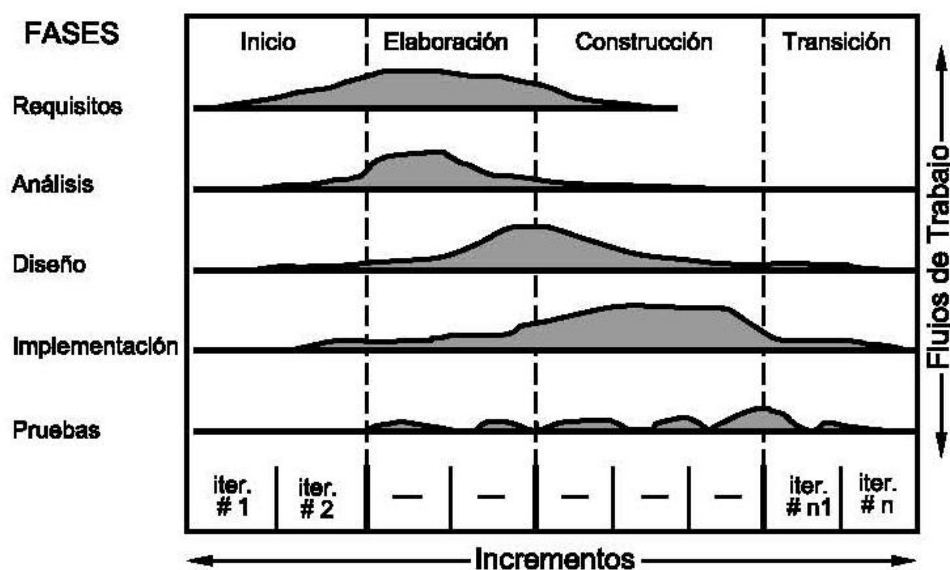

Figura 4.6: Proceso Unificado.

Procesos Ágiles

Los modelos de Procesos Ágiles se definen como un proceso de desarrollo que intenta evitar las burocracias de las metodologías tradicionales centrándose en la gente y en los resultados [10]. Los modelos de Proceso Ágil se caracterizan por:

- Son procesos iterativos e incrementales.
- Se centran en desarrollar *software* que funcione, más que conseguir una buena documentación.
- Se basan en iteraciones cortas en el tiempo, para minimizar riesgos.
- Buscan la colaboración con el cliente.
- Responden a los cambios, más que seguir estrictamente un plan.

Existen multitud de Procesos Ágiles, los más conocidos son la Programación Extrema (XP), Agile Modeling (AM) y Scrum.

4.1.3. Elección de la metodología

De todos los modelos descritos anteriormente, se ha decidido elegir las metodologías ágiles. De entre todas ellas, se ha escogido la metodología de Programación Extrema, en la cual se profundizará más adelante. Esta elección viene ligada a que en este proyecto se producen cambios frecuentes, lo cual hace que sea imposible adaptarlo a una planificación estricta como la que ofrece el Proceso Unificado. Otro motivo es que, en este tipo de proyecto orientado a la experimentación, hay que centrarse más en que el *software* funcione de forma correcta para obtener resultados válidos, que en conseguir una buena documentación. Lo único que no se respeta, es la programación por parejas, debido al carácter individual del proyecto.

4.2. Programación Extrema

La Programación Extrema, como proceso de creación de *software*, nace de la mano de Kent Beck. Es una metodología de desarrollo ágil basada en una serie de valores y de prácticas de buenas maneras, que persigue el objetivo de aumentar la productividad a la hora de desarrollar programas.

Este modelo de programación se basa en una serie de metodologías de desarrollo de *software* en la que se da prioridad a los trabajos que dan resultado directo y que reducen la burocracia que hay entorno a la programación.

Se rige por una serie de valores: comunicación, simplicidad, respeto y humildad. También sigue una serie de reglas como que haya claridad y calidad en el código implementado, que exista un vocabulario común, que se realicen pruebas continuadas, que se trabaje de forma conjunta con el cliente, que se realicen entregas frecuentes para reducir los riesgos y lo más importante, que se entregue *software* que funcione y se vele por el bienestar del programador.

El proceso de desarrollo es iterativo e incremental con un modelo en espiral con prototipos. De esta forma, en cada iteración se irá incorporando cierta funcionalidad al proyecto a la vez que se corrigen los posibles errores que vayan surgiendo durante el desarrollo. Este proceso de desarrollo se divide en cuatro etapas:

- Interacción con el cliente: En la Programación Extrema, el cliente es de gran importancia para el proyecto y llega a formar parte del equipo de programación. De esta forma, se elimina la entrevista previa y el usuario está presente durante todo el proceso.

- Planificación del Proyecto: Se hace una planificación por etapas, definiendo el número de iteraciones que se van a realizar. Por cada iteración hay que entregar al cliente una nueva versión del proyecto.

- Diseño y Desarrollo: Es la fase más importante del proyecto, esta debe ser rápida y sin interrupciones y en la dirección correcta.

- Pruebas: En cada iteración, antes de entregar la versión al cliente, hay que realizan pruebas unitarias.

4.3. Adaptación de la metodología al PFC

En esta sección se va a adaptar la metodología de Programación Extrema, al desarrollo de este proyecto. Este tipo de metodología, se basa en la realización de pequeños hitos, en espacios cortos de tiempo, con entregas frecuentes. A continuación se detalla los hitos, y los flujos de trabajo que se han realizado en cada uno de ellos.

4.3.1. Hitos del proyecto

Un hito es un punto de referencia que marca un evento importante de un proyecto y se usa para supervisar el progreso del mismo. Con cada hito se logra alcanza un objetivo del proyecto. A continuación se muestras los distintos hitos que se han establecido a lo largo de este PFC.

Hito Semana 1

Este es el primer hito del proyecto, los flujos de trabajo llevados acabo son:

Interacción con el Cliente y Planificación

Se produce la primera reunión con ambos tutores, Eduardo y Abraham, para determinar el alcance del proyecto y realizar su planificación. Se decide dividir el proyecto en subobjetivos y se comenta, de forma resumida, qué se va a desarrollar en cada uno de ellos. También se establece la fecha de entrega final y se aporta la documentación necesaria para poder entender el problema a desarrollar. Se decide comenzar con el desarrollo de métodos exactos, más concretamente el método de *backtracking*.

Diseño

En este primer Hito, se han desarrollado las clases Solucion, Instancia, GrafoAleatorio y Backtracking.

- Clase `Solucion`.
  - Descripción: Esta clase se encarga de representar la solución del problema del Vertex Separation.
  - Atributos:
    - `solucion`: Array unidimensional que se encarga de almacenar la solución del problema.
    - `matrizAdyacencia`: Array bidimensional, que representa la matriz de adyacencia de la instancia.
    - `valorSolucion`: Entero que representa la FO de una solución.
  - Operaciones:
    - `evaluarSolucion()`: Calcula la solución del problema.
    - `conecta()`: Comprueba si un vértice v, dado un punto de corte i, tiene adyacentes en el conjunto derecho.
    - `evaluarFuncion()`: Calcula la FO en un punto de corte i determinado.
    - `evaluarSolucionIntervalo()`: Calcula la FO para los puntos de cortes pertenecientes a un determinado intervalo.
    - `obtenerValorMaximo()`: Obtiene la FO máxima de todos los puntos de corte.
    - `getters and setters`: Obtención y modificación de atributos de la clase.

- Clase `Instancia`.
  - Descripción: Esta clase se encarga de representar una instancia del problema.
  - Atributos:
    - `nombre`: Representa el nombre de la instancia.
    - `numVertices`: Representa el número de vértices de la instancia.
    - `numAristas`: Representa el número de aristas de la instancia.
    - `matrizAdyacencia`: Array bidimensional que representa la matriz de adyacencia de la instancia.
  - Operaciones:
    - `getters and setters`: Obtención y modificación de atributos de la clase.
- Clase `Backtracking`.
  - Descripción: Esta clase se encarga de implementar el método exacto de *backtracking*.
  - Atributos:
    - `solucion`: Representa la solución obtenida por el método de *backtracking*.
    - `instancia`: Representa la instancia a resolver.
    - `valorSolucion`: Entero que representa la FO de una solución.
    - `bestFO`: Almacena la mejor FO.
    - `bestSolucion`: Almacena la mejor solución durante el *backtracking*.
  - Operaciones:
    - `ejecutarBacktracking()`: Ejecuta el método de Backtracking.
    - `esFactible()`: Comprueba si una solución es válida.
    - `getters and setters`: Obtención y modificación de atributos de la clase.
- Clase `GrafoAleatorio`.
  - Descripción: Esta clase se encarga de generar grafos conexos y sin ciclos de forma aleatoria.
  - Atributos:
    - `instancia`: Representa el grafo que se va a crear.
  - Operaciones:

- `calcularNumeroAristas()`: Calcula el número de aristas de la instancia de forma aleatoria.
- `generarAristas()`: Asigna a las aristas, un par de vértices.
- `generarGrafo()`: Genera el grafo.
- `getters and setters`: Obtención y modificación de atributos de la clase.

En el Figura 4.7 se muestra el diseño UML correspondiente a este hito, donde se puede apreciar los atributos, métodos y relaciones entre las distintas clases.

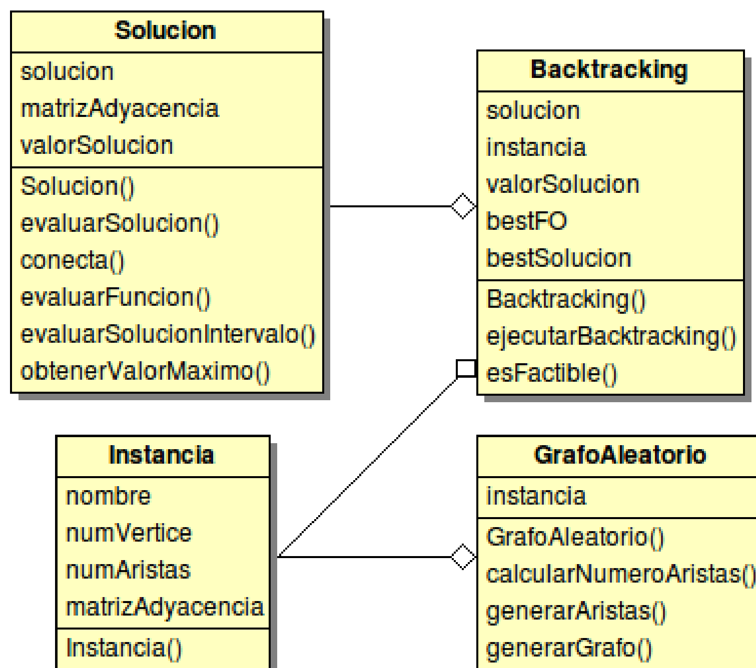

Figura 4.7: Diseño correspondiente al primer hito del proyecto

Pruebas

En cuanto a las pruebas realizadas, lo primero que se probó fue la clase `Solucion`, posteriormente se realizaron varias pruebas sobre la clase `GrafosAleatorios`, comprobando que generaba de forma correcta, grafos conexos y sin cilos. Finalmente se probó la clase `Backtracking`, para ello se utilizaron instancias de poca complejidad, generadas con la clase `GrafosAleatorios`, para comprobar que su funcionamiento era correcto.

Hito Semana 2

Este es el segundo hito del proyecto, los flujos de trabajo llevados acabo son:

Interacción con el Cliente y Planificación

Se produce una reunión, donde se verifica haber cumplido con los objetivos marcados en el primer hito. Una vez comprobado que se han cumplido de forma correcta, se propone realizar un refinamiento del *backtracking* introduciendo una técnica de poda, con la que se espera obtener mejores resultados.

Diseño

En este segundo hito, se ha desarrollado únicamente la clase de `BacktrackingConPoda`.

- Clase `BacktrackingConPoda`.
  - Descripción: Esta clase se encarga de implementar el método exacto de *backtracking* introduciendo la técnica de poda.
  - Atributos:
    - `solucion`: Representa la solución obtenida por el método de *backtracking* con poda.
    - `instancia`: Representa la instancia a resolver.
    - `valorSolucion`: Entero que representa la FO de una solución.
    - `bestFO`: Almacena la mejor FO.
    - `bestSolucion`: Almacena la mejor solución durante el *backtracking*.
    - `cota`: Entero, que representa la estimación de beneficio y que permite aplicar la técnica de poda.
  - Operaciones:
    - `ejecutarBacktrackingConPoda()`: Ejecuta el método de *backtracking* con poda.
    - `esFactible()`: Comprueba si una solución es válida.
    - `getters and setters`: Obtención y modificación de atributos de la clase.

