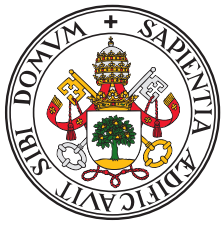


Universidad de Valladolid

Facultad de Ciencias

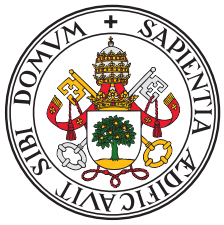
Trabajo Fin de Grado

Grado en Estadística

Métodos escalables de resolución para el Dial-a-Ride Problem (DARP)

Autor:

Sergio García Prado



Universidad de Valladolid

Facultad de Ciencias

Trabajo Fin de Grado

Grado en Estadística

Métodos escalables de resolución para el Dial-a-Ride Problem (DARP)

Autor:

Sergio García Prado

Tutor:

Jesús Saez Aguado

Abstract

[TODO]

Resumen

[TODO]

Este trabajo puede ser consultado a través del siguiente enlace:
`https://github.com/garciparedes/tfg-dial-a-ride-problem`

Agradecimientos

[TODO]

Índice general

Capítulo 1

Introducción

1.1. Objetivos

[TODO]

1.2. Metodología

[TODO]

1.3. Aplicaciones

[TODO]

Capítulo 2

Formulación del Problema

2.1. Introducción

El problema *Dial-a-Ride* (o *DARP* en modo abreviado) representa una de las modelizaciones más interesantes en el ámbito de los problemas de *optimización combinatoria*. Esto se debe a la gran cantidad de situaciones del mundo real que pueden ser representadas siguiendo dicho esquema. Sin embargo, antes de profundizar en los aspectos más detallados que caracterizan este modelo, es necesario describir el contexto del mismo así como la clase problemas a la cual pertenece. Una vez se haya completado dicha tarea, se estará en condiciones suficientes para poder describir tanto la versión básica como las extensiones más interesantes, tando desde el punto de vista de los aspectos matemáticos, como desde la cantidad de situaciones reales que permiten resolver.

En los últimos años, el número de compañias que basan su modelo de negocio en alguna variante relacionada con el problema *Dial-a-Ride* ha crecido de manera desmesurada. Algunos ejemplos son aquellas basadas en el reparto de comida (u otros productos) desde los establecimientos hasta la casa de los solicitantes, o la modernización del sector del transporte privado en ciudad gracias a la solicitud de viajes desde el dispositivo móvil. Es importante darse cuenta de que dichas compañías llevan a cabo la planificación en tiempo real ya que en contadas ocasiones los clientes solicitan sus servicios con demasiada antelación. Sin embargo, para poder desarrollar soluciones interesantes sobre entornos en tiempo real, lo primero es poder comprender el problema de manera detallada en su versión estática, para después poder llevar a cabo las simplificaciones y extensiones pertinentes que permiten la toma de decisiones de manera eficiente en entornos dinámicos. Por tanto, este capítulo se centra especialmente en la descripción del modelo estático.

En cuanto a la organización del capítulo, en el apartado 2.2 se describe de manera detallada el problema *Dial-a-Ride* desde el punto de vista de la jerarquía de clases a la cual pertenecen siguiendo un enfoque descendente (o *top-down*) en los apartados 2.2.1 a 2.2.6. Seguidamente, en el apartado apartado 2.3 se indica la notación que se pretende seguir a lo largo del documento

(y que pretende ser utilizada a modo de referencia en capítulos posteriores). El siguiente paso es determinar la *Formulación Básica*, lo cual se lleva a cabo en el apartado 2.4. Finalmente, en el apartado 2.5 se describen brevemente los aspectos más importantes del capítulo.

2.2. Contextualización del problema

A continuación se procede a describir la clase de problemas matemáticos a la cual pertenece el problema *Dial-a-Ride*. Dicha descripción se llevará a cabo de fuera hacia dentro, esto es desde la categoría de problemas más amplia hasta la más concreta, pasando por una breve contextualización así como ejemplificación de problemas similares. En la figura 2.1 se muestra un breve resumen gráfico acerca de las clases de problemas que se describen.

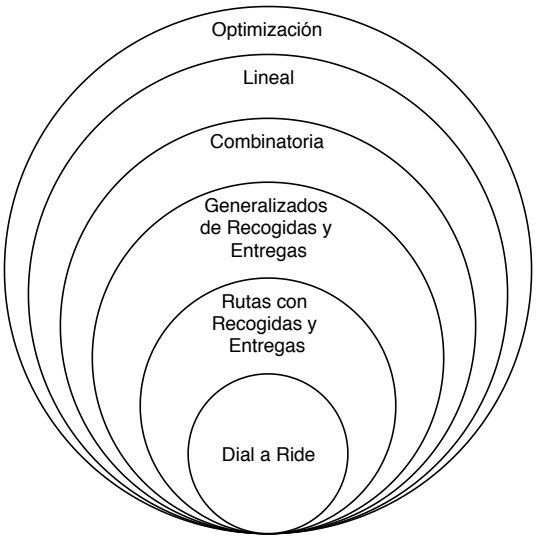


Figura 2.1: Contextualización del *Problema Dial-a-Ride* desde el punto de vista de la jerarquía de problemas a la que pertenece.

2.2.1. Problemas de Optimización

La clase de *problemas de optimización* representa una de las áreas de investigación más interesante en la actualidad, ya que muchas de las innovaciones obtenidas en dicho campo permiten resolver problemas aplicables al mundo real de manera práctica que antes únicamente podían ser resueltos teóricamente. En concreto, los problemas de optimización son aquellos que se basan en la minimización (o maximización) de una determinada función objetivo (posiblemente vectorial, lo cual define modelos multiobjetivo) de manera que se satisfaga un conjunto de restricciones previamente fijadas sobre un conjunto de variables de decisión que afectan mutuamente a la satisfacibilidad de las restricciones y el valor de la función objetivo.

Dichas variables de decisión pueden ser tanto categóricas como numéricas (discretas o continuas), lo cual genera una gran cantidad de subproblemas diferences (Nótese que las variables categóricas

con k niveles diferentes pueden ser representadas de manera sencilla a partir de $k-1$ variables binarias). De la misma manera, tanto el valor de función objetivo como las restricciones pueden tener una naturaleza muy diferente: estas pueden estar formadas por funciones lineales de las variables de decisión, como por complicadas funciones no lineales que complican el proceso de obtención del valor óptimo del problema. De manera matemática, la ecuación 2.1 define la formulación de optimización, donde tanto las funciones $f_i(\cdot)$ como $g_k(\cdot)$ son funciones arbitrarias que proyectan el vector de variables de decisión n-dimensional $\mathbf{x}$ en un espacio unidimensional (generando un valor escalar).

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{Minimizar}} \quad & f_i(\mathbf{x}), && \forall i \in \{1, ..., I\} \\ \text{sujeto a} \quad & g_k(\mathbf{x}) \leq 0, && \forall k \in \{1, ..., K\} \end{aligned} \tag{2.1}$$

Ecuación 2.1: Formulación del modelo de Optimización General

Muchos de los problemas que resolvemos a diario en nuestra vida cotidiana son en cierta medida *problemas de optimización*, desde qué elementos decidimos añadir a nuestra mochila cada día (basados en restricciones de capacidad, funciones objetivo de utilidad y variables de decisión binarias) hasta el la detección del rostro por nuestros teléfonos móviles para aplicar un filtro de la manera más realista posible en una videollamada (basados en restricciones de forma, funciones objetivo multidimensionales y millones de variables de decisión numéricas).

2.2.2. Problemas de Optimización Lineal

Una de las categorías de problemas de optimización más ampliamente estudiados por su relativa simplicidad (ya se han desarrollado métodos capaces de obtener soluciones óptimas en un número reducido de pasos) y su gran capacidad de modelización ante muchas situaciones del mundo real son los *problemas de optimización lineal*. Dichos problemas se caracterizan por estar compuestos por variables de decisión compuestas por transformaciones lineales respecto de la función objetivo. Esto quiere decir que tanto el valor de la función objetivo como las posibles restricciones escritas en forma de desigualdades están compuestas por sumas de las variables de decisión multiplicadas por determinados pesos.