En el Figura 4.8 se muestra el diseño UML correspondiente a este hito, donde se puede apreciar los atributos, métodos y relaciones entre las distintas clases.

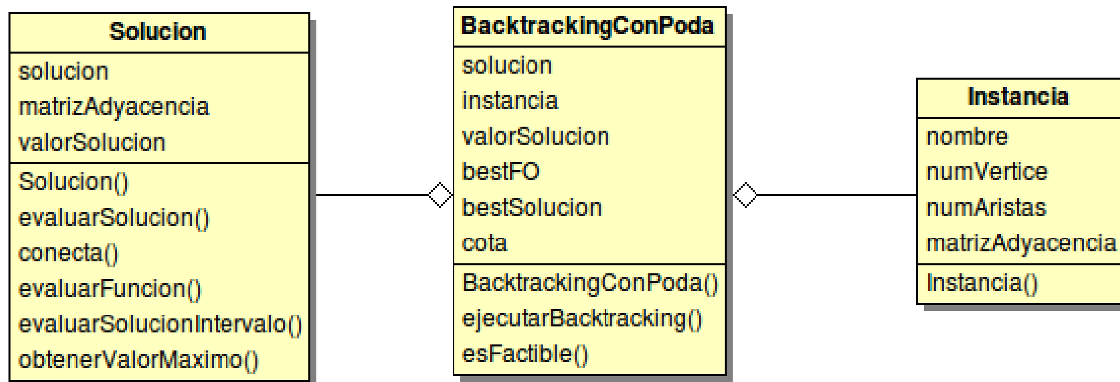

Figura 4.8: Diseño correspondiente al hito dos, donde se especifica el *backtracking* con poda

Pruebas

En cuanto a las pruebas realizadas, para comprobar el correcto funcionamiento del *backtracking* con esta nueva técnica de poda, lo que se hace es comparar los resultados obtenidos con el anterior *backtracking* y validar si se obtiene la misma FO. Por otro lado también se tiene que verificar que los tiempos de cómputo se han reducido de manera significativa.

Hito Semana 3

Este apartado hace referencia al tercer hito del proyecto, donde los flujos de trabajo desarrollados son:

Interacción con el Cliente y Planificación

Se produce una nueva reunión, donde se comprueba que el empleo de técnicas exactas no resuelve de forma eficiente el problema del Vertex Separation para instancias grandes y se decide explorar técnicas aproximadas, empezando con el desarrollo de constructivos heurísticos.

Diseño

En este tercer hito, se han desarrollado las clases `ConstructivoGreedy`, `CGMenorGrado`, `CGMenorAdyacencia`, `CGMayorAdyacencia` y `CGMenorDiferencia`.

- Clase `ConstructivoGreedy`.
    - Descripción: Se trata de una clase abstracta, de la cual heredarán el resto de clases constructivas, que implementa el esquema básico de un constructivo

voraz. Únicamente el resto de clases, tendrá que implementar el criterio de selección.

- Atributos:
  - `solucion`: Representa la solución obtenida por el constructivo.
  - `etiquetados`: Lista de vértices, pertenecientes a la solución.
  - `noEtiquetados`: Lista de vértices, que son candidatos para formar parte de la solución constructiva, pero todavía no han sido seleccionados.
  - `instancia`: Representa la instancia del problema a resolver.
- Operaciones:
  - `estudiarAdyacencia()`: Genera una lista de vértices, calculando sus adyacentes.
  - `obtenerAdyacente()`: Dado un vértice v, calcula sus adyacentes.
  - `generarSolucion()`: Genera la solución del constructivo.
  - `seleccionarVertice()`: Método abstracto, que se encargarán de implementar las clases constructivas que hereden de esta clase, basándose en distintos criterios de selección.
  - `getVerticeMenorAdyacencia()`: Devuelve el vértice de menor adyacencia.
  - `getters and setters`: Obtención y modificación de atributos de la clase.

- Clase `CGMenorGrado`.
  - Descripción: Esta clase implementa un constructivo voraz, cuyo criterio de selección se basa en seleccionar el vértice que menor grado de adyacencia posee.
  - Operaciones:
    - `seleccionarVertice()`: Implementa este método basándose en el criterio de selección descrito anteriormente.
- Clase `CGMenorAdyacencia`.
  - Descripción: Esta clase implementa un constructivo voraz, cuyo criterio de selección se basa en seleccionar el adyacente de los vértices ya etiquetados, que posea menor grado de adyacencia.
  - Operaciones:
    - `seleccionarVertice()`: Implementa este método basándose en el criterio de selección descrito anteriormente.

- Clase `CGMayorAdyacencia`.
  - Descripción: Esta clase implementa un constructivo voraz, cuyo criterio de selección se basa en seleccionar el adyacente de los vértices ya etiquetados, que posea mayor grado de adyacencia.
  - Operaciones:
    - `seleccionarVertice()`: Implementa este método basándose en el criterio de selección descrito anteriormente.
- Clase `CGMenorAdyacencia`.
  - Descripción: Esta clase implementa un constructivo voraz, cuyo criterio de selección se basa en seleccionar el vértice que tenga menor diferencia, entre sus adyacentes no etiquetados y sus adyacentes etiquetados.
  - Operaciones:
    - `seleccionarVertice()`: Implementa este método basándose en el criterio de selección descrito anteriormente.

En el Figura 4.9 se muestra el diseño UML correspondiente a este hito, donde se puede apreciar los atributos, métodos y relaciones entre las distintas clases.

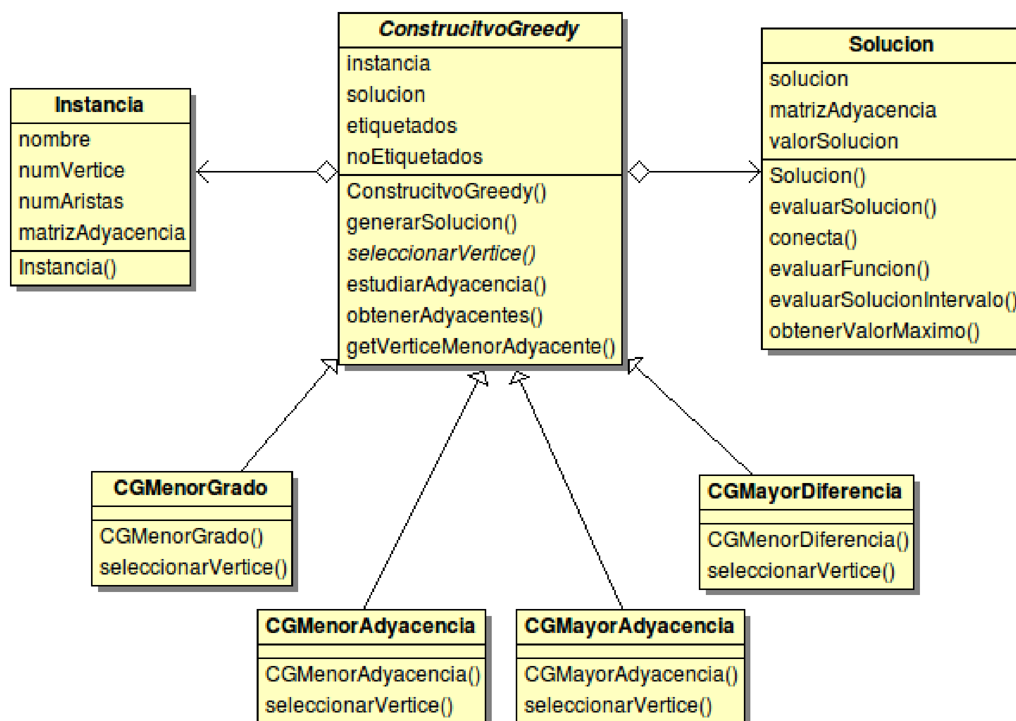

Figura 4.9: Diseño correspondiente al hito tres, referente a los constructivos voraces

Pruebas

Para comprobar el correcto funcionamiento de los distintos constructivos, se han realizado las pruebas con instancias poco complejas, que permiten analizar de forma sencilla, si se está aplicando de manera correcta el criterio de selección.

Hito Semana 4

Este apartado hace referencia al cuarto hito del proyecto, donde los flujos de trabajo desarrollados son:

Interacción con el Cliente y Planificación

Se produce una nueva reunión, donde se comprueba que efectivamente, con los constructivos heurísticos implementados, se obtienen soluciones de buena calidad, en tiempos razonables, y se plantea la idea de mejorar estas soluciones introduciendo una BLE, basado en dos procedimientos distintos (intercambio e inserción).

Diseño

En este cuarto hito, se han desarrollado las clases BusquedaLocalExhaustiva, BLE_Intercambio y BLE_Insercion.

- Clase `BusquedaLocalExhaustiva`.
  - Descripción: Se trata de una clase abstracta, de la cual heredarán las otras dos clases mencionadas anteriormente, que implementa el esquema básico de una búsqueda local, probando todos los vértices y evaluando la solución por completo. Únicamente el resto de clases, tendrá que implementar la forma de generar las soluciones de la vecindad.
  - Atributos:
    - `solucion`: Representa la solución obtenida por el constructivo.
  - Operaciones:
    - `calcularPosicionDestino()`: Calcula de forma aleatoria, la posición destino a la que se va a mover el elemento.
    - `realizarMejora()`: Se encarga de aplicar la búsqueda local a la solución generada por el constructivo voraz.
    - `calcularSolucion()`: Método abstracto, que se encargarán de implementar las clases hijas dependiendo del procedimiento de búsqueda empleado.

- `getters and setters`: Obtención y modificación de atributos de la clase.

- Clase `BLE_Intercambio`.
  - Descripción: Esta clase hereda de la clase `BusquedaLocalExhaustiva` y únicamente tiene que redefinir el método de `calcularSolucion()`, basándose en intercambios (Explicado en detalle en el capítulo de Descripción Algorítmica).
  - Operaciones:
    - `calcularSolucion()`: Modifica una solución dada mediante intercambios.

- Clase `BLE_Insercion`.
  - Descripción: Esta clase, también hereda de la clase `BusquedaLocalExhaustiva`, la única diferencia con respecto a la clase anterior es que tiene que redefinir el método de calcularSolucion() basándose en inserciones (Explicado en detalle en el capítulo de Descripción Algorítmica).
  - Operaciones:
    - `calcularSolucion()`: Modifica una solución dada mediante inserciones.

En el Figura 4.10 se muestra el diseño UML correspondiente a este Hito, donde se puede apreciar los atributos, métodos y relaciones entre las distintas clases.

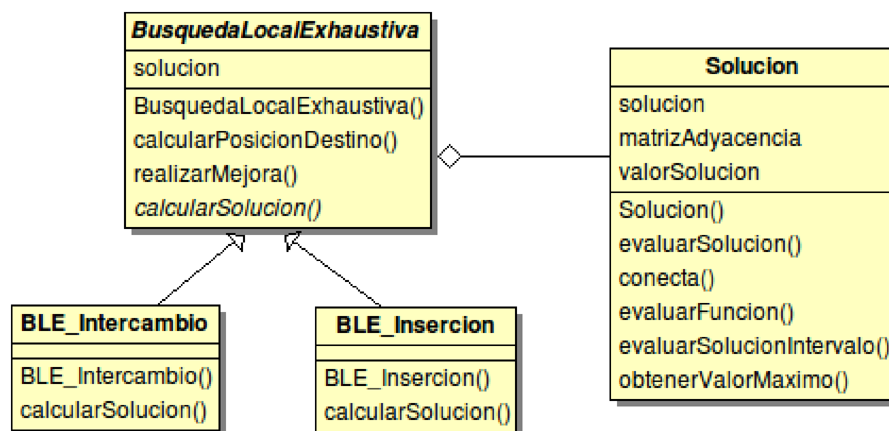

Figura 4.10: Diseño correspondiente al hito cuarto, referente a la BLE

Pruebas

Para verificar que se ha realizado de forma correcta el procedimiento de intercambio y de inserción, se ha probado con instancias de poca complejidad, de un tamaño no superior a diez vértices, para así poder comprobar que todo se hacia de forma adecuada.