En la ecuación 2.2 se muestra a modo de ejemplo la formulación de un problema de optimización lineal (del cual se hablará posteriormente). Como se puede apreciar, en este caso las funciones arbitrarias definidas en la formulación general han sido sustituidas por transformaciones lineales respecto de las variables de decisión. Para la resolución de problemas de este tipo se han desarrollado una gran cantidad de métodos, entre los que destaca un algoritmo altamente eficiente el cual se conoce como *Simplex* [KM70]. Este algoritmo se basa en pivotaje entre soluciones de manera que tras cada iteracción se llegue a una solución igual o mejor. Una de las mayores ventajas de la

formulación de un problema como lineal es que algoritmos como *Simplex* proporcionan garantias de optimalidad al alcanzar el valor óptimo al terminar completamente su ejecución.

$$\begin{aligned} \text{Minimizar} \quad & \sum_{i=1}^{N}\sum_{j=1}^{M} c_{ij}x_{ij} \\ \text{sujeto a} \quad & \sum_{i=1}^{N} x_{ij} = s_i, \quad \forall i \in \{1,...,N\} \\ & \sum_{j=1}^{M} x_{ij} = d_j, \quad \forall j \in \{1,...,M\} \\ & x_{ij} \geq 0, \quad \forall i \in \{1,...,N\}, \forall j \in \{1,...,M\} \end{aligned} \tag{2.2}$$

Ecuación 2.2: Formulación de un modelo de *Optimización Lineal*. En concreto, el *Problema de Transporte*.

A pesar de que el algoritmo *Simplex* siempre proporcione resultados óptimos, en algunas ocasiones no se posee la capacidad suficiente de cálculo para llegar a la mejor solución. Por lo tanto, se han desarrollado una gran cantidad de métodos conocidos como *heurísticos* (de los cuáles se hablará más en detalle en el los apartados 3.3 y 3.4) que a pesar de proporcionar unos buenos resultados, no ofrecen ninguna garantia de optimalidad. Por contra, también existen otros métodos exactos que son capaces de llegar al valor óptimo utilizando menor cantidad de recursos (algunos de los cuales se describen en el apartado 3.2).

Antes de describir algunos de los ejemplos y aplicaciones reales más destacadas basadas en *problemas de optimización lineal*, es necesario describir unos de los problemas más populares de esta categoria, el cual se conoce como *problema de transporte* y se caracteriza por permitir representar de manera matemática la tarea sobre cómo distribuir un conjunto de recursos procedentes de N puntos de origen hasta M puntos de destino, donde cada trayecto tiene un coste diferente, y cada origen y destino unas capacidades de oferta y demanda. La formulación sobre dicho problema se corresponde con la utilizada a modo de ejemplo en la ecuación 2.2.

Sin embargo, los problemas de optimización lineal permiten representar una amplia cantidad de situaciones de nuestra vida diaria. Entre ellos se encuentran los *problemas de mezclas* (donde se pretende generar un compuesto con unas ciertas características a partir de la combinación de otros tratando de reducir los costes) aunque los problemas de optimización lineal también son de gran utilidad en el ámbito de la economia y los estudios de mercado, permitiendo representar de una manera relativamente sencilla el comportamiento de los clientes ante cambios de precio u otras variables más elaboradas.

2.2.3. Problemas de Optimización Combinatoria

Uno de los factores más relevantes cuando se trata de *problemas de optimización lineal* se refiere a l naturaleza de las variables de decisión, en que (como se ha comentado anteriormente) estas pueden ser de una naturaleza continua o discreta. En este sentido, los problemas pueden clasificarse en 4 categorías: problemas de optimización continua puros (donde existen métodos muy eficientes que permiten ser resueltos en un tiempo razonable), problemas de *optimización entera*, problemas de optimización binaria (o de *optimización combinatoria*) y problema de optimización mixtos (donde mezclan ambos tipos de variables). Para profundizar más en el tema de la *optimización entera* y *optimización combinatoria* se recomienda [WN14]

A pesar de que la exigencia de que las variables de decisión sean discretas en un primer momento puede parecer muy sencilla en un primer momento (por ser muy intuitiva a nivel conceptual), esta complica mucho la labor de optimización del problema. En modo abstracto, esto se debe a que si el espacio de soluciones de un problema de optimización lineal era visto como un prima en un espacio n-dimensional (donde n es el número de variables de decisión) y los puntos de interés se refieren a los vértices, que presentarán el máximo/mínimo valor óptimo, en el caso de las variables discretas el prima se transforma en una estructura "pixelada" lo cual incrementa de manera exponencial el número de vértices (y por tanto de puntos de evaluación) del problema en la búsqueda del valor óptimo.

Las dificultades descritas en el párrafo anterior provocan que la clase de problemas de optimización de variables discretas, entre los que se encuentra la de los problemas de optimización combinatoria (a la cuál a su vez pertenecen los de rutas, de los que se hablará posterirmente), hacen que el espacio de soluciones se extienda de tal manera que para casos relativamente pequeños, este sea inabordable. A pesar de existir métodos exactos (de los cuales se hablará en el apartado 3.2), la mayor parte de la literatura sobre este tema se ha dedicado a la investigación de métodos que a pesar de no ofrecer garantías de optimalidad, proporcionan un buen acuerdo entre eficiencia computacional y calidad de las soluciones obtenidas (los cuales se detallan en los apartados 3.3 y 3.4).

$$\begin{aligned} \text{Maximizar} \quad & \sum_{i=1}^{N} u_i x_i \\ \text{sujeto a} \quad & \sum_{i=1}^{N} w_i x_i \leq W, \qquad \forall i \in \{1, ..., N\} \\ & x_i \in \{0, 1\}, \quad \forall i \in \{1, ..., N\} \end{aligned} \tag{2.3}$$

Ecuación 2.3: Formulación de un modelo de *Optimización Combinatoria*. En concreto, el *Problema de la Mochila*.

Los problemas de *optimización combinatatoria* se caracterizan porque las variables de decisión del modelo son de caracter binario. Dicha razón permite representar situaciones muy interesantes y diversas del mundo real, lo cual ha hecho que los campos de aplicaciones de este tipo de problemas sean desde la generación de rutas de vehículos con garantías de conectividad, hasta problemas de decisión relacionados con la toma o no de una determinada acción. Dentro de los problemas de decisión, existe uno que destaca sobre el resto por su relativa simplicidad teórica, pero a la vez extremada practicidad, el cual se conoce como *problema de la mochila*. A pesar de tener muchas vertientes y extensiones, la idea básica de este es la de maximizar el valor en el proceso de selección de un subconjunto de elementos, dadas unas limitaciones de capacidad y un valor determinado para cada producto. La "dificultad" de este modelo reside en que los elementos no pueden ser escogidos parcialmente, por lo que surge un problema de combinatoria entre los elementos que añadir o no al subconjunto que forma la solución. A modo de ejemplo, en la ecuación 2.3 se muestra la versión más básica del *problema de la mochila*, donde tal y como se puede apreciar, las variables de decisión tan solo pueden tomar valores $\{0, 1\}$ lo cual determina la presencia o no del elemento i-ésimo en el subconjunto solución.

[TODO: describir problemas NP]

Tal y como se ha remarcado a lo largo de todo el apartado, es especialmente importante indicar que la mayor dificultad surgida al añadir las restriciones en el soporte de las variables de decisión genera un mayor espacio de puntos de evaluación, lo cual complica extremadamente los métodos de generación de soluciones. Por contra, esta dificultad amplia en gran medida la capacidad de representación de los modelos, lo cual es algo beneficioso ya que permite aprovechar en mayor medida recursos que de otra forma serían imposibles de aprovechar de la misma manera.

2.2.4. Problemas Generalizados de Recogidas y Entregas

Los problemas de optimización descritos hasta ahora permiten abarcar una gran cantidad de situaciones basadas en tratar de aprovechar en mayor medida uno o más recursos disponibles sobre una colección de restricciones. En el caso de los problemas de *optimización combinatoria*, dicho proceso de aprovechamiento de recursos puede ser visto como un conjunto de problemas de decisión codependientes entre si sobre la utilización (o no) de cada uno de ellos. Sin embargo, en dichos problemas la relación de codependencia no queda fijada de manera explícita, sino que esta es marcada por cada problema concreto. Entonces, tiene sentido pensar que distintos problemas puedan presentar similitudes desde el punto de vista de las relaciones de dependencia entre sus variables de decisión. Para el resto de subclases de problemas la descripción se apoya en cierta medida en la jerarquía descrita en [PDH08] por su sencillez conceptual pero a la vez potencia desde el punto de vista de los problemas que engloba.

Los *problemas generalizados de recogidas y entregas* se estudiaron inicialmente a partir del *problema del viajante*, del cual hay registros de su existencia desde el siglo XIX. A pesar de ello, uno de los primeros papers que describen de manera detallada su formulación [MTZ60] no se publicaría hasta los años 60. Con el paso de los años se ha desarrollado un amplio marco de conocimiento alrededor de este tipo de problemas, entre los que destaca especialmente [TV02]. Estos problemas surgen precisamente como una subcategoría englobada dentro del conjunto de los problemas de *optimización combinatoria*. La restrición que más caracteriza este tipo de problemas se basa en el mantenimiento tanto de orden en el cumplimiento de tareass como del mantenimiento de un concepto de estado a lo largo dicho proceso. Dicho estado en algunos subproblemas es muy débil, mientras que en otros representa una componente muy importante durante el proceso de resolución del mismo.