Hito Semana 5

En este quinto hito, se van a desarrollar los siguientes flujos de trabajo:

Interacción con el Cliente y Planificación

En la reunión mantenida con los tutores se comprueba que se ha cumplido con los objetivos marcados en el cuarto hito, y se decide refinar la BLE, para conseguir mejores tiempos de cómputo a través de una factorización. Este hecho hace que el diseño mostrado en el cuarto hito, se vea modificado, ya que ahora se hará una clase genérica denominada BusquedaLocal, de la cual hereden la BLE y la BLF, ya que únicamente se diferencian en la evaluación de la solución.

Diseño

En este hito, se va a suprimir la explicación detallada del diseño, ya que la nueva clase desarrollada es igual que la anterior, diferenciándose únicamente en la forma de evaluar la solución. En la Figura 4.11 se muestra cómo quedaría el nuevo diseño, tras añadir esta nueva funcionalidad.

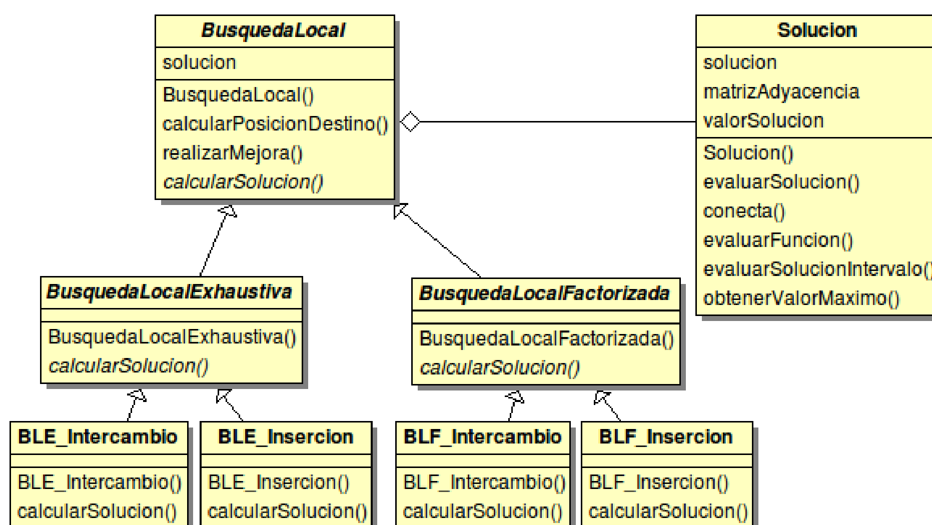

Figura 4.11: Diseño correspondiente al hito quinto, referente a la BLF

Pruebas

Para verificar que se ha realizado de forma correcta la factorización, se comparan los resultados obtenidos de la BLF con la BLE, donde la FO tiene que ser la misma y los tiempos de cómputo tienen que ser notablemente inferiores en la BLF, ya que no se evalúa por completo toda la solución, sino únicamente el intervalo afectado por el cambio.

Hito Semana 6

En este apartado se hace referencia al sexto hito del proyecto, donde los flujos de trabajo desarrollados son:

Interacción con el Cliente y Planificación

Durante la reunión, se analizan y se comparan los resultados de la BLE frente a los resultados de la BLF, y se decide seguir explorando el campo de las búsquedas locales. Se opta por realizar una búsqueda local que no coja como candidatos a mover todos los vértices de la solución, sino un conjunto restringido de ellos.

Diseño

En este sexto hito, se han desarrollado las clases `BusquedaLocalInteligente`, `BLI_Intercambio` y `BLI_Insercion`.

- Clase `BusquedaLocalInteligente`.
    - Descripción: Se trata de una clase abstracta que implementa el esquema básico de una búsqueda local factorizada, añadiendo la funcionalidad de que los candidatos a ser movidos no son todos los vértices de la solución, sino sólo aquellos vértices que se encuentran en las posiciones críticas.
    - Atributos:
        - `solucion`: Representa la solución obtenida por el constructivo.
    - Operaciones:
        - `calcularPosicionDestino()`: Calcula de forma aleatorio, la posición destino a la que se va a mover el elemento.
        - `obtenerVerticesCriticos()`: Obtiene una lista, con los vértices críticos de la solución.
        - `realizarMejora()`: Se encarga de aplicar la búsqueda inteligente a la solución generada por el constructivo voraz.
        - `calcularSolucion()`: Método abstracto, que se encargarán de implementar las clases hijas, dependiendo del procedimiento de búsqueda empleado.
        - `getters and setters`: Obtención y modificación de atributos de la clase.
- Clase `BLI_Intercambio`.

- Descripción: Esta clase hereda de la clase `BusquedaLocalInteligente`, y únicamente tiene que redefinir el método de `calcularSolucion()` basándose en intercambios.
- Operaciones:
  - `calcularSolucion()`: Modifica una solución dada mediante intercambios.

- Clase `BLI_Insercion`.
  - Descripción: Esta clase hereda de la clase `BusquedaLocalInteligente`, y únicamente tiene que redefinir el método de `calcularSolucion()` basándose en inserciones.
  - Operaciones:
    - `calcularSolucion()`: Modifica una solución dada mediante inserciones.

En el Figura 4.12 se muestra el diseño UML correspondiente a este hito, donde se puede apreciar los atributos, métodos y relaciones entre las distintas clases.

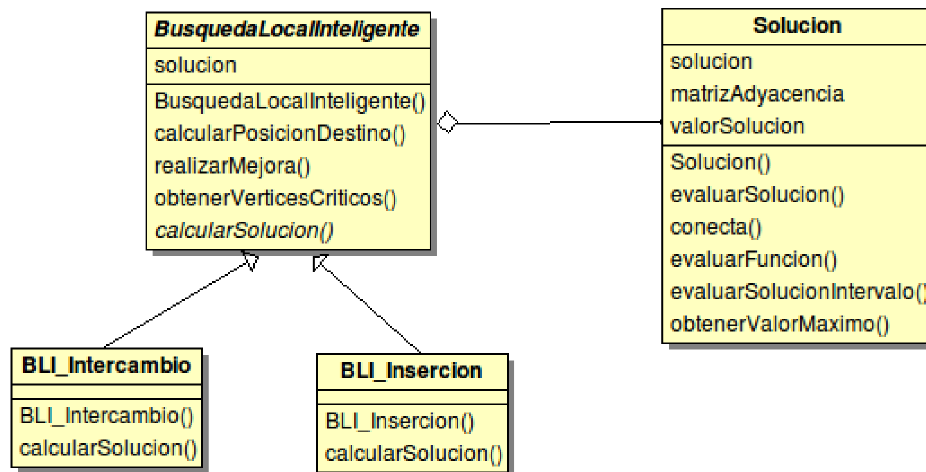

Figura 4.12: Diseño correspondiente al hito sexto, referente a la BLI

Pruebas

Para verificar que se ha realizado todo de forma correcta, se ha probado con instancias de poca complejidad, de un tamaño no superior a diez vértices, para así poder comprobar que los vértices seleccionados, son aquellos que se encuentran en las posiciones críticas.

Hito Semana 7

En este séptimo hito, se han desarrollado los siguientes flujos de trabajo:

Interacción con el Cliente y Planificación

Durante esta reunión, tras analizar y comparar los resultados de las distintas búsquedas locales, se decide ampliar la funcionalidad e investigar el campo de las metaheurísticas, mas concretamente se habla de la implementación de una técnica GRASP.

Diseño

Esta nueva ampliación de funcionalidad hace que el esquema de diseño, referente a la clase ConstructivoGreedy varíe, ya que al implementar el constructivoGrasp, se detecta que se puede crear una clase genérica denominada Constructivo, y que de ella hereden tanto el constructivoGreedy como el constructivoGrasp, ya que comparten prácticamente los mismos métodos y atributos y únicamente se diferencian en el proceso de generar la solución.

El nuevo diseño obtenido tras este cambio, se muestra en la Figura 4.13, que si la comparamos con la Figura 4.9, observamos que los cambios que se han realizado, es que la clase Constructivo, pasa a tener todos los métodos y atributos de la clase Constructivo-Greedy, manteniendo el método de seleccionarVertice() y generarSolucion() de manera abstracta, con el objetivo de que lo implementen sus clases hijas.

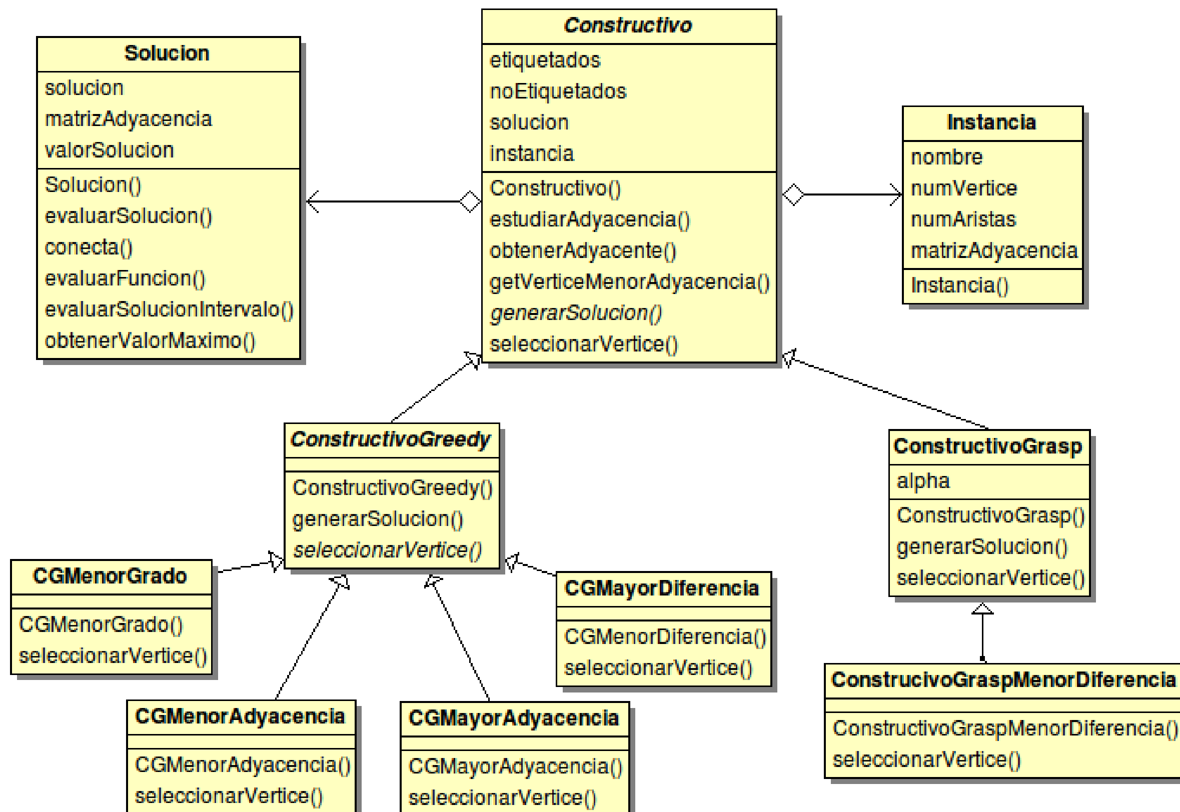

Figura 4.13: Diseño correspondiente al hito séptimo, referente a la modificación del esquema de constructivos

Por otro lado, una vez mostrado cómo quedaría el esquema referente a los constructivos, faltaría establecer el diseño referente al método GRASP, que se basa en un constructivo y en una mejora. Para su desarrollo, se ha decidido crear una clase genérica denominada `GenerarGrasp`, de la cual heredan las clases `GenerarGraspBLF_Intercambio`, `GenerarGraspBLF_Insercion`, `GenerarGraspBLI_Intercambio` y `GenerarGraspBLI_Insercion`, que únicamente se diferencia en el criterio de búsqueda empleado. Todas ellas utilizan el mismo constructivo voraz. Únicamente, se va a detallar la clase genérica `GenerarGrasp`, ya que el resto de clases que heredan de él, sólo tendrán que redefinir el método de `aplicarBusquedaLocal()`.