En muchos casos, los *problemas generalizados de recogidas y entregas* (o *Vehicle Routing Problems*) pueden resumirse de la siguiente manera: Dada una colección de tareas a realizar en un determinado lugar del espacio (con posibles restricciones de orden, duración, capacidad, etc.) así como una colección de vehículos cuya función es moverse de un punto a otro en del espacio generado por las tareas (con posibles restricciones de duración, capacidad, etc.) generando un determinado incremento de coste, se trata de encontrar la *ruta* (o *tour*) para cada vehículo que satisface más y/o con menor coste las tareas disponibles. Entonces, dicho marco engloba una gran grantidad de problemas de optimización caracterizados por dicha restricción, donde están contenidos problemas relativamente dispares, desde el *problema del viajante* (que se describirá brevemente a continuación) como el *problema de recogidas y entregas* (que se describirá de manera más detallada en próximos apartados).

Una de las características más importantes desde el punto de vista de la resolución en este tipo de problemas es la estructura secuencial que estos presentan, así como el elevado número de variables de decisión necesario para su formulación formal. Dichas características conllevan interesantes peculiaridades que convierten en prácticamente inabordable la búsqueda de soluciones factibles utilizando técnicas de resolución genéricas. Por contra, proporcionan ciertas ventajas y simplificaciones sobre el espacio de soluciones las cuales permiten tomar una gran ventaja a métodos de resolución específicos. Entre ellas, la idea que más destaca es la una reducción de orden logarítmico del tamaño del espacio de soluciones conforme las tareas van siendo selecionadas.

Para la formulación de *problemas generalizados de recogidas y entregas*, se utiliza una serie de restricciones comunes para poder modelar situaciones coherentes. Para ello en la ecuación 2.3 se incluye la formulación de un problema de rutas que será utilizado a modo de ejemplo. Dicha formulación se corresponde con la versión más simple del *problema del viajante*. Es importante remarcar que la modelización de este problema asume que únicamente existe un vehículo. Antes de proceder con la descripción de las restricciónes es necesario describir mínimamente la notación

que se utiliza en este caso: Suponemos que el problema está compuesto por N tareas, que el punto de inicio se indexa con la posición 0 (y que también deberá ser la posición final), que las variables de decisión x_{ij} representan la acción de que el vehículo se desplace desde la tarea i-ésima hasta la tarea j-ésima (por lo que X será una matriz de tamaño $(N+1)x(N+1)$) y los movimientos conllevas un coste determinado por c_{ij}. Además, se añade un vector auxiliar de variables de decisión U de longitud N el cual se explicará posterirmente. A continuación se describen dichas restricciones:

- **Unicidad**: Dado que este tipo de problemas se basan en la construcción de rutas cuyo objetivo principal es visitar una colección de tareas que representan una determinada posición en un hiperplano espacial, es necesario restringir el número de veces que estas se visitan a únicamente una. La forma de llevar a cabo esto es restringiendo el número de *enlaces* entre cualquier otra tarea y la tarea encuestión (así como la tarea en cuestión y cualquier otra tarea) a uno. Esto impone tanto la restricción citada, como que todas las tareas tengan que ser visitadas (existen técnicas para relajar dicha restricción en problemas no factibles totalmente mediante la redefinición como un *problemas multiobjetivo*). Entonces, en la formulación de la ecuación 2.4 las dos primeras restricciones generan la propiedad de unicidad.

- **Secuencialidad**: Además de construir rutas que permitan visitar todas las tareas del problema determinadas en el problema, también es necesario que dichas rutas cumplan las restricciones de secuencialidad entre un "enlace" y el siguiente. Típicamente este tipo de restricciones se conocen como de eliminación de subtours y es un tema ampliamente estudiado en la literatura entre las que destacan las de *Miller-Tucker-Zemlin* descritas en [MTZ60]. Sin embargo, también se han desarrollado soluciones específicas para problemas generalizados de recogidas y entregas tal y como se describe en [DL91]. En su versión más simple, estas restricciones se basan en el apoyo de un vector de variables de decisión (en nuestro caso U) que anotan el orden (a partir del índice o del momento temporal dependiendo de la formulación utilizada) de las tareas a partir de la cual se apoyan las restricciones para que los "enlaces" tengan una relación secuencial entre si. En el caso de la formulación descrita en la ecuación 2.4 esto se consigue a partir de la tercera restricción (que "asigna" y obliga a que exista la relación de orden en los índices anotados en el vector U).

Tal y como se puede apreciar a partir del ejemplo anterior, modelar *problemas generalizados de recogidas y entregas* es una tarea relativamente sencilla. Sin embargo, es importante remarcar que en la formulación de la ecuación 2.4 (*problema del viajante*) únicamente se requiere visitar todas las tareas y la solución óptima se consigue encontrando aquella que menor coste proporcione. Sin embargo, el marco de *problemas generalizados de recogidas y entregas* permite modelar situaciones mucho más complicadas. Esto se consigue ampliando las restricciones descritas inicialmente. Algunas de las más comunes se describen a continuación:

$$
\begin{aligned}
\text{Minimizar} \quad & \sum_{i=0}^{N} \sum_{j=1, j\neq i}^{N} c_{ij} x_{ij} \\
\text{sujeto a} \quad & \sum_{i=0, i\neq j}^{N} x_{ij} = 1, \qquad \forall i \in \{1, ..., N\} \\
& \sum_{j=0, j\neq i}^{N} x_{ij} = 1, \qquad \forall j \in \{1, ..., N\} \\
& u_i - u_j + N x_{ij} \leq N - 1, \quad 1 \leq i \neq j \leq N \\
& x_{ij} \in \{0, 1\}, \quad \forall i \in \{1, ..., N\}, \forall j \in \{1, ..., N\}
\end{aligned}
\tag{2.4}
$$

Ecuación 2.4: Formulación de un modelo *Generalizado de Recogidas y Entregas*. En concreto, el *Problema del viajante*.

- *Capacidad*: Muchos de los problemas generalizados de recogidas y entregas se basan en cargar y/o recoger una determinada mercancia para después llevar al almacén (u otro destino), por lo tanto es necesario tener en cuenta la capacidad del vehículo así como tener constancia de la ocupación del mismo en cada momento de la ruta. Dicha descripción se amplia para el caso de los *problemas de recogidas y entregas* en el apartado 2.4.

- *Ventanas Temporales*: Las tareas no siempre perduran a lo largo del tiempo, sino que tienen unas determinadas franjas temporales en que pueden ser servidas, por lo que dicha información debe ser incluida en la formulación del modelo para que este sea lo más próximo posible a la realidad. Dicha descripción se amplia para el caso de los *problemas de recogidas y entregas* en el apartado 2.4.

- *Duración*: Existen ocasiones en que es necesario imponer restricciones de duracción a lo largo de la ruta. Dichas restricciones pueden ser de diferentes tipos, desde el punto de vista de la duración total de la ruta, así como del tiempo que pasa desde el principio de la ruta hasta que se visita una determinada tarea, desde que se visita una determinada tarea hasta el final de la ruta, etc. Dicha descripción se amplia para el caso de los *problemas de recogidas y entregas* en el apartado 2.4.

Hasta ahora, se han descrito los *problemas generalizados de recogidas y entregas* en los que únicamente se considera un vehículo. Sin embargo, es relativamente sencillo ampliar la formulación a casos en los cuales el modelo permite la utilización de variaos vehículos. Nótese que esto no es equivalente a la construcción de un modelo para cada vehículo ya que el objetivo es que estos se repartan tareas entre si. Para ello existen distintas formulaciones, entre las que destacan la utilización de modelos de 3 índices (añadiendo una nueva dimensión a X que determina el vehículo) o de 2 índices (añadiendo nuevas restricciones que restringen el solapamiento de rutas). Cada una de estas alternativas presentan ventajas y desventajas que se describen más en detalle para el caso del *problema de recogidas y entregas* en el apartado 2.4.

2.2.4.1. Relación con problemas de Secuenciación de Tareas

Tras la descripción de los *problemas generalizados de recogidas y entregas* es fácil darse cuenta de la gran similitud que existe con los *problemas de secuenciación de tareas* ya que, al igual que estos, se basan en la optimización de un conjunto de tareas a servir. Antes de proceder con la relación y diferencias entre ambas categorías es necesario detallar que por *problemas de secuenciación de tareas* nos referimos a problemas *Job-Shop Scheduling*, en los cuales el objetivo es la planificación de una serie de tareas codependientes entre si de tal manera que estas se consigan resolver optimizando la duración total.