- Clase `GenerarGrasp`.
  - Descripción: Se trata de una clase abstracta, de la cual heredarán las cuatro clases mencionadas anteriormente, que implementa el esquema básico de un GRASP.
  - Atributos:
    - `instancia`: Representa la instancia del problema.
    - `solucionC`: Representa la solución obtenida por el constructivoGrasp.
    - `numExperimentos`: Representa el número de iteraciones de construcción y mejora que se van a realizar.
    - `bestSolucion`: almacena la mejor solución, obtenida por el método GRASP.
    - `bestFO`: almacena la mejor FO, que se va obteniendo en el método GRASP.
    - `conjuntoSoluciones`: Lista que almacena todas las soluciones generadas por el método GRASP.
    - `almacenarSoluciones`: Booleano, encargado de determinar si es necesario almacenar o no todas las soluciones obtenidas mediante GRASP.
  - Operaciones:
    - `aplicarMetodoGrasp()`: Método que contiene el esquema básico de la técnica GRASP, es decir repetir un número de veces un `ConstructivoGraspMenorDiferencia` más una búsqueda local, y se queda con la mejor solución de todas.
    - `aplicarBusquedaLocal()`: Método abstracto que implementarán el resto de clases hijas, y hace referencia al tipo de búsqueda local empleada.

En la Figura 4.14 se muestra el diseño correspondiente al método GRASP.

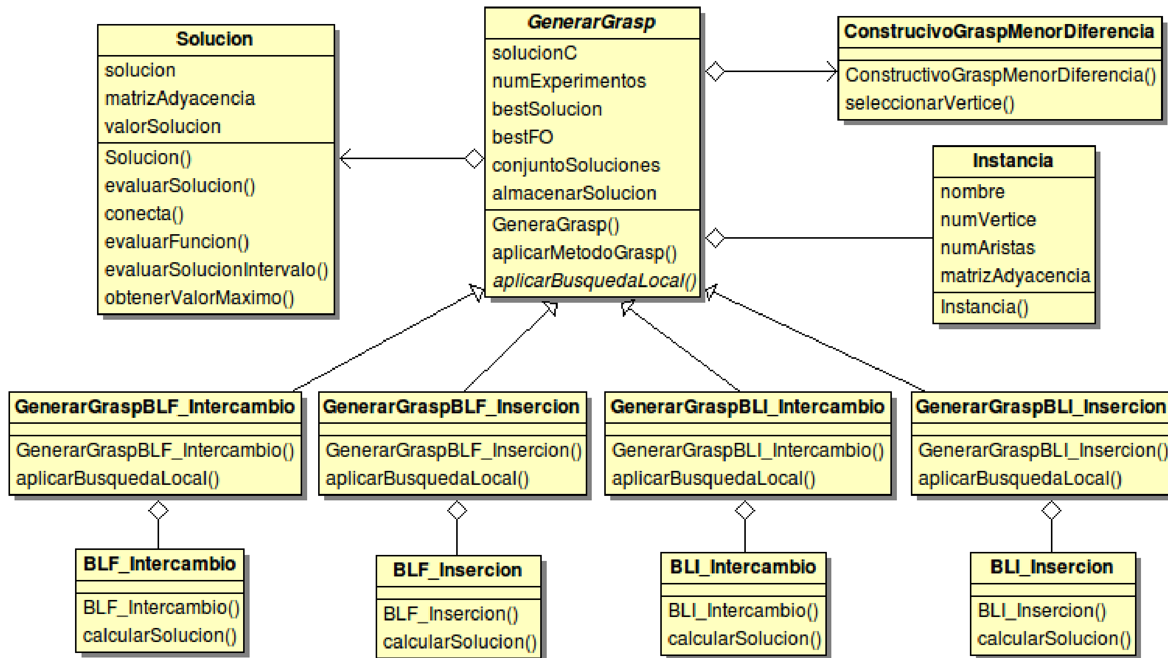

Figura 4.14: Diseño correspondiente al hito séptimo, referente al esquema del método GRASP

Pruebas

Para verificar que el método GRASP, se ha implementado de manera correcta, se han utilizado instancias poco complejas. Además se han comparado los resultados obtenidos mediante GRASP, con los resultados obtenidos en las distintas búsquedas locales, comprobando que los valores de la FO son menores o como mucho iguales a los obtenidos por las búsquedas locales.

Hito Semana 8

En este octavo hito, se han desarrollado los siguientes flujos de trabajo:

Interacción con el Cliente y Planificación

En esta reunión, se decide que además de la técnica metaheurística GRASP, se va a implementar la técnica metaheurística de Path Relinking. Se acuerda, que van a existir dos tipos distintos de Path Relinking, uno de ellos estático y otro dinámico.

Diseño

Como existen dos tipos distintos de Path Relinking, en principio se piensa realizar dos clases distintas, que no estén relacionadas entre sí. Sin embargo, al estudiar el diseño un poca más en profundidad, se observa que ambas clases comparten los mismos atributos, y

tienes los mismos métodos, con algunas funcionalidades distintas. Por ello se decide crear una clase genérica denominada `PathRelinking` y que de ella que hereden dos clases, `PathRelinkingEstatico` y `PathRelinkingDinamico`. Ambas clases serán abstractas, ya que de ambas heredarán dos clases que únicamente se diferenciaran a la hora de realizar el recorrida entre la solución inicial y la solución final, ya que en una de ellas, en el nivel veinte de profundidad se aplicara una BLI_Intercambio.

- Clase `PathRelinking`.
  - Descripción: Se trata de una clase abstracta, de la cual heredarán la clases `PathRelinkingEstatico` y `PathRelinkingDinamico`, e implementa el esquema básico del Path Relinking.
  - Atributos:
    - `instancia`: Representa la instancia del problema.
    - `eliteSet`: Representa las soluciones candidatas para ser utilizadas durante el Path Relinking.
    - `sizeEliteSet`: Tamaño del *Elite Set*.
    - `solucionPathRelinking`: Solución obtenida del proceso de Path Relinking.
  - Operaciones:
    - `ejecutarPathRelinking()`: Método abstracto que se encargará de representar la estructura del método Path Relinking.
    - `generarEliteSet()`: Método abstracto que se encargará de generar las soluciones candidatas del Path Relinking.
    - `realizarIntercambio()`: Realiza el intercambio entre dos posiciones, de una solución dada.
    - `realizarRecorrido()`: Método abstracto que implementarán las clases hijas, dependiendo del tipo de Path Relinking.
- Clase `PathRelinkingEstatico`.
  - Descripción: Este Path Relinking, se caracteriza por que el *Elite Set*, se construye con las diez mejores soluciones del método GRASP y se comparan entre sí.
  - Operaciones:
    - `ejecutarPathRelinking()`: Representa el esquema básico del Path Relinking Estático.

  - `generarEliteSet()`: Genera el *Elite Set* con las diez mejores soluciones de la técnica GRASP.
  - `realizarRecorrido()`: Método abstracto que se encargarán de implementar las clases hijas, dependiendo de si se aplica o no una BLI_Intercambio.

- Clase `PathRelinkingDinamico`.
  - Descripción: Este Path Relinking se caracteriza por que el *Elite Set*, se construye con las diez primeras soluciones del método GRASP, y se comparan de forma aleatoria con el resto de soluciones obtenidas por dicho método.
  - Atributo:
    - `elementosComparados`: Lista con las noventa soluciones restantes obtenidas por la técnica GRASP.
  - Operaciones:
    - `ejecutarPathRelinking()`: Representa el esquema básico del Path Relinking Dinámico.
    - `generarEliteSet()`: Genera el *Elite Set* con las diez primeras soluciones de la técnica GRASP y almacena el resto en la lista de elementosComparados.
    - `realizarRecorrido()`: Método abstracto, que se encargarán de implementar las clases hijas, dependiendo de si se aplica o no una BLI_Intercambio.

Por último. de ambos Path Relinking, estático y dinámico, heredan dos clases, que únicamente se diferenciarían entre sí, a la hora de realizar el recorrido entre la solución inicial y la solución final ya que, en una de ellas, se aplica una BLI basada en intercambios en el nivel veinte de profundidad. Se muestra el diseño UML en la Figura 4.15.

Pruebas

Para verificar que el método Path Relinking, funcionaba de manera correcta, se han utilizado instancias pequeñas para comprobar que tanto el Path Relinking estático como el dinámico generaban de forma correcta el *Elite Set*. Una vez probado esto, también mediante instancias pequeñas, se ha comprobado si se realizaba bien el recorrido entre la solución inicial y la solución final, generando de forma correcta las soluciones intermedias. Por último, para probar que se aplica de forma correcta la BLI en el nivel veinte de profundidad, se han utilizado instancias más grandes, para permitir llegar a dicho nivel de expansión.

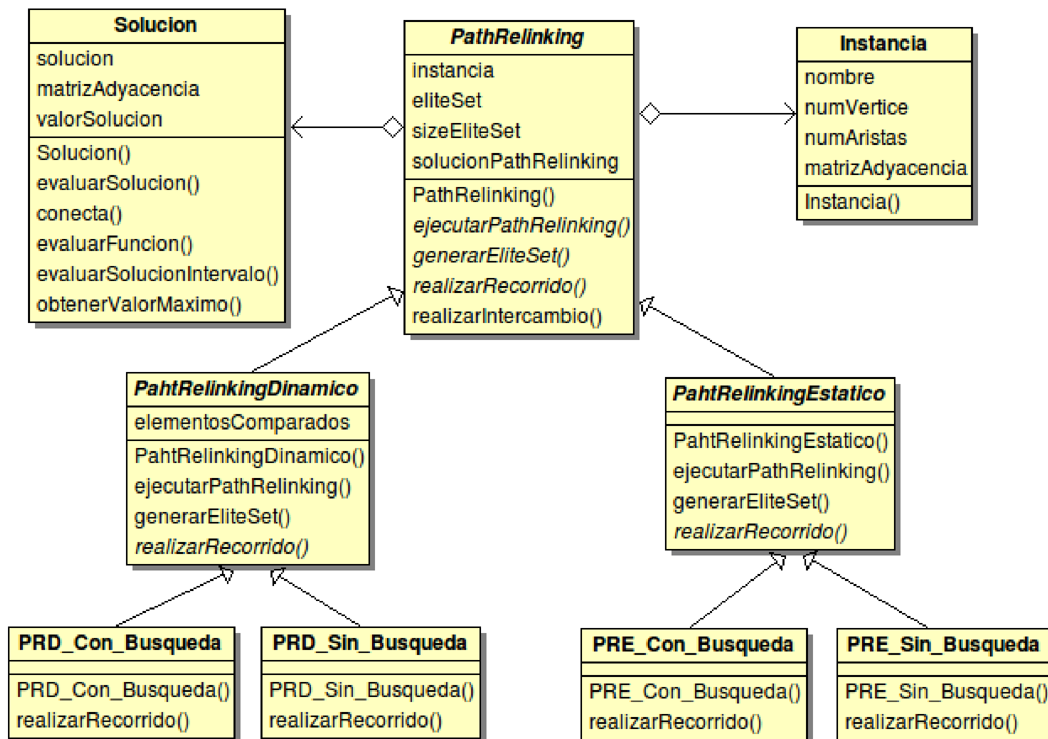

Figura 4.15: Diseño correspondiente al hito octavo, referente al esquema del método Path Relinking

4.4. Tecnología empleada

En esta sección se va a comentar qué lenguaje de programación se ha escogido, cuáles son sus características, qué entorno de desarrollo se ha seleccionado para llevar acabo la implementación y qué procesador de texto se ha utilizado para realizar la memoria.

4.4.1. Java

En cuanto al lenguaje de programación, este proyecto se ha desarrollado bajo la tecnología de Java. Java es un lenguaje de Programación Orientado a Objetos (POO) desarrollado por James Gosling en Sun Microsystems a comienzo de los 90.

Algunas características de este lenguaje son:

- **Simple**, ya que facilita cosas como:
    - Aritmética de punteros.
    - No existen referencias.
    - Registros.
    - Definiciones de tipos.
    - Macros.

   - La necesidad de liberar memoria (Reciclador de memoria dinámica).

- **Orientado a Objetos**: Los objetos agrupan en estructuras encapsuladas tanto sus datos como los métodos que manipulan dichos datos. La POO aporta una serie de características como:

   - Abstracción.
   - Encapsulamiento.
   - Polimorfismo.
   - Herencia.

- **Distribuido**: Permite el intercambio de mensajes entre un conjunto de ordenadores (nodos de procesamiento) a través de la red.

- **Robusto**: Realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución.

- **Independiente de la arquitectura**: El compilador de Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en la que se ejecutará.

- **Portable**

- **Interpretado y Compilado**: Java es compilado, en la medida en que su código fuente se transforma en una especie de código máquina, los *bytecodes*, semejantes a las instrucciones de ensamblador. Por otra parte es interpretado, ya que los *bytecodes* se pueden ejecutar directamente sobre una máquina a la cual se haya portado el intérprete y el sistema de ejecución en tiempo real (*run-time*).

- **Dinámico**: Es dinámico en la fase de enlazado, es decir las clases sólo se enlazan a medida que son necesitadas.

4.4.2. Eclipse

Eclipse es un entorno de desarrollo de código abierto que en su versión básica permite desarrollar aplicaciones Java. Una de sus principales ventajas, es que por medio de instalación de *plugins*, es posible desarrollar código para otro tipo de lenguajes como Haskell, C, C++, Python e incluso LaTeX.

4.4.3. LaTeX

Para el desarrollo de la memoria de este PFC se ha usado LaTeX. Es un sistema de composición de textos que está formado por órdenes (macros) a partir de comandos de TeX. Se ha elegido LaTeX ya que aporta a los textos un aspecto bastante profesional.

Capítulo 5

Resultados experimentales

En este capítulo se van a mostrar y analizar los datos empíricos que se han obtenido al realizar los distintos experimentos sobre un conjunto de instancias determinadas.

5.1. Descripción de los conjuntos de instancias

En esta sección se van introducir las instancias con las cuales se van a llevar a cabo los distintos experimentos sobre los diversos métodos de optimización implementados durante este PFC.

Una instancia es un conjunto de datos que representan un problema para el cual se quiere calcular el mejor Vertex Separation. En concreto, cada una de las instancias es un grafo que cumple con las siguientes características:

- **Finito**: Número de vértices finitos.
- **No dirigido**: El par de vértices que representa una arista está desordenado ({u,v} y {v,u} representan la misma arista).
- **Sin ciclos**: No puede coincidir el comienzo y el fin de una trayectoria.
- **Conexo**: Se puede llegar desde cualquier vértice a cualquier otro siguiendo una secuencia de arcos.

Estas instancias se encuentran divididas en distintos grupos, dependiendo de sus características:

- Harwellboeing
- Small_Instances
- Grafos_Aleatorios
- Grid

5.1.1. Harwellboeing

Se trata de un subconjunto formado por 37 instancias, obtenidas de la librería Matrix Market [16], que presentan las siguientes características:

- El número de vértices de las instancias está comprendido entre 30 y 199 vértices.
- El número de aristas de las instancias está comprendido entre 90 y 2145 aristas.

De todas las instancias pertenecientes al conjunto de Harwellboeing, únicamente se han seleccionado diez de ellas para realizar las experimentaciones preliminares. Estas instancias se muestran en la Tabla 5.1.

Instancia	Número de Vértices	Número de Aristas
bcspwr01.mtx.rnd	39	46
arc130.mtx.rnd	130	715
bcsstk04.mtx.rnd	132	1758
curtis54.mtx.rnd	54	124
gre_185.mtx.rnd	185	650
lund_a.mtx.rnd	147	1151
west0167.mtx.rnd	167	489
gent113.mtx.rnd	104	549
fs_183_3.mtx.rnd	183	701
will199.mtx.rnd	199	660

Tabla 5.1: Instancias Harwellboeing

5.1.2. Small_Instances

Este conjunto consta de 84 instancias y presentan las siguientes características [17]:

- El número de vértices de las instancias está comprendido entre 16 y 24 vértices.
- El número de aristas de las instancias está comprendido entre 18 y 49 aristas.

De todas las instancias pertenecientes a este conjunto, únicamente se han seleccionado diez de ellas para realizar las experimentaciones. Estas instancias se muestran en la Tabla 5.2.

Instancia	Número de Vértices	Número de Aristas
p17_16_24	16	24
p24_17_29	17	29
p40_18_32	18	32
p45_19_25	19	25
p54_20_28	20	28
p63_21_42	21	42
p72_22_49	22	49
p87_23_30	23	30
p92_24_26	24	26
p100_24_34	24	34

Tabla 5.2: Instancias Small_Instances

5.1.3. Grafos_Aleatorios

Este conjunto de instancias, han sido creadas de forma aleatoria. Estas instancias han sido utilizadas para comprobar las técnicas exactas, ya que para instancias de mayor tamaño, los tiempos de cálculo se disparaban.

Este conjunto esta formado por 15 instancias, las cuales se caracterizan por ser instancias pequeñas entre 4 y 18 vértices, y con un número reducido de aristas comprendido entre 5 y 78 aristas. En la Tabla 5.3 se muestran las instancias pertenecientes a este conjunto.

Instancia	Número de Vértices	Número de Aristas
grafo_v4_bt	4	5
grafo_v5_bt	5	10
grafo_v6_bt	6	12
grafo_v7_bt	7	9
grafo_v8_bt	8	25
grafo_v9_bt	9	26
grafo_v10_bt	10	19
grafo_v11_bt	11	18
grafo_v12_bt	12	40
grafo_v13_bt	13	52
grafo_v14_bt	14	38
grafo_v15_bt	15	48
grafo_v16_bt	16	55
grafo_v17_bt	17	64
grafo_v18_bt	18	78

Tabla 5.3: Instancias Grafos_Aleatorios

5.1.4. Grid

Este conjunto de instancias, han sido creadas siguiendo un determinado diseño. Está formado por 14 instancias, las cuales se caracterizan por ser rejillas cuadradas entre 4 y 225 vértices, y con un número de aristas comprendido entre 5 y 420 aristas. Puede verse un ejemplo en la Figura 5.1.

Este conjunto de instancias se ha elegido ya que la FO de los Grid Cuadrados, para el problema del Vertex Separation se conoce a priori, ya que la solución del problema coincide con el lado m del Grid. Esto nos permitirá valorar la calidad de los distintos métodos y criterios utilizados [8]:

$$\text{MINVS(Grid)} = \text{m}$$

De todas estas instancias, se han escogido 10 de ellas. Las instancias seleccionadas se muestran en la Tabla 5.4.

Instancia	Número de Vértices	Número de Aristas
grid_6	36	60
grid_7	49	84
grid_8	64	122
grid_9	81	144
grid_10	100	180
grid_11	121	220
grid_12	144	264
grid_13	169	312
grid_14	196	364
grid_15	225	420

Tabla 5.4: Instancias Grid_cuadrados

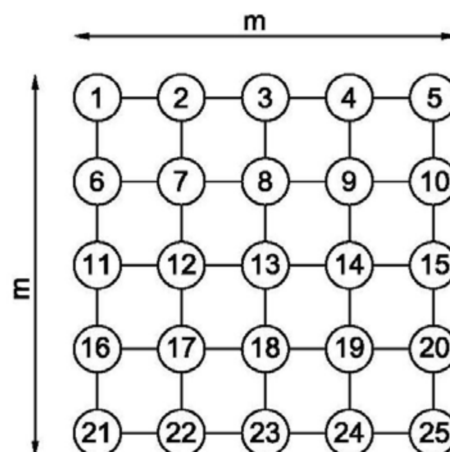

Figura 5.1: Ejemplo de Grid cuadrado

5.2. Experimentación con métodos exactos

En esta sección se van a mostrar los resultados obtenidos de aplicar los métodos exactos de *backtracking* y *backtracking* con poda.

5.2.1. Experimentación Backtracking

El método de *backtracking* únicamente se ha aplicado al conjunto de instancias de Grafos_Aleatorios, ya que para el resto de conjuntos, el tamaño de las instancias hace que la utilización del *backtracking* no sea eficiente, debido a que los tiempos de cómputo son demasiado grandes.

Observando los resultados de la Tabla 5.5, se puede deducir que el método de *backtracking* únicamente es eficiente para instancias de tamaño muy reducido, ya que ha medida que aumenta el tamaño de las instancias, el tiempo computacional crece de manera muy significativa. Sin embargo lo interesante es poder resolver instancias de mayor tamaño en un tiempo de cómputo razonable, objetivo que con este método exacto no se puede conseguir.

Instancia	Nº Vértices	NºAristas	FO	Tiempo(s)
grafo_v4_bt	4	5	2	0,001
grafo_v5_bt	5	10	4	0,002
grafo_v6_bt	6	12	3	0,009
grafo_v7_bt	7	9	2	0,016
grafo_v8_bt	8	25	5	0,015
grafo_v9_bt	9	26	5	0,36
grafo_v10_bt	10	19	4	4,96
grafo_v11_bt	11	18	3	69,63
grafo_v12_bt	12	40	3	1016,23
grafo_v13_bt	13	52	6	13636,4

Tabla 5.5: Experimentación Backtracking para Grafos_Aleatorios

En la Tabla 5.5 se observa como, a partir de instancias cuyo número de vértices es mayor a once, el *backtracking* aumenta su tiempo de ejecución de manera significativa, pasándose de cinco segundos a prácticamente un minuto de ejecución, y aumentando estos tiempos en gran medida según aumenta el número de vértices.

5.2.2. Experimentación Backtracking con poda

Para intentar reducir estos tiempos, se ha implementado el método de *backtracking* con poda, un refinamiento del *backtracking* mucho más eficiente, ya que el número de posibles candidatos se reduce de manera significativa. Para este método exacto, a parte de

experimentar con las instancias del *backtracking*, se han añadido nuevas instancias del conjunto de Grafos_Aleatorios para poder observar de manera más clara su comportamiento a medida que aumentan los tamaños de las instancias.

Instancia	Nº Vértices	NºAristas	FO	Tiempo(s)
grafo_v4_bt	4	5	2	0,001
grafo_v5_bt	5	10	4	0,001
grafo_v6_bt	6	12	3	0,002
grafo_v7_bt	7	9	2	0,003
grafo_v8_bt	8	25	5	0,010
grafo_v9_bt	9	26	5	0,015
grafo_v10_bt	10	19	4	0,5
grafo_v11_bt	11	18	3	0,8
grafo_v12_bt	12	40	3	1,2
grafo_v13_bt	13	52	6	3,4
grafo_v14_bt	14	38	4	4,6
grafo_v15_bt	15	48	7	113,9
grafo_v16_bt	16	55	8	399,2
grafo_v17_bt	17	64	9	2557,4
grafo_v18_bt	18	78	11	12862,3

Tabla 5.6: Experimentación Backtracking con poda para Grafos_Aleatorios

Como se observa en la Tabla 5.6, el *backtracking* con poda resuelve de manera eficiente todas las instancias que resolvía el *backtracking*. Sin embargo se puede observar, que a pesar de que el *backtracking* con poda es más eficiente, al final acaba comportándose de la misma manera que el *backtracking*, la única diferencia está en que este comportamiento se hace visible para instancias más complejas.

Basándonos en las experimentaciones realizadas, podemos concluir que el uso de métodos exactos para la resolución del problema del Vertex Separation únicamente podría emplearse para instancias pequeñas, ya que en cuanto aumentamos su tamaño, estos algoritmos se vuelven poco eficaces.

5.3. Experimentación con métodos heurísticos

Una vez demostrado de forma experimental que los métodos exactos no nos permiten resolver el problema del Vertex Separation de forma eficiente para instancias complejas, se va a proceder a realizar las experimentaciones referentes a los métodos heurísticos. Este tipo de métodos consiguen soluciones de buena calidad en tiempos razonables.

5.3.1. Experimentación con constructivos voraces

Se han implementado cuatro tipos distintos de constructivos voraces, que únicamente se diferencian en el criterio de selección.

Los resultados mostrados en las Tablas 5.7, 5.8 y 5.9 permiten deducir que a priori, sin aplicar una búsqueda local que mejore la solución, el constructivo que obtiene mejores soluciones es el "Constructivo Voraz de Menor Diferencia", en el cual los candidatos son todos aquellos vértices que no ha sido etiquetados y donde el criterio de selección cumple con la restricción de coger, de todo el conjunto de candidatos, el vértice que tenga menor diferencia entre sus adyacentes no etiquetados y sus adyacentes etiquetados.