A pesar de que el enfoque conceptual es muy similar y tal (y como se expone en [BPS03] existen diferencias muy sutiles desde el punto de vista de la formulación), tanto el objetivo como el contexto de ambas categorías de problemas es muy diferente. En [BPS03] se exponen 5 razones por las cuales deben ser considerados como dos problemas diferentes, las cuales se proceden a enumerar a continuación:

- Recursos disponibles: Generalmente los *problemas generalizados de recogidas y entregas* planifican un número elevado de vehículos mientras que los *problemas de secuenciación* suelen poseer un número reducido de servidores.
- Dependencias temporales: Los *problemas generalizados de recogidas y entregas* presentan tareas independientes entre si (excepto en *problemas de rutas con recogidas y entregas* como se describe en próximos apartados) mientras que los *problemas de secuenciación* presentan una elevada componente de dependencias temporales entre las tareas.
- Estructura de tiempo: En los *problemas generalizados de recogidas y entregas* el mayor peso temporal se corresponde en el proceso de enlace entre una tarea y la siguiente, mientras que en el *problema de secuenciación* el peso temporal reside principalmente en la duracción de la propia tarea.
- Función Objetivo: En la mayoría de casos los *problemas generalizados de recogidas y entregas* tratan de reducir los costes para generar los trayectos óptimos, mientras que en los *problemas de optimización* el objetivo es minimizar la duración total del proceso (desde el inicio de la primera tarea hasta la finalización de la última).
- Ventanas Temporales: Por lo general, la mayoría de *problemas generalizados de recogidas y entregas* se caracterizan por estar compuestos por "amplias" ventanas temporales, mientras que los *problemas de secuenciación* suelen tener intervalos mucho más ajustados.

A pesar de que dichas diferencias se refieren más a la estructura de la instancia específica del problema a resolver que a la propia modelización del mismo, estas también afectan en cierta manera a la forma de definir la formulación. Dichas diferencias son apreciables sobre todo desde

el punto de vista de la definición de la función objetivo. Sin embargo, en el aspecto en que más se diferencian entre si es en la estrategia de resolución a seguir en cada caso desde el punto de vista de la utilización de heurísticas de búsqueda. Es fácil darse cuenta de que las estrategias que funcionen de manera adecuada con los *problemas generalizados de recogidas y entregas* probablemente no se comporten como deberían en *problemas de secuenciación*, y viceversa. Por todas estas razones, deben ser estudiados como problemas diferentes a pesar de tener muchas similitudes entre si.

Tal y como se ha descrito a lo largo del apartado, los problemas generalizados de recogidas y entregas representan un marco de modelización muy interesante que permite representar una gran cantidad de situaciones reales, desde el caso más sencillo relacionado con el *problema del viajante* hasta modelizaciones muy complejas como las de empresas de reparto a gran escala que requieren de la construcción de rutas de reparto que aprovechen al máximo los recursos de la misma. El ejemplo más notable en este campo es el sistema de distribución de productos de empresas como *Amazon* dado que sin técnicas de optimización sería imposible alcanzar un nivel de servicio similar sin caer en costes de servicio inmanejables.

2.2.5. Problemas de Rutas con Recogidas y Entregas

Hasta ahora se ha definido la categoría de *problemas generalizados de recogidas y entregas* de manera general, indicando que estos son aquellos problemas basados en la planificación de trayectos que visitan una serie de puntos definidos como tareas. Sin embargo, este es un marco muy general que a su vez permite modelar una gran cantidad de problemas. Este apartado se centra en el caso de *problemas de rutas con recogidas y entregas*. Dicha modelización se caracteriza porque las tareas no solo se basan en la entrega (o recogida) de una determinada mercancia desde (o hasta) el almacén de origen (o destino), si no que el concepto de tarea queda divido en dos. Una de ellas representa la recogida de mercancia en un determinado punto, mientras que otra representa la entrega de mercancia en otro punto diferente. Nótese que este modelo se contrapone al enfoque de los *problemas de rutas* donde las mercancias son recogidas (o entregas) hasta (o desde) un almacén común.

Entonces, la característica de recogida y entrega en posiciones arbitrarias del hiperplano espacial sobre el cual se ha definido el problema ofrece una gran potencia desde el punto de vista de los problemas de modelización que permite representar. Sin embargo, dicha potencia conlleva una mayor dificultad desde el punto de vista de los métodos de resolución donde el espacio de soluciones es inabordable. Sin embargo, tal y como se comenta al principio del apartado 2.2.4 la estructura de codependencia entre variables de decisión permite tomar ventaja reduciendo notablemente el espacio de búsqueda.

Dentro de esta clase de problemas, es necesario hacer una importante distinción entre las dos grandes alternativas en que se pueden producir las recogidas y entregas. Estas pueden ser o no

pareadas por lo tanto, a continuación se describen las diferencias más notables entre ambas clases:

1. **Pareadas**: Cuando las tareas son pareadas es necesario que exista una recogida para que haya una entrega y ambas deben cumplir la misma capacidad. Además, estas no pueden ser servidas en ordenes opuestos (primero se debe producir la recogida y después la entrega).

2. **No Pareadas**: En el caso de problemas con tareas no pareados las limitaciones anteriormente descritas son eliminadas. Es decir, no es necesario que exista una recogida para su correspondiente entrega (y viceversa). Es de especial importancia darse cuenta de que para que esto sea posible se debe asumir que la mercancia es uniforme entre si (al menos en la formulación más básica) de tal manera que pueda ser intercambiable entre todas las tareas.

2.2.6. Problemas Dial-a-Ride

El *problema Dial-a-Ride* se engloba dentro del marco de los *problemas de rutas con recogidas y entregas* donde las tareas son pareadas entre si. Es decir, donde existe una tarea que representa la recogida con una capacidad igual a la de su correspondiente tarea de entrega, y además estas han de ser realizadas siguiendo un orden estricto. A este tipo de problemas se les conoce como *problemas de recogidas y entregas* o *Pickup and Delivery Problem* (*PDP*). Es en esta categoría de problemas donde surge el concepto de *viaje*, que se define como el trayecto dentro de una ruta que realiza cada una de las mercancias desde que estas se recogen hasta que se entregan. Nótese entonces que cada viaje esta formado por dos tareas.

A pesar de que el *problema Dial-a-Ride* pertenece a la categoría de *problemas de recogidas y entregas*, este se caracteriza porque la mercancia a transportar está formada por personas (siendo el ejemplo más significativo el problema de planificación de una compañía de taxis), lo cual implica la creación de restricciones adicionales. Típicamente se añaden restricciones que limitan la duración máxima de cada uno de los viajes a un tiempo máximo, algo que no es tan importante en el caso de mercancias pero si en el de personas.

[TODO: Hablar sobre las descripciones definidas en [CL07]]

2.2.6.1. Aplicaciones en Situaciones Reales

[TODO]

[TODO]

2.3. Notación

Una vez contextualizado de una manera detallada el *problema Dial-a-Ride* es fácil intuir cuáles son los factores que más complican la implementación y el uso de métodos que permitan resolver instancias reales que tan frecuentemente se producen en situaciones del mundo real. En este caso, el *problema Dial-a-Ride* es relativamente sencillo desde el punto de vista conceptual (tal y como se ha indicado anteriormente, se trata de cumplir un conjunto de tareas de transporte desde un punto de recogida hasta un punto de entrega a partir del uso de un conjunto de vehículos). Sin embargo, a pesar de su sencillez conceptual, en muchas ocasiones es complicado representar el problema en lenguaje matemático. Por dicha razón, es de vital importancia definir una nomenclatura común a lo largo de todo el trabajo que permita facilitar la representación matemática de las ideas que se pretende transmitir. Para ello, se incluye este apartado, el cual (como se ha comentado previamente) pretende servir como una guía de referencia que clarifique el sentido de ciertas definiciones futuras.

A pesar de lo expuesto en el párrafo anterior, la tarea de definir una nomenclatura común y aplicable a las posteriores explicaciones no es una tarea sencilla. La dificultad surge principalmente porque los términos no siempre se utilizan de la misma manera y en ocasiones son muy dependientes del contexto. Sin embargo, esto se tratará de hacer en la medida de lo posible. Una vez introducido el apartado, en el apartado 2.3.1 se procede a enumerar todos los términos necesarios para comprender adecuadamente el *problema Dial-a-Ride* para posteriormente definir las variables que se utilizarán en los procesos de formulación, así como en los posteriores métodos de resolución en subsiguientes capítulos. Seguidamente, en el apartado 2.3.2 se definen las Constantes que se utilizan en las formulaciones matemáticas y, finalmente, en el apartado 2.3.3 se describen las variables de decisión necesarias para llevar a cabo la formulación básica del problema.

2.3.1. Palabras clave

A pesar de que para la formulación del problema no sea estrictamente necesario detallar las palabras clave que se utilizan para tratar el problema (y sus correspondientes métodos de resolución), se cree que estos pueden faciliar el entendimiento posterior de las variables utilizadas. Además, estos conceptos serán de vital importancia para definir los métodos de resolución basados en heurísticas y metaheurísticas en capítulos posteriores.

- **Posición**: Vector de coordenadas que representa el lugar en que se ha de realizar una determinada acción (recoger o entregar) para resolver la instancia del problema.
- **Superficie**: El subespacio formado por el conjunto de todas las posiciones que forman la instancia del problema. Este es quien define la métrica de distancia (tanto espacial como temporal) entre las distintas posiciones del problemas.
- **Servicio**: Está compuesto por una posición, un tiempo para ser efectuado, y una ventana temporal.

- **Viaje**: Está compuesto por los servicios de recogida y entrega de una mercancia. Además contiene información necesaria para poder proceder a su realización, es decir, la capacidad (o el tamaño de la carga a desplazar), el tiempo máximo admisible desde la recogida hasta la entrega el tiempo de servicio en las posiciones de recogida y entrega, etc. En la literatura es común diferenciar entre dos tipos de viajes a partir de las ventanas temporales. Cuando estas se fijan sobre la posición de recogida se denominan *outbound* mientras que cuando se hacen sobre la posición de entrega son denominadas *inbound*. Sin embargo, esto es simplemente terminología del problema ya que desde un punto de vista de la formulación del modelo ambos conceptos podrían ser aplicados a la vez sobre un viaje si se fijan de manera pertinente las ventanas temporales.

- **Trabajo**: Representa el conjunto de todos los viajes a realizar durante una determinada instancia del problema.

- **Vehículo**: Representa el actor encargado de llevar a cabo un subconjunto de viajes que permiten resolver una instancia del problema de manera parcial (o completa). Se define por una posicion inicial, así como otra final, así como por una determinada capacidad máxima y un tiempo máximo de servicio. Además, tiene asociada una determinada ventana temporal.

- **Flota**: Está compuesta por el conjunto de todos los vehículos disponibles para resolver la instancia del problema. En ciertas ocasiones el tamaño de la flota de vehículos es fija, mientras que en otras ocasiones se permite que la instancia del problema sea planificada con tantos vehículos como sea posible. Dichas ideas se discutirán posteriormente.

- **Parada**: Representa la acción de visitar una determinada posición por un vehículo. Esta está compuesta por el instante de tiempo en que se visita así como la razón, la cual puede ser una recogida o entrega referida a un determinado viaje.

- **Viaje Planificado**: Representa el enlace entre un viaje y sus dos correspondientes paradas (la de recogida y la de entrega), que (como se ha dicho previamente) mantienen los instantes de tiempo en que sucede.

- **Ruta**: Representa una serie de viajes planificados por un determinado vehículo.

- **Planificación**: Está formada por el conjunto de rutas que permiten resolver la instancia del problema.

- **Resultado**: Está compuesta por la planificación que resuelve parcial (o totalmente) la instancia del problema, junto con la correspondiente flota de vehículos y el trabajo formado por un conjunto de viajes a realizar.

Una vez, descritas las palabras clave más importantes es fácil darse cuenta de que los conceptos más importantes para la resolución del *problema Dial-a-Ride* son el de *viaje* y el de *ruta*, ya que son quienes contienen la información tanto del qué resolver (*viae*), como del cómo resolver (*ruta*).

2.3.2. Constantes

El siguiente paso antes de proceder a la descripción del *problema Dial-a-Ride* de manera matemática a través de su formulación es definir el significado de las constantes que se utilizan en el mismo. A continuación se incluye el listado junto con su correspondiente definicion.

- P: Conjunto de nodos generado por las posiciones referidas a recogidas. Esto es el conjunto formado por la secuencia $\{1, 2, ..., n\}$.
- D: Conjunto de nodos generado por las posiciones referidas a entregas. Esto es el conjunto formado por la secuencia $\{n+1, n+2, ..., 2n\}$.
- N: Conjunto de nodos generado por todas las posiciones de la superficie. Esto es el conjunto formado por $\{0, 2n+1\} \cup P \cup D$.
- $A_{(i,j)}$: El arista que se encarga de relacionar el nodo definidos por la posición i-ésima con el nodo definido por la posición j-ésima. Nótese por tanto que inicialmente la cardinalidad es de $card(N)^2$. Sin embargo, asumiendo que no se pueda dar la situación de visitar un nodo desde si mismo (para respetar las restricciones de unicidad) este número se reduce a $card(N)(card(N)-1)$. Sin embargo, es fácil darse cuenta de que muchos de los aristas serán marcados como "prohibidos" por las restricciones del modelo (No se puede visitar una posición de entrega antes que su correspondiente posición de recogida).
- $G = (N, A)$: El grafo generado por el conjunto de nodos sobre el conjunto de aristas, sobre el cual se apoya toda la planificación.
- K_k: Conjunto de índices generados por los vehículos disponibles para resolver el problema.
- c_{ij}: Coste por moverse del nodo i-ésimo al nodo j-ésimo.
- t_{ij}: Tiempo para moverse del nodo i-ésimo al nodo j-ésimo.
- e_i: Cota inferior de la ventana temporal sobre el nodo i-ésimo.
- l_i: Cota superior de la ventana temporal sobre el nodo i-ésimo.
- d_i: Tiempo de servicio sobre el nodo i-ésimo.
- q_i: Capacidad del servicio sobre el nodo i-ésimo. Nótese que para los nodos 0 y $2n+1$ la capacidad a servir deberá ser 0 mientras que se deberá cumplir que $q_i \geq 0$ si $i \in P$ y $q_i = q_{i-n}$ si $i \in D$.

- T_k: Duración máxima de ruta por el vehículo k-ésimo.
- Q_k: Capacidad máxima del vehículo k-ésimo.
- E_k: Cota inferior de la ventana temporal sobre el vehículo k-ésimo.
- L_k: Cota superior de la ventana temporal sobre el vehículo k-ésimo.

Nótese la equivalencia conceptual entre las definiciones descritas en este apartado con respecto al apartado anterior (de *Palabras clave*). Se ha decidido describir los dos enfoques para permitir una mayor versatilidad (y comprension) a lo largo del documento.

2.3.3. Variables de decisión

Por último, se procede a definir el significado de las variables que se utilizan en el mismo. A continuación se incluye el listado junto con su correspondiente definicion. Antes de nada es importante remarcar que existen dos formulaciones populares: basadas en 3 índices (la cual se seguirá en la fomulación básica) y basadas en 2 índices (la cual se expone en futuros apartados). En este caso se describe la de 3 índices por ser más general. A pesar de ello es trivial entender la simplificación cuando se trate de la de 2 índices.

- x_{ij}^k: Variable de naturaleza binaria que toma el valor 1 cuando existe un enlace entre los nodos i-ésimo y j-ésimo llevado a cabo por el vehículo k-ésimo
- u_i^k: Variable de decisión continua que representa el instante de tiempo en que comienza la ruta generada por el vehículo k-ésimo llega al nodo i-ésimo.
- w_i^k: Variable de decisión continua que representa la capacidad ocupada en la ruta generada por el vehículo k-ésimo en el nodo i-ésimo.

A pesar de que dichas variables de decisión serán las utilizadas principalmente durante el proceso de formulación de los problemas, puede darse el caso de que en algunas situaciones concretas (para tratar de corregir la linealidad de restricciones, simplificarlas o algún paso similar, se utilicen variables diferentes).

Para concluir el apartado se quiere remarcar lo siguiente: Tal y como se ha repetido en varias ocasiones a lo largo del mismo, es de vital importancia definir de manera adecuada la notación utilizada para así simplificar en la medida de lo posible la aclaración de los conceptos que se construyen posteriormente a partir de la misma. En este caso se pretende seguir una notación común que después sea aplicable a la definición de los distintos métodos de resolución. Por lo tanto se ha preferido llevar a cabo de la manera más general posible.