Constructivo	**Media FO**	**CPU Time(s)**	**Desviación(%)**	**#Best**
Menor Grado	12,9	0,001	49	0
Menor Adyacencia	8,9	0,002	24	2
Mayor Adyacencia	9	0,003	27	3
Menor Diferencia	7	0,015	7	7

Tabla 5.7: Experimentación constructivos voraces para Small_Instances

Constructivo	**Media FO**	**CPU Time(s)**	**Desviación(%)**	**#Best**
Menor Grado	95,2	0,02	59	0
Menor Adyacencia	91,5	1,9	50	1
Mayor Adyacencia	62,9	2,9	37	3
Menor Diferencia	46,2	4,9	14	6

Tabla 5.8: Experimentación constructivos voraces para Harwellboeing

Constructivo	**Media FO**	**CPU Time(s)**	**Desviación(%)**	**#Best**
Menor Grado	57,3	0,016	78	0
Menor Adyacencia	43,1	1,1	73	0
Mayor Adyacencia	40,1	1,2	72	0
Menor Diferencia	14,6	2,26	26	0

Tabla 5.9: Experimentación constructivos voraces para Grid Cuadrados

Se puede observar que el patrón de comportamiento de los constructivos se repite para los distintos conjuntos de instancia.

5.3.2. Experimentación con búsquedas locales

Una vez realizada la experimentación referente a los constructivos voraces, se van a mostrar los resultados que se han obtenido tras aplicar las distintas búsquedas locales

(BLE, BLF y BLI) a las soluciones construidas por los distintos constructivos voraces.

Además de comparar cuál de las búsquedas locales implementadas es más eficiente, también podrá determinarse qué combinación de constructivo y búsqueda local funciona mejor.

Primero se van a mostrar los resultados obtenidos de aplicar la Búsqueda Local Exhaustiva a los conjuntos de Small_Instances, Harwellboeing y Grid.

Experimentación referente a la BLE de intercambio.

En las Tablas 5.10, 5.11 y 5.12 se puede ver respectivamente los resultados obtenidos de aplicar una BLE basada en intercambios, a los conjuntos de instancias Small_Instances, HarwellBoeing y Grid_cuadrados.

Constructivo+BLE_Intercambio	Media FO	CPU Time(s)
Menor Grado	7,1	0,006
Menor Adyacencia	6,1	0,006
Mayor Adyacencia	6,6	0,006
Menor Diferencia	5,7	0,008

Tabla 5.10: Experimentación Búsqueda Local Exhaustiva de intercambio para Small_Instances

Constructivo+BLE_Intercambio	Media FO	CPU Time(s)
Menor Grado	62	66,3
Menor Adyacencia	52,8	76,2
Mayor Adyacencia	52,2	58,6
Menor Diferencia	32,6	72,3

Tabla 5.11: Experimentación Búsqueda Local Exhaustiva de intercambio para Harwellboeing.

Constructivo+BLE_Intercambio	Media FO	CPU Time(s)
Menor Grado	37,5	89,36
Menor Adyacencia	31,5	93,14
Mayor Adyacencia	29,5	127,6
Menor Diferencia	14,5	109,2

Tabla 5.12: Experimentación Búsqueda Local Exhaustiva de intercambio para Grid

Se puede observar que, para todos los conjuntos de instancias, aplicando un constructivo más una BLE de intercambio, el mejor, en todos los casos, es el Constructivo de Menor

Diferencia. Se puede apreciar como *CPU Time* aumenta con respecto a los constructivos voraces, sin embargo la FO es de mejor calidad.

Experimentación referente a la BLE de inserción.

En las Tablas 5.13, 5.14 y 5.15 se puede ver respectivamente los resultados obtenidos de aplicar una BLE basada en inserciones, a los conjuntos de instancias Small_Instances, HarwellBoeing y Grid_cuadrados.

Constructivo+BLE_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	7,8	0,008
Menor Adyacencia	6,2	0,006
Mayor Adyacencia	6,7	0,005
Menor Diferencia	5,7	0,006

Tabla 5.13: Experimentación Búsqueda Local Exhaustiva de inserción para Small_Instances

Constructivo+BLE_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	67	105,47
Menor Adyacencia	67,2	110,6
Mayor Adyacencia	52,6	85,98
Menor Diferencia	33,8	93,67

Tabla 5.14: Experimentación Búsqueda Local Exhaustiva de inserción para Harwell-boeing

Constructivo+BLE_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	41,6	128,39
Menor Adyacencia	35,1	94,74
Mayor Adyacencia	29	282,9
Menor Diferencia	12,8	125,45

Tabla 5.15: Experimentación Búsqueda Local Exhaustiva de inserción para Grid

Se puede apreciar que, para todos los conjuntos de instancias, aplicando un constructivo más una BLE de inserción, ocurre lo mismo que en el experimento anterior, el mejor en todos los casos, es el Constructivo de Menor Diferencia.

Comparación entre la BLE de intercambio e inserción

Observando los resultados obtenidos para la Búsqueda Local Exhaustiva, en ambos experimentos, se pueden alcanzar las siguientes conclusiones:

- El constructivo que obtiene mejores soluciones al aplicarle la búsqueda local, tanto de inserción, como de intercambio, es el Constructivo Voraz de Menor Diferencia.
- Otro detalle relevante es, que a pesar que el Constructivo Voraz de Menor Diferencia es el que más tiempo de cómputo necesita para construir la solución, cuando se aplica una búsqueda local, los tiempos con los demás constructivos se equiparan. Esto es debido a que el Constructivo Voraz de Menor Diferencia, genera soluciones de mejor calidad y, al aplicarle la búsqueda local, encuentra en menor tiempo el óptimo local.
- Por último, destacar que los valores medios de la FO son muy similares en ambos procedimientos, sin embargo, en el tiempo de cómputo existe una mayor diferencia, siendo más rápido el procedimiento de intercambio.

	CVMD+BLE_Intercambio	**CVMD+BLE_Inserción**
Media FO	32,6	33,8
CPU Time(s)	72,3	93,67

Tabla 5.16: Comparación entre la BLE de intercambio y de inserción para el conjunto de las Harwellboeing

En la Tabla 5.16 se muestra una comparativa entre la BLE basada en intercambios y la BLE basada en inserciones, utilizando el Constructivo Voraz de Menor Diferencia, ya que es el que consigue mejores resultados en todos los conjuntos de instancias, aplicado al conjunto de las instancias Harwellboeing, ya que es el conjunto en el que más diferencias significativas se pueden apreciar.

Experimentación referente a la BLF de intercambio.

En las Tablas 5.17, 5.18 y 5.19 se muestran los resultados obtenidos de aplicar una BLF basada en intercambios, a los conjuntos de instancias Small_Instances, HarwellBoeing y Grid_cuadrados.

Constructivo+BLF_Intercambio	**Media FO**	**CPU Time(s)**
Menor Grado	7,1	0,004
Menor Adyacencia	6,1	0,003
Mayor Adyacencia	6,6	0,003
Menor Diferencia	5,7	0,003

Tabla 5.17: Experimentación Búsqueda Local Factorizada de intercambio para Small_Instances

Constructivo+BLF_Intercambio	Media FO	CPU Time(s)
Menor Grado	62	25,32
Menor Adyacencia	52,8	29,8
Mayor Adyacencia	52,2	22,76
Menor Diferencia	32,6	28,71

Tabla 5.18: Experimentación Búsqueda Local Factorizada de intercambio para Harwellboeing

Constructivo+BLF_Intercambio	Media FO	CPU Time(s)
Menor Grado	37,5	34,77
Menor Adyacencia	31,5	36,81
Mayor Adyacencia	29,5	50,71
Menor Diferencia	14,5	49,73

Tabla 5.19: Experimentación Búsqueda Local Factorizada de intercambio para Grid

Se observa que la BLF de intercambio, obtiene la misma media de FO que la BLE de intercambio. Esto es debido a que se evalúan las mismas soluciones. Sin embargo si se observan los tiempos de ejecución, en la BLF de intercambio los tiempos se reducen de manera significativa, esto es debido a que la solución no es evaluada por completo, sino únicamente la zona afectada.

Experimentación referente a la BLF de inserción.

En las Tablas 5.20, 5.21 y 5.22 se muestran los resultados obtenidos de aplicar una BLF basada en inserciones, a los conjuntos de instancias Small_Instances, HarwellBoeing y Grid_cuadrados.

Constructivo+BLF_Inserción	Media FO	CPU Time(s)
Menor Grado	7,8	0,004
Menor Adyacencia	6,2	0,003
Mayor Adyacencia	6,7	0,004
Menor Diferencia	5,7	0,003

Tabla 5.20: Experimentación Búsqueda Local Factorizada de inserción para Small_Instances

En el caso de la BLF de inserción, se puede aplicar el mismo análisis que el de la BLF de intercambio, ya que obtiene el mismo valor medio de FO que la BLE de inserción y, sin embargo, el tiempo de computo es mucho menor.

Constructivo+BLF_Inserción	Media FO	CPU Time(s)
Menor Grado	67	42,49
Menor Adyacencia	67,2	43,63
Mayor Adyacencia	52,6	34,1
Menor Diferencia	33,8	38,02

Tabla 5.21: Experimentación Búsqueda Local Factorizada de inserción para Harwell-boeing

Constructivo+BLF_Inserción	Media FO	CPU Time(s)
Menor Grado	41,6	52,12
Menor Adyacencia	35,1	38,12
Mayor Adyacencia	29	115,2
Menor Diferencia	12,8	51,2

Tabla 5.22: Experimentación Búsqueda Local Factorizada de inserción para Grid

Comparando los resultados de la Búsqueda Local Exhaustiva con la Búsqueda Local Factorizada, podemos obtener las siguientes conclusiones:

- El valor medio de la FO es el mismo, ya que se obtienen las mismas soluciones.
- El tiempo de cómputo de la BLF es menor que el de la BLE, ya que en la BLF no se evalúa por completo toda la solución, sino únicamente la zona afectada por el cambio.

A continuación, en la Tabla 5.23, se muestran los resultados de comparar la BLE frente a la BLF, para el conjunto de instancias Harwellboeing.

	BLE_Intercambio	BLE_Inserción	BLF_Intercambio	BLF_Insercion
Media FO	32,6	33,8	32,6	33,8
CPU Time(s)	72,3	93,67	28,71	38,02

Tabla 5.23: Búsqueda Local Exhaustiva vs Búsqueda Local Factorizada

Experimentación referente a la BLI de intercambio.

En la BLI de intercambio, al igual que ocurre en la BLF de intercambio, el mejor constructivo para todos los conjuntos de instancias es el Constructivo Voraz de Menor Diferencia. Por otro lado, hay que señalar que en la BLI de intercambio la FO es muy parecida a la obtenida en la BLF de intercambio, mientras que el tiempo de ejecución es mucho menor. La explicación de que el tiempo de cómputo de la BLI de intercambio sea menor, se debe a que el número de candidatos son sólo aquellos vértices que se encuentran

en las posiciones críticas. A continuación, en las Tablas 5.24, 5.25 y 5.26, se muestran los resultados obtenidos de aplicar una BLI basada en intercambios a los conjuntos de instancias Small_Instances, Harwellboeing y Grid_cuadrados.

Constructivo+BLI_Intercambio	Media FO	CPU Time(s)
Menor Grado	5,5	0,003
Menor Adyacencia	5,3	0,002
Mayor Adyacencia	6,2	0,001
Menor Diferencia	5,2	0,001

Tabla 5.24: Experimentación Búsqueda Local Inteligente de intercambio para Small_Instances

Constructivo+BLI_Intercambio	Media FO	CPU Time(s)
Menor Grado	58,1	14,27
Menor Adyacencia	56,3	12,6
Mayor Adyacencia	44,3	9,95
Menor Diferencia	36,1	11,27

Tabla 5.25: Experimentación Búsqueda Local Inteligente de intercambio para Harwellboeing

Constructivo+BLI_Intercambio	Media FO	CPU Time(s)
Menor Grado	22,1	24,42
Menor Adyacencia	22,7	13,65
Mayor Adyacencia	16	43,49
Menor Diferencia	12,4	7,2

Tabla 5.26: Experimentación Búsqueda Local Inteligente de intercambio para Grid

Experimentación referente a la BLI de inserción.

A continuación, en las Tablas 5.27, 5.28 y 5.29, se muestran los resultados obtenidos de la BLI mediante inserciones. En estos resultados se pueden aplicar las mismas conclusiones obtenidas en la experimentación anterior, ya que sigue el mismo patrón. La BLI de inserción obtiene una FO muy parecida a la BLF de inserción, sin embargo el tiempo de cómputo es mucho menor.