2.4. Formulación básica

Tras contextualizar de una manera detallada el *problema Dial-a-Ride*, tanto desde el punto de vista de la situación que pretende resolver, partiendo desde una base muy general (*Problemas de Optimización* en el apartado 2.2.1) hasta las características concretas del mismo (*Problemas Dial-a-Ride* en el apartado 2.2.6), así como la descripción de la notación que se utiliza a lo largo del documento, tanto desde una perspectiva de bocabulario como matemática, ya se está en condiciones suficientes como para presentar la formulación específica para dicho problema. En este caso, se describe la formulación básica (u original) formada por 3 índices en el tensor tridimensional que contine las variables de decición binarias x_{ij}^k. En la literatura existen distintas representaciones de la misma, entre las que se encuentran [Cor06; RCL07; CL07] entre otros. Sin embargo, en este caso se ha preferido mostrar una versión más cercana a la formulación como modelo lineal, simplificando previamente las restricciones no lineales, las variables redundates etc.

$$
\begin{aligned}
\text{Minimizar} \quad & \sum_{k\in K}\sum_{j\in V}\sum_{i\in V} c_{ij}^k x_{ij}^k & & \\
\text{sujeto a} \quad & \sum_{k\in K}\sum_{j\in V} x_{ij}^k = 1, & & \forall i \in P \\
& \sum_{i\in V} x_{0i}^k = 1, & & \forall k \in K \\
& \sum_{i\in V} x_{i,2n+1}^k = 1, & & \forall k \in K \\
& \sum_{j\in V} x_{ij}^k - \sum_{j\in V} x_{n+i,j}^k = 0, & & \forall i \in P, \forall k \in K \\
& \sum_{j\in V} x_{ji}^k - \sum_{j\in V} x_{ij}^k = 0, & & \forall i \in P \cup D, \forall k \in K \\
& u_i^k + d_i + t_{ij} - M_{ij}^k(1 - x_{ij}^k) \leq u_j^k, & & \forall i,j \in V, \forall k \in K \\
& w_i^k + q_j - W_{ij}^k(1 - x_{ij}^k) \leq w_j^k, & & \forall i,j \in V, \forall k \in K \\
& u_{2n+1}^k - u_0^k \leq T_k, & & \forall k \in K \\
& e_i \leq u_i^k, & & \forall i \in V, \forall k \in K \\
& u_i^k \leq l_i, & & \forall i \in V, \forall k \in K \\
& t_{i,n+i}^k \leq u_{n+i}^k - (u_i^k + d_i), & & \forall i \in P, \forall k \in K \\
& u_{n+i}^k - (u_i^k + d_i) \leq L_k, & & \forall i \in P, \forall k \in K \\
& max\{0, q_i\} \leq w_i^k, & & \forall i \in V, \forall k \in K \\
& w_i^k \leq Q_k + min\{0, q_i\}, & & \forall i \in V, \forall k \in K \\
& x_{ij}^k \in \{0,1\}, & & \forall i,j \in V, \forall k \in K
\end{aligned}
\tag{2.5}
$$

Ecuación 2.5: Formulación del modelo basado en 3 índices para el *Problema Dial-a-Ride*.

En la ecuación 2.5 se muestran tanto las restricciones como la función objetivo del problema. En este último caso, es sencillo darse cuenta de que el valor a optimizar en el modelo básico es el coste total de la planificación, que viene dado por la acumulación de costes para cada enlace entre posiciones generadas por los servicios del problema. Esto se puede ver de forma sencilla en la ecuación (2.6).

$$\sum_{k \in K} \sum_{j \in V} \sum_{i \in V} c_{ij}^k x_{ij}^k \tag{2.6}$$

En cuanto a las restricciones, estas pueden ser agrupadas en tres grupos principales: restricciones de unicidad (apartado 2.4.1), de conectividad (apartado 2.4.2), de acumulación (apartado 2.4.3), y de tiempo y capacidad (apartado 2.4.4). A continuación se describen un poco más en detalle los distintos grupos de restricciones.

2.4.1. Restricciones de Unicidad

Tal y como se ha indicado previamente en el apartado 2.2.4, las restricciones de unicidad en los problemas de rutado son las encargadas de asegurarse de que cada servicio sea realizada una única vez, así como que sea realizada obligatoriamente. En este caso, se pretende además de que distintos vehículos no realicen el mismo servicio. Dicha restricción se muestra en la ecuación (2.7). Es interesante darse cuenta de que esta restricción tan solo se aplica a los servicios correspondientes a recogidas. La razón de ello, es que por transitividad procedente de otro conjunto de restricciones seria redundante.

Dicho conjunto addional de restricciones se refiere a la necesidad de que si una ruta visita la posición correspondiente a un servicio de recogida, entonces será necesario que también visite la posición del correspondiente servicio de entrega. Esto es necesario en el *problema Dial-a-Ride* ya que tal y como se comentó previamente, en este caso las mercancias no son intercambiables entre distintos pares de recogida/entrega. La restricción correspondiente se describe en la ecuación (2.8).

$$\sum_{k \in K} \sum_{j \in V} x_{ij}^k = 1, \qquad \forall i \in P \tag{2.7}$$

$$\sum_{j \in V} x_{ij}^k - \sum_{j \in V} x_{n+i,j}^k = 0, \qquad \forall i \in P, \forall k \in K \tag{2.8}$$

Una vez añadidas dichas restricciones, ya se puede asegurar que para llegar a una solución factible del problema, todas las posiciones en que se localizan sus correspondientes servicios hayan sido visitadas, así como que las posiciones de inicio y fin de ruta se cumplan.

2.4.2. Restricciones de Secuenciación

Las restricciones de secuenciación (las cuales se comentaron brevemente en el apartado 2.2.4) se refieren a la necesidad de que tras cada parada en un servicio, únicamente se pueda continuar hacia otro servicio. Es decir, que no surjan problemas relacionados con la secuencialidad de la ruta. Esto se lleva a cabo a partir de la restricción descrita en la ecuación (2.9). Nótese que esto tan solo se aplica a las posiciones relacionadas con servicios, ya que las posiciones referidas al inicio y fin de ruta, el comportamiento ha de ser diferente.

Dicho comportamiento requiere que las posiciones de inicio y fin de ruta sean visitadas por todas las rutas (por lo que no se aplica ni la unicidad ni la secuencialidad). Sin embargo, es necesario ser capaz de controlar las posiciones de origen y destino de la planificación (que típicamente coinciden aunque podría no ser así). Asumiendo que todas las rutas comienzan y terminan en las posiciones 0 y $2n+1$ respectivamente, es necesario añadir las restricciones descritas en las ecuaciones (2.10) y (2.11).

$$\sum_{j \in V} x_{ji}^k - \sum_{j \in V} x_{ij}^k = 0, \qquad \forall i \in P \cup D, \forall k \in K \tag{2.9}$$

$$\sum_{i \in V} x_{0i}^k = 1, \qquad \forall k \in K \tag{2.10}$$

$$\sum_{i \in V} x_{i,2n+1}^k = 1, \qquad \forall k \in K \tag{2.11}$$

Una vez incluidas las restricciones de secuencialidad, ya es posible conseguir rutas válidas desde el punto de vista de las posiciones que se visitan y el orden entre estas. Sin embargo, estas no tienen en cuenta ni tiempos ni capacidades por lo que es necesario añadir un conjunto de restricciones addicionales para poder modelar de manera completa el *problema Dial-a-Ride*.

2.4.3. Restricciones de Acumulación

En problemas basados en el mantenimiento de un cierto nivel de estado para la planificación, donde este se basa en incrementos y decrementos sobre determinadas variables, es necesario incluir distintas restricciones que "asignen" a las variables relacionadas sus correspondientes valores. En el caso del *problema Dial-a-Ride* es necesario mantener la información tanto del instante de tiempo como de la capacidad con que se llega a la posición de un determinado servicio. Dichas restricciones pueden ser expresadas de distintas formas, siendo algunas de ellas muy intuitivas pero no lineales. En este caso, se incluye la versión lineal tanto para tiempos como para capacidades en las ecuaciones (2.12) y (2.13) respectivamente.

$$u_i^k + d_i + t_{ij} - M_{ij}^k(1 - x_{ij}^k) \leq u_j^k, \qquad \forall i, j \in V, \forall k \in K \tag{2.12}$$
$$w_i^k + q_j - W_{ij}^k(1 - x_{ij}^k) \leq w_j^k, \qquad \forall i, j \in V, \forall k \in K \tag{2.13}$$

Las restricciones inicialmente no lineales pueden ser vistas como implicaciones en el sentido siguiente: "Si se utiliza el enlace entre las posiciones i-ésima y j-ésima, entonces el incremento será de y". Para linearizar este tipo de restricciones sobre variables de decisión de carácter binario, se puede utilizar una cota superior. En [Cor06] se indica que los valores descritos en las ecuaciones (2.14) y (2.15) son suficientes para tiempos y capacidad respectivamente.

$$M_{ij}^k = max\{0, l_i + d_i + t_{ij} - e_j\} \tag{2.14}$$
$$W_{ij}^k = Q_k + min\{0, q_i\} \tag{2.15}$$

Las restricciones definidas en este apartado permiten mantener el estado a nivel de tiempo y capacidad para cada ruta en cada parada de la misma. Sin embargo, estas no aportan ninguna restricción sobre la lógica del problema. A pesar de ello, es necesario incluirlas en la formulación para poder apoyarnos en ellas durante la definición de las restricciones propiamente relacionadas con la lógica del problema.

2.4.4. Restricciones de Tiempo y Capacidad

Una vez descritas todas aquellas restricciones que permiten mantener de manera coherente tanto la unicidad y secuenciación de paradas, como el estado a lo largo de las rutas, se pueden añadir las restricciones más específicas y que utilizan al resto como base.

$$u_{2n+1}^k - u_0^k \leq T_k, \qquad \forall k \in K \tag{2.16}$$
$$e_i \leq u_i^k, \qquad \forall i \in V, \forall k \in K \tag{2.17}$$
$$u_i^k \leq l_i, \qquad \forall i \in V, \forall k \in K \tag{2.18}$$
$$t_{i,n+i}^k \leq u_{n+i}^k - (u_i^k + d_i), \qquad \forall i \in P, \forall k \in K \tag{2.19}$$
$$u_{n+i}^k - (u_i^k + d_i) \leq L_k, \qquad \forall i \in P, \forall k \in K \tag{2.20}$$
$$max\{0, q_i\} \leq w_i^k, \qquad \forall i \in V, \forall k \in K \tag{2.21}$$
$$w_i^k \leq Q_k + min\{0, q_i\}, \qquad \forall i \in V, \forall k \in K \tag{2.22}$$

En primer lugar, es necesario fijar un tiempo máximo de ruta, lo cual se puede llevar a cabo apoyandose en las variables de estado de tiempo, mediante la restricción descrita en la ecuación (2.16). De manera similar, para respectar las ventanas temporales de llegada a cada posición para poder

cumplir con un servicio es necesario utilizar las restricciones definidas en las ecuaciones (2.17) y (2.18) que controlan los tiempo más tempranos y tardíos de llegada respectivamente.

La restricción que diferencia el *problema de recogidas y entregas* del *problema Dial-a-Ride* consiste en la fijación de un tiempo máximo de trayecto por cada viaje. Esto es el tiempo que pasa desde el servicio de recogida hasta el de entrega para un viaje concreto. Las restricciones corresponientes se incluye en las ecuaciones (2.19) y (2.20).

Finalmente, es necesario restringir de manera apropiada la capacidad máxima que el vehículo puede tranportar durante cada etapa de la ruta. Para ello, se han de actualizar de manera coherente tanto las cargas como las descargas, lo cual queda resuelto con las restricciones de acumulación descritas anteriormente. Por lo tanto, en este caso, tan solo es necesario asegurar que la capacidad resultante tras cada servicio siempre se encuentre dentro del intervalo válido ($[0, Q_k]$). Esto se puede llevar a cabo de manera sencilla mediante el uso de las restricciones definidas en las ecuaciones (2.21) y (2.22).

[TODO]

2.5. Conclusiones

[TODO]

Capítulo 3

Métodos de Resolución

3.1. Introducción

Los problemas de *optimización combinatoria* constituyen una de las categorías más interesantes dentro del área de la optimización de variables. Esto se debe a la gran aplicabilidad de los mismos en situaciones de la vida real, así como la gran reducción de costes que se puede alcanzar cuando estos son aplicados en los puntos estratégicos de cualquier proceso de producción.

Sin embargo, esta categoría de problemas de optimización presenta un mayor grado de dificultad en su resolución, tal y como se ha indicado a lo largo del capítulo 2. Entre otros, esto se debe a la imposibilidad de utilizar técnicas basadas en gradientes (que en problemas con variables de decisión continuas proporcionan simplifican mucho su resolución). Puesto que estas técnicas no son aplicables, la estrategia alternativa se basa en la enumeración de posibles configuraciones de variables de manera inteligente hasta alcanzar aquella que genere el resultado óptimo para la instancia del problema que se pretenda resolver.

Por tanto, a pesar de no ser posible el apoyo en técnicas basadas en gradientes, si que existe la aternativa basada en enumeración de configuraciones como enfoque para dar con el óptimo. A pesar de ello, es fácil darse cuenta de que esta estrategia no es lo suficientemente potente como para permitir resolver problemas reales (o incluso de pequeño tamaño). La dificultad radica en la explosión combinatoria generada por la enumeración de soluciones. Por ejemplo, en un problema compuesto por 20 variables de decisión de naturaleza binaria, el número de configuraciones a evaluar alcanzaría el valor de $2^{20} = 1048576$, lo cual no es escalable a problemas de gran tamaño, donde hay cientos o miles de variables de decisión con las que trabajar.

A pesar de esta dificultad, es posible enfrentarse a problemas de *optimización combinatoria* de tamaños relativamente grande teniendo en cuenta distintas estrategias que permiten ignorar un gran número de configuraciones no factibles dadas las restricciones descritas por el problema en cuestión. Es por ello que muchos de los métodos de resolución se apoyan en estas garantias para

enfocar la búsqueda sobre un subespacio de configuraciones factibles. Otra de las ideas más utilizadas por los métodos de resolución es apoyarse en los resultados de evaluación de configuraciones anteriores para que el coste computacional de evaluación de configuraciones actuales sea mucho menor. Dicha estrategia algorítmica se conoce como *Programación Dinámica*, la cual se describe en mayor detalle en [Bel54].

A lo largo del capítulo se decriben distintas situaciones en que es posible aplicar las técnicas anteriormente descritas sobre el *problema Dial-a-Ride* para tratar de reducir en la medida de los posible la complejidad computacional que este presenta. Entonces, el resto del capítulo se organiza de la siguiente manera: En el apartado 3.2 se describen las distintas técnicas de resolución exactas que la literatura especializada en el problema a probado para tratar de resolver el problema. Seguidamente, en el apartado 3.3 se describen estrategia basadas en heurísticas (tanto de construcción como mejora de soluciones) que a pesar de no ofrecer ninguna garantía de optimalidad, ofrecen un buen equilibrio entre tiempo de cómputo y calidad de las soluciones. Después, en el apartado 3.4 se describen un conjunto de metaheurísticas basadas en la utilización de las heurísticas descritas en el apartado anterior de manera inteligente de tal manera que las configuraciones generadas proporcionan soluciones de mayor calidad. Por último, en el apartado 3.5 se incluyen unas conclusiones generales acerca de los métodos de resolución aplicados al *problema Dial-a-Ride*.

3.2. Métodos de Resolución Exactos

Los métodos de resolución exactos representan una de las categorías de resolución de problemas de optimización más interesantes debido a distintos factores, entre los que se encuentran aquellas situaciones en que es estrictamente necesario obtener el valor óptimo para una instancia dada. Sin embargo, esta categoría no solo es interesante por dicha razón, sino que el estudio de estos problemas además proporciona en muchas ocasiones razonamientos y justificaciones teóricas sobre el problema que pretenden resolver que ayudan a comprender mejor los factores del mismo. Lo interesante de este punto es que esto sirve de utilidad para el diseño de diferentes métodos de resolución así como su justificación.

Tal y como se ha descrito anteriormente, el método de resolución exacta más básico consiste en la enumeración de todas las configuraciones posibles dadas la variables de decisión del problema para después determinar como configuración óptima aquella que maximiza/minimiza la función objetivo. Esto es teóricamente posible por la naturaleza combinatoria del problema pero a la vez prácticamente imposible por la misma razón.

Puesto que los problemas de *optimización combinatoria* se pueden modelar como problemas de *optimización lineal* mediante un conjunto de restricciones de naturaleza lineal, la siguiente alternativa a la enumeración de configuraciones es la resolución como un problema de esta naturaleza.

Para ello, una de las opciones es la utilización del algoritmos *Simplex* [KM70] tal y como se indica en el apartado 2.2.2. Como se ha descrito anteriormente, este algoritmo se basa en la búsqueda de la mejor solución contigua a la actual de manera iterativa hasta alcanzar aquella que no tenga una solución contigua mejor. Es posible demostrar que este algoritmo alcanza el valor óptimo cuando se aplica sobre problema de *optimización lineal*. Tal y como se puede apreciar en la figura 3.1, gráficamente puede ser visto como la evaluación de vértices en un grafo de soluciones. La complejidad que esta estrategia presenta en problemas de *optimización combinatoria* reside en el gran número de vértices que surgen del cambio de valor de cada una de las variables de decisión del problema, generando un grafo con gran número de vértices y aristas lo cual hace inalcanzable computacionalmente navegar por el mismo hasta alcanzar la solución óptima.

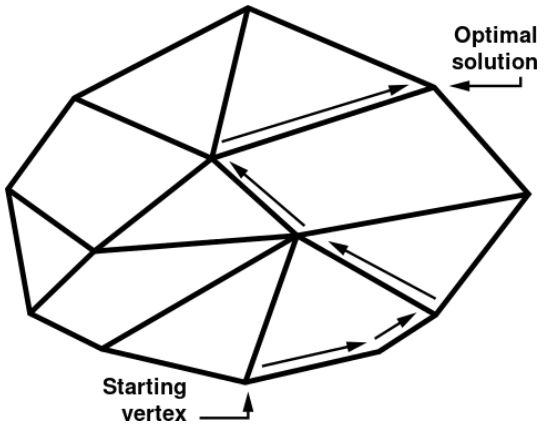


Figura 3.1: Representación gráfica del modo de ejecución del algoritmo *Simplex*.

Hasta ahora, en este apartado se ha descrito únicamente el proceso de resolución mediante algunos métodos exactos. Sin embargo, no se ha realizado ninguna particularización sobre el problema *Dial-a-Ride*. Por tanto, es momento de definir el punto de enlace entre la definición del método y el problema en cuestión. Tal y como se ha descrito a lo largo del capítulo 2 el problema *Dial-a-Ride* se enmarca dentro de los problemas de *optimización lineal*. En la ecuación 2.5 se describe la formulación de 3 índices para este problema, incluyendo tanto las variables de decisión como las restricciones que lo componen. En cuanto a la estrategia de enumeración, es fácil darse cuenta de que esta puede llevarse a cabo tomando todas las combinaciones posibles para las variables binarias del problema (lo cual tal y como se ha comentado anteriormente es algo inabordable en problemas no demostrativos). En cuanto a la estrategia de resolución basada en el algoritmo *Simplex*, tal y como se ilustra en [KM70] es posible construir la matriz simplex a partir de la matriz de restricciones generada por el problema en forma lineal.

A lo largo de la literatura relacionada con métodos de resolución exactos para el problema *Dial-a-Ride*, existen un gran número de referencias basadas en el estudio de técnicas basadas en *Branch and Bound* que tratan de tomar ventaja de alguna manera de la estructura del problema para así simplificar el camino hasta la solución óptima. En el siguiente apartado se describen de manera más detallada los conceptos en que se apoya este método.

3.2.1. Branch and Bound

Se pueden englobar en la categoría de algoritmos *Branch and Bound* todos aquellos que basan su estrategia de resolución en comenzar por una versión relajada del problema base para posteriormente realizar un proceso jerarquico de cambios sobre la estructura del problema hasta alcanzar el valor óptimo del mismo. Esta estrategia es muy usada sobre problemas de *optimización entera* ya que facilita en gran medida la problemática generada por las variables de decisión de carácter discreto del problema.

La estrategia habitual consiste en partir de una versión del problema sobre la cual las restricciones sobre variables discretas se han relajado a valores sobre intervalos. Es decir, supongamos que la variable de decisión x únicamente puede tomar valores enteros entre 1 y 5, lo que es equivalente a $x \in \{1, 2, 3, 4, 5\}$. Entonces, una relajación lineal podría venir dada por la transformación en $1 \leq x \leq 5$ la cual es más sencilla de resolver por el algoritmo *Simplex*. Posteriormente, el resto de pasos de ramificación consisten en añadir restriciones (mediante la generación de filas o columnas) que permitan satisfacer las restricciones del problema original.

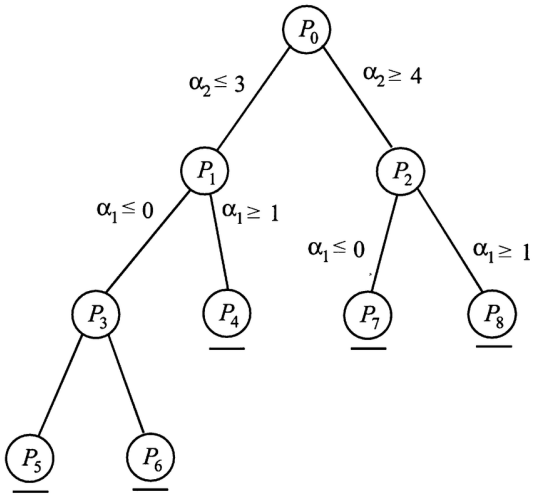


Figura 3.2: Ejemplo de ramificación del problema mediante el algoritmo *Branch and Bound*.

Es importante tener en cuenta que las relajaciones aplicadas al problema inicial para aplicar el algoritmo *Branch and Bound* no tienen por qué ser de naturaleza lineal, si no que se pueden aplicar sobre otros tipos de restricciones. Un ejemplo de ello se podría dar eliminando por ejemplo las restricciones de duración máximo de viaje en el problema *Dial-a-Ride*, o la eliminación de restrcciones de ventana temporal en el problema *Pickup and Delivery with Time Windows*. Una vez generada una solución factible sobre estos casos, el proceso de ramificación consistirá en "modificar" dicha solución hasta alcanzar aquella que cumpla todas las restricciones (de tiempo máximo de viaje, de ventanas temporales, etc.).

Dependiendo de la estrategia de ramificación del algoritmo, este se denomina de distintas maneras. Cuando la ramificación se basa en la generación de filas el algorimo se conoce como *Branch and Cut*, mientras que si la estrategia se basa en la generación de columnas este se denomina *Branch and Price*. Por último, cuando la estrategia se basa en la combinación de ambos entonces el algorimo es denominado *Branch and Cut and Price*.

3.2.1.1. Branch and Cut

Tal y como se ha comentado anteriormente, los algoritmos *Branch and Cut* consisten en una caracterización del algoritmo general de ramificación donde el proceso de "estrechamiento" de la versión relajada para cumplir con todas las restricciones impuestas por el problema viene dado por la adicción de planos de corte a partir de nuevas restricciones. Es por ello que este método se apoya en estrategias de generación de filas.

Para el caso del problema *Dial-a-Ride*, en [Cor06] se propone una implementación completa basada en el algoritmo *Branch and Cut* la cual se apoya en la generación inicial de una solución del problema relajado (para el cual se eliminan las restricciones que imponen variables de decisión binarias, permitiendo que estas tomen cualquier valor en el intervalo $[0, 1]$) para posteriormente, a partir de un proceso de ramificación añadir restricciones adiccionales que fuerzan a las variables de naturaleza binaria con valores no binarios a tomar un valor válido (ahí surge el proceso de ramificación). Para detectar sobre qué variable concreta generar el corte en cada caso, la estrategia seguida se apoya en la definición de unas heurísticas que generan una lista ordenada sobre la cual seleccionar dicha variable. En la línea de este enfoque, también es interesante remarcar [RCL07] donde además se propone un modelo de programación lineal basado en 2 índices, que a pesar de no representar el mismo caso que el problema *Dial-a-Ride*, sus diferencias son mínimas (número indeterminado de vehículos, todos los vehículos con misma capacidad, etc.).

3.2.1.2. Branch and Price

Del mismo modo que el método *Branch and Cut* es una estrategia de generación de filas enmarcada dentro de la familia de algoritmos *Branch and Bound*, el método *Branch and Price* es la versión correspondiente cuya estrategia se basa en la generación de columnas sobre la matriz de restricciones del problema de programación lineal. Es decir, se basa en la inclusión de nuevas variables sobre una versión relajada del problema inicial. Puesto que puede dar lugar a confusión, es necesario detallar cómo se puede llevar esto a cabo para reflejar de manera adecuada la diferencia respecto del método de generación de filas. Mientras que en la estrategia de generación de filas, la relajación se produce permitiendo que las variables de decisión tomen valores que no satisfacen las restricciones originales, en este caso el enfoque consiste en resolver una versión más simple del problema original para después, mediante un proceso iterativo, ampliar el alcance del problema simplificado hasta alcanzar la definición del mismo que satisfaga todas las restricciones de la

versión original. El proceso de ramificación surge cuando no es posible satisfacer las condiciones impuestas tras la inclusión de nuevas variables, donde esta es forzada con la esperanza de que sea satisfecha en la siguiente iteración, creando una ramificación por cada posible alternativa.

Aplicado al problema *Dial-a-Ride*, una posible estrategia se apoya en limitar los aristas disponibles del grafo en una primera iteración, para posteriormente aplicar un proceso inclusión/eliminación de aristas del grafo para así controlar la complejidad del problema a la vez que se exploran distintas soluciones. El proceso de ramificación de soluciones surge cuando son eliminados aristas que en alguna de las soluciones hayan sido utilizados. Para llevar a cambo dicho proceso de inclusión/eliminación en la literatura se han propuesto estragias apoyadas en *Programación dinámica*.

3.2.1.3. Branch and Cut and Price

Una vez descritos lo métodos basados en generación de filas (*Branch and Cut*) así como de columnas (*Branch and Price*), es fácil entender el funcionamiento del método *Branch and Cut