De los resultados obtenidos de la Búsqueda Local Inteligente podemos destacar varios aspectos relevantes:

- Los tiempos de la Búsqueda Local Inteligente son bastante inferiores a los tiempos de la Búsqueda Local Factorizada, la explicación a esto es que en la BLF los candidatos

a mover son todos los vértices, mientras que en la BLI son sólo aquellos que se encuentran en los puntos de corte críticos, donde la FO toma su valor máximo.

- Por otro lado, el valor medio de la FO en ambas búsquedas es bastante similar, siendo un poco mejor el de la BLF. Por tanto, aún no se puede determinar cuál de ellas será la más apropiada para su integración en la técnica metaheurística tipo GRASP. Para decidirlo habrá que realizar las experimentaciones correspondientes y evaluar la relación entre tiempo y valor medio de la FO obtenida mediante la técnica de GRASP.

Constructivo+BLI_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	9,9	0,002
Menor Adyacencia	7,2	0,0016
Mayor Adyacencia	6,4	0,0015
Menor Diferencia	5,9	0,0016

Tabla 5.27: Experimentación Búsqueda Local Inteligente de inserción para Small_Instances

Constructivo+BLI_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	85,1	4,40
Menor Adyacencia	82,6	3,16
Mayor Adyacencia	54,9	3,66
Menor Diferencia	41,2	2,9

Tabla 5.28: Experimentación Búsqueda Local Inteligente de inserción para Harwellboeing

Constructivo+BLI_Inserción	**Media FO**	**CPU Time(s)**
Menor Grado	41	8,2
Menor Adyacencia	33,8	6,3
Mayor Adyacencia	23,8	38,04
Menor Diferencia	10,9	52

Tabla 5.29: Experimentación Búsqueda Local Inteligente de inserción para Grid

En la Tabla 5.30 se muestra un resumen, donde se van a comparar las distintas búsquedas locales combinadas con los diferentes constructivos. Esta comparación se ha hecho únicamente para el conjunto de instancias Harwellboeing, ya que es el conjunto en el que más diferencias significativas se pueden apreciar. En ella, además de representarse la FO y el tiempo medio de ejecución, se muestra la desviación media y el número de veces que cada una de las búsquedas locales obtiene la mejor solución con respecto al resto.

	Media FO	CPU Time(s)	Desviación(%)	#Best
CVMG+BLF_Intercambio	62	25,32	52	0
CVMG+BLF_Inserción	67	42,49	57	0
CVMG+BLI_Intercambio	58,1	14,27	41	1
CVMG+BLI_Inserción	85,1	4,40	67	0
CVMeA+BLF_Intercambio	52,8	29,8	37	2
CVMeA+BLF_Inserción	67,2	43,63	50	1
CVMeA+BLI_Intercambio	56,3	12,6	37	1
CVMeA+BLI_Inserción	82,6	3,16	58	1
CVMaA+BLF_Intercambio	52,2	22,76	43	1
CVMaA+BLF_Inserción	52,6	34,1	44	0
CVMaA+BLI_Intercambio	44,3	9,95	31	0
CVMaA+BLI_Inserción	54,9	3,66	45	0
CVMD+BLF_Intercambio	32,6	28,71	13	2
CVMD+BLF_Inserción	33,8	38,02	15	5
CVMD+BLI_Intercambio	36,1	11,27	15	5
CVMD+BLI_Inserción	41,2	2,9	28	1

Tabla 5.30: Comparación de la BLF frente a la BLI

En la Tabla 5.30 se puede observar que, la combinación que obtiene soluciones de mejor calidad entre un constructivo y una búsqueda local, es la del CVMD más BLF basada en intercambios, sin embargo, en relación FO/tiempo no es así, ya que la BLI basada en intercambios, utilizando el mismo constructivo mencionado anteriormente es más eficaz en relación FO/tiempo y además, a pesar de tener peor promedio en cuanto a FO, encuentra más veces la mejor solución (#Best).

Para finalizar con el análisis de las búsquedas locales, hay que destacar que queda demostrado experimentalmente que las búsquedas locales obtienen soluciones de mejor calidad que las heurísticas constructivas, sin embargo los tiempos de cómputo son mayores.

5.4. Experimentación con métodos metaheurísticos

Los métodos metaheurísticos guían el proceso de búsqueda incluyendo heurísticas subordinadas. Con este tipo de métodos se evita quedar atrapado en óptimos locales. Por regla general resuelven problemas complejos, consiguiendo soluciones de buena calidad, en tiempos razonables.

5.4.1. Experimentación GRASP

A continuación se van a mostrar los resultados obtenidos de aplicar el método GRASP al conjunto de instancias Harwellboeing. Únicamente se aplica a este conjunto de instan-

cias ya que el resto no aportan diferencias significativas por lo que no permite evaluar los algoritmos.

Como se ha comentado en la descripción algorítmica, el método GRASP consiste en aplicar un constructivo y una mejora repetidas veces y quedarse con la mejor solución de todo el proceso. En las experimentaciones que se han llevado a cabo, este proceso se ha repetido cien veces.

Como constructivo, se va a utilizar el Constructivo Voraz de Menor Diferencia, ya que es el constructivo con el que se ha obtenido soluciones de mejor calidad. En cuanto al proceso de mejora, se van a probar tanto la BLF como la BLI para determinar cuál de las dos es más eficiente en relación Tiempo/Función Objetivo.

Por último también se podrá determinar dentro del método de búsqueda local qué proceso de búsqueda es más eficiente en relación Tiempo/Función Objetivo. Esto nos permitirá seleccionar uno de ellos (Intercambio o Inserción).

GRASP	**FO**	**CPU Time(s)**	**Desviación(%)**	**#Best**
BLF Intercambio	30,3	2664,9	13,8	3
BLF Inserción	30,6	3705,82	12,1	4
BLI_Intercambio	27,5	871,9	6,01	7
BLI_Inserción	31	526,22	15,87	3

Tabla 5.31: Experimentación GRASP con BLF y BLI referente a las Harwellboeing

Según los resultados mostrados en la Tabla 5.31, referentes al método GRASP basado en un Constructivo Voraz de Menor Diferencia y utilizando un proceso de mejora de búsqueda local, se pueden obtener las siguientes conclusiones:

- Utilizando la técnica de GRASP basada en una BLF de intercambios se obtienen mejores soluciones que la basada en inserciones, ya que tanto la FO como el tiempo de ejecución es inferior. Sin embargo las BLF, han sido descartadas, debido a que el tiempo de cómputo es muy elevado.

- La técnica GRASP basada en una BLI de intercambios es la mejor alternativa de todas, ya que no sólo obtiene una mejor FO, sino que el tiempo de cómputo es notablemente inferior. El hecho de que haya una gran diferencia en los tiempos de cómputo, es que en la BLI, los candidatos no son todos los elementos de la solución como ocurre en la BLF, sino que únicamente son aquellos que se encuentran en las posiciones críticas, es decir, en puntos de corte donde la FO toma su máximo valor.

- Por último, queda demostrado experimentalmente que con el empleo de metaheurísticas, la calidad de las soluciones se mejora, aunque la media de tiempos aumente, con respecto al uso de heurísticas.

5.4.2. Experimentación Path Relinking

A continuación, se van a mostrar los resultados obtenidos de aplicar la técnica de Path Relinking al subconjunto de las diez instancias pertenecientes a al conjunto de Harwellboeing.

Path Relinking	**FO**	**CPU Time(s)**	**Desviación(%)**	**#Best**
PRE Sin Búsqueda Local Inteligente	26,6	1314,8	8,4	5
PRE Con Búsqueda Local Inteligente	24,7	1387,5	1,2	7
PRD Sin Búsqueda Local Inteligente	25,5	1751,3	4,7	6
PRD Con Búsqueda Local Inteligente	24,6	2484,3	0,8	9

Tabla 5.32: Experimentación Path Relinking estático y dinámico referente a las Harwellboeing

Como se puede observar en las Tablas 5.32, tanto en el Path Relinking Estático (PRE), como en el Path Relinking Dinámico (PRD), si se le aplica una BLI se obtienen soluciones de mejor calidad, pero el tiempo de cómputo aumenta. Matemáticamente podemos deducir que en el PRE, el tiempo con BLI aumenta un 0,05 %, mientras que la la Función Objetivo disminuye entorno a un 0,04 % con respecto al PRE sin BLI.

Por otro lado, el tiempo de cómputo del PRD con BLI aumenta entorno a un 41 %, mientras que la FO disminuye entorno a un 0,035 % con respecto al PRD sin BLI.

Con estos datos, podemos deducir que en el PRE, mejorar un poco la calidad de la solución aplicando una BLI, apenas supone coste computacional. Sin embargo en el PRD, mejorar un poco la calidad de la solución mediante una BLI, implica un aumento en tiempo de ejecución bastante considerable. Esto se debe a que el conjunto de soluciones utilizados en el PRD es superior.

Haciendo la comparativa entre el PRE y PRD, se observa que la FO que obtiene el PRD es de mejor calidad. El motivo de esto, es que las soluciones que se utilizan, son más heterogéneas que en el PRE. Sin embargo, hay un detalle que hay que señalar, y es que cuando se aplica una BLI, la FO media obtenida por ambos, prácticamente se iguala. Esto se debe a que ambos generan soluciones en la misma vecindad, y aplicando la BLI acaban alcanzando el mismo óptimo local.

Para concluir con el análisis del Path Relinking, comentar que entre las dos alternativas, la más eficiente en relación Tiempo/FO, es la del Path Relinking Estático con BLI basada en intercambios.

5.4.3. Evolución experimental

A continuación, se va a mostrar la evolución obtenida a lo largo de todo este proceso experimental, aplicada al conjunto de las Harwellboeing, para poder ver como evolucionan los resultados con las distintas técnicas empleadas. La evolución que se muestra en la Tabla 5.33 es la siguiente:

- Constructivo Voraz de Menor Diferencia.
- Constructivo Voraz de Menor Diferencia + BLI basada en intercambios.
- GRASP basado en Constructivo Voraz de Menor Diferencia + BLI basada en intercambios.
- Path Relinking estático, aplicando en el nivel veinte una BLI de intercambio.

	Constructivo	**Constructivo+BL**	**GRASP**	**Path Relinking**
Media FO	46,2	36,1	27,5	24,7
CPU Time(s)	4,9	11,27	871,9	1387,5

Tabla 5.33: Evolución de la experimentación

La conclusión que se puede sacar de los datos que se muestran en la Tabla 5.33, es que a medida que se consiguen mejores FO utilizando técnicas metaheurísticas, el tiempo de cómputo aumenta.

5.4.4. Experimentación final

Una vez que se han probado los distintos algoritmos, la elección final ha sido el uso de la técnica Path Relinking Estático, aplicando en él una BLI basada en intercambios en el nivel veinte de profundidad.

En la Tabla 5.34 se muestran los resultados obtenidos para los tres conjuntos de instancias, donde se observa que en el conjunto de las Small_Instances el tiempo de resolución es muy bajo, debido a la poca complejidad de las instancias. Sin embargo en las Harwellboeing el tiempo de cómputo es el mayor de todos, ya que posee instacias de mayor tamaño que el resto de conjuntos.

	Small_Instances	**Harwellboeing**	**Grid cuadrados**
Media FO	3,2	20,3	10,29
CPU Time(s)	0,149	964,4	456,4

Tabla 5.34: Experimentación final

Puede verse en en Anexo B, los resultados específicos para cada una de las instancia de los conjuntos Small_Instances, Harwellboeing y Grid_cuadrados, donde se muestra la FO y el tiempo de ejecución de cada instancia, aplicando la técnica de PRE basado en una BLI de intercambios.

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se van a explicar las conclusiones obtenidas a lo largo del desarrollo de este PFC y los trabajos futuros que pueden realizarse para ampliar este proyecto.

6.1. Conclusiones

En este PFC se han desarrollado algoritmos heurísticos capaces de resolver el problema del Vertex Separation de una forma eficiente. Como primera alternativa, se utilizaron diversas técnicas exactas, concretamente los algoritmos de *backtracking* y *backtracking* con poda. Una vez comprobado que este tipo de técnicas no eran eficientes para instancias grandes, se decidió investigar sobre distintas heurísticas, centrándose en constructivos voraces y búsquedas locales. Posteriormente una vez comprobado que con este tipo de técnicas se obtenían soluciones aceptables, en un tiempo de computo razonable, se decidió explorar algunas técnicas metaheurísticas, en concreto dos de ellas: GRASP y Path Relinking. Con este tipo de técnicas se comprobó que se obtenían soluciones de mejor calidad con respecto al resto de técnicas aplicadas anteriormente.

Se han realizado experimentos mediante los cuales se ha podido determinar que constructivo voraz implementaba el criterio de selección más eficiente. También se han comparado las distintas búsquedas locales implementadas, para decidir cuál de ellas era la más efectiva. Una vez determinada la mejor combinación entre un constructivo voraz y una búsqueda local, se procedió a integrar dicha combinación en la técnica GRASP. Finalmente se experimentó con la técnica Path Relinking, donde quedó demostrado experimentalmente que el PRE basado en una BLI de intercambios era el más efectivo, en comparación con el resto de técnicas.

6.2. Trabajos futuros

Los trabajos futuros que se pueden desarrollar para ampliar este PFC son:

- Explorar otro tipo de metaheurísticas.
- Investigar un método alternativo a la factorización de inserción más eficaz.
- Mejorar el tiempo de cómputo del Path Relinking con BLI mediante la utilización de varios hilos de ejecución.
- Realizar una interfaz gráfica, que permita dibujar la solución del problema.
- Explorar nuevos criterios de selección para el constructivo voraz.

Anexo A

Contenido del CD

En este apéndice, se va a describir el contenido del *CD-Room*, que se ha entregado junto con la memoria. Su contenido es el siguiente:

- Memoria del proyecto en formato PDF.
- Código fuente empleado para el desarrollo del PFC, el cuál contiene los siguientes paquetes:
  - algorithm: Contiene las clases de *backtracking* y *backtracking* con poda.
  - constructive: Contienen los distintos constructivos voraces.
  - grasp: Contiene las clases necesarias para implementar la técnica de GRASP.
  - mejora1: Referente a la Búsqueda Local Exhaustiva.
  - mejora2: Referente a la Búsqueda Local Factorizada.
  - mejora3: Referente a la Búsqueda Local Inteligente.
  - pathRelinking: Contiene las clases necesarias para implementar la técnica de Path Relinking.
- Conjunto de instancias, con las cuales se ha llevado acabo la experimentación. Estas instancias están agrupadas en cuatro conjuntos:
  - Small_Intances.
  - Harwellboeing.
  - Grid_cuadrados.
  - Grafos_Aleatorios.
- Resultados experimentales, en formato excel y de texto.

Anexo B

Resultados por instancia

A continuación, en las Tablas B.1, B.2 y B.3, se van a mostrar los resultados por instancia obtenidos de aplicar la técnica de Path Relinking Estático con una BLI basada en intercambios a todas las instancias pertenecientes a los conjuntos: Grid_cuadrados, Harwellboeing y Small_Instances. En ellas se detalla, el nombre de la instancia, el número de vértices y aristas que la componen, la FO y el tiempo de resolución.

Instancia	Nº Vértices	Nº Aristas	FO	CPU Time(s)
grid_2	4	4	2	0,003
grid_3	9	12	3	0,01
grid_4	16	24	4	0,17
grid_5	25	40	5	0,25
grid_6	36	60	6	0,52
grid_7	49	84	7	4,3
grid_8	64	122	9	8,57
grid_9	81	144	11	63,2
grid_10	100	180	12	135,7
grid_11	121	220	13	234,5
grid_12	144	264	15	670,63
grid_13	169	312	17	1367,3
grid_14	196	364	19	1800,2
grid_15	225	420	21	2104,3

Tabla B.1: Experimentación mediante Path Relinking, para todas las instancias pertenecientes al conjunto Grid_cuadrados

Intancia	Nº Vértices	Nº Aristas	FO	CPU Time(s)
arc130.mtx.rnd	130	715	4	474,000
ash85.mtx.rnd	85	219	8	83,1
bcspwr01.mtx.rnd	39	46	16	2,156
bcspwr02.mtx.rnd	49	59	4	13,563
bcspwr03.mtx.rnd	118	179	10	436,125
bcsstk01.mtx.rnd	48	176	14	4,468
bcsstk02.mtx.rnd	66	2145	65	9,375
bcsstk04.mtx.rnd	132	1758	34	159,594
bcsstk05.mtx.rnd	153	1135	22	314,641
bcsstk22.mtx.rnd	110	254	11	234,984
can__144.mtx.rnd	144	576	9	562,282
can__161.mtx.rnd	161	608	20	711,578
curtis54.mtx.rnd	54	124	7	6,484
dwt__234.mtx.rnd	117	162	7	552,969
fs_183_1.mtx.rnd	183	701	26	1230,3281
fs_183_3.mtx.rnd	183	701	27	1603,515
fs_183_4.mtx.rnd	183	701	26	1348,7390
fs_183_6.mtx.rnd	183	701	24	458,2516
gent113.mtx.rnd	104	549	25	141,344
gre__115.mtx.rnd	115	267	22	50,8547
gre__185.mtx.rnd	185	650	22	1751,234
ibm32.mtx.rnd	32	90	10	0,672
impcol_b.mtx.rnd	59	281	20	7,281
impcol_c.mtx.rnd	137	352	25	732,719
lns__131.mtx.rnd	123	275	19	271,406
lund_a.mtx.rnd	147	1151	25	184,688
lund_b.mtx.rnd	147	1147	28	309,828
mcca.mtx.rnd	168	1662	44	1423,906
nos1.mtx.rnd	158	312	3	1697,579
nos4.mtx.rnd	100	247	9	193,015
pores_1.mtx.rnd	30	103	7	0,281
steam3.mtx.rnd	80	424	7	23,672
west0132.mtx.rnd	132	404	26	500,313
west0156.mtx.rnd	156	371	31	1134,562
west0167.mtx.rnd	167	489	24	1740,500
will57.mtx.rnd	57	127	7	8,813
will199.mtx.rnd	199	660	63	1921,688

Tabla B.2: Experimentación mediante Path Relinking, para todas las instancias pertenecientes al conjunto Harwellboeing

Instancia	Nº Vértices	Nº Aristas	FO	CPU Time(s)
p17_16_24	16	24	4	0,063
p18_16_21	16	21	3	0,062
p19_16_19	16	19	3	0,063
p20_16_18	16	18	3	0,062
p21_17_20	17	20	3	0,063
p22_17_19	17	19	2	0,093
p23_17_23	17	23	3	0,078
p24_17_29	17	29	4	0,079
p25_17_20	17	20	3	0,062
p26_17_19	17	19	2	0,078
p27_17_19	17	19	3	0,078
p28_17_18	17	18	2	0,078
p29_17_18	17	18	2	0,063
p30_17_19	17	19	3	0,078
p31_18_21	18	21	2	0,078
p32_18_20	18	20	3	0,078
p33_18_21	18	21	3	0,094
p34_18_21	18	21	3	0,094
p35_18_19	18	19	2	0,094
p36_18_20	18	20	3	0,109
p37_18_20	18	20	3	0,078
p38_18_19	18	19	2	0,125
p39_18_19	18	19	2	0,078
p40_18_32	18	32	5	0,094
p41_19_20	19	20	2	0,109
p42_19_24	19	24	4	0,094
p43_19_22	19	22	3	0,094
p44_19_25	19	25	4	0,109
p45_19_25	19	25	3	0,094
p46_19_20	19	20	4	0,078
p47_19_21	19	21	3	0,109
p48_19_21	19	21	2	0,110
p49_19_22	19	22	3	0,109
p50_19_25	20	109	3	0,156
p51_20_28	20	28	4	0,156
p52_20_27	20	27	3	0,125
p53_20_22	20	22	2	0,141
p54_20_28	20	28	3	0,141
p55_20_24	20	24	3	0,109
p56_20_23	20	23	4	0,156
p57_20_24	20	24	3	0,125
p58_20_21	20	21	3	0,141
p59_20_23	20	23	3	0,109
p60_20_22	20	22	3	0,125
p61_21_22	21	22	3	0,141
p62_21_30	21	30	4	0,172
p63_21_42	21	42	6	0,141
p64_21_22	21	22	2	0,156

Instancia	Nº Vértices	Nº Aristas	FO	CPU Time(s)
p65_21_24	21	24	3	0,156
p66_21_28	21	28	3	0,141
p67_21_22	21	22	2	0,172
p68_21_27	21	27	3	0,156
p69_21_23	21	23	3	0,156
p70_21_25	21	25	3	0,156
p71_22_29	22	29	4	0,141
p72_22_49	22	49	7	0,156
p73_22_29	22	29	4	0,156
p74_22_30	22	30	3	0,219
p75_22_25	22	25	3	0,203
p76_22_30	22	30	4	0,157
p77_22_37	22	37	5	0,171
p78_22_31	22	31	4	0,172
p79_22_29	22	29	3	0,203
p80_22_30	22	30	4	0,204
p81_23_46	23	46	7	0,187
p82_23_24	23	24	3	0,219
p83_23_24	23	24	2	0,219
p84_23_26	23	26	3	0,187
p85_23_26	23	26	3	0,188
p86_23_24	23	24	2	0,250
p87_23_30	23	30	4	0,171
p88_23_26	23	26	3	0,282
p89_23_27	23	27	3	0,281
p90_23_35	23	35	4	0,234
p91_24_33	24	33	4	0,266
p92_24_26	24	26	3	0,344
p93_24_27	24	27	3	0,218
p94_24_31	24	31	4	0,266
p95_24_27	24	27	3	0,250
p96_24_27	24	27	3	0,250
p97_24_26	24	26	2	0,328
p98_24_29	24	29	3	0,281
p99_24_27	24	27	3	0,282
p100_24_34	24	34	4	0,234

Tabla B.3: Experimentación mediante Path Relinking, para todas las instancias pertenecientes al conjunto Small_Instances

Bibliografía

[1] AIMMS. Página oficial sobre Baron. http://www.aimms.com/features/solvers/baron.

[2] AIMMS. Página oficial sobre Xa. http://www.aimms.com/features/solvers/xa.

[3] Robert Bixby. Página oficial sobre Gurobi. http://www.gurobi.com/.

[4] Edwin K. P. Chong. *An introduction to optimization*. John Wiley and Sons, 2008.

[5] Universidad de Lleida Departamento de Matemáticas. Conceptos básicos sobre grafos. http://web.udl.es/usuaris/p4088280/teaching/terminologia.pdf.

[6] Universidad de Alicante Departamento de tecnología informática y compilacion. Diseño tolerante a fallos de sistemas VLSI. http://www.dtic.ua.es/asignaturas/STF/TEMA6.doc.

[7] Arne Drud. Conopt. *GAMS*, 2006.

[8] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *Universidad Politécnica de Cataluña*, 2002.

[9] J. A. Ellis, H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *University of Victoria*, 2002.

[10] G.Booch, J.Rumbaugh, and Jacobson. *El Lenguaje Unificado de Modelado*. Addison-Wesley, 2005.

[11] Ignacio E. Grossman and Aldo Vecchietti. Dicopt. *GAMS*, 2006.

[12] IBM. Página oficial sobre Cplex. http://www.ilog.com/products/cplex/.

[13] I.Sommerville. *Ingeniería del Software*. Addison-Wesley, 2002.

[14] Daniel Kirsch and Philipp Kühl. Detexify-latex symbol classifier. http://detexify.kirelabs.org/classify.html.

[15] Alejandro Marano and José Luis Martínez Ramos. Introducción a GAMS. *Universidad de Sevilla*, 2009.

[16] Matrix Market. Instancias HarwellBoeing. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/.

[17] R. Martí, V. Campos, and E. Piñana. Branch and Bound for the Matrix Bandwidth Minimization. *European*, 2008.

[18] Microsoft. Página oficial sobre Excel. http://www.MicrosoftStore.com/ES/Excel.

[19] Abraham Duarte Muñoz, Juan José Pantrigo Fernández, and Micael Gallego Carrillo. *Metaheurísticas*. Universidad Rey Juan Carlos, Servicio de Publicaciones, 2007.

[20] S.R.Schach. *Ingeniería del Software Clásica y Orientada a Objetos*. McGraw-Hill, 2002.

[21] Gonzalo Sainz Trápaga. Técnicas algorítmicas. http://www.gonzalosainztrapaga.com.ar/media/algoritmos/tecnicas.pdf.

[22] Ziena. Página oficial sobre Knitro. http://www.ziena.com/knitro.htm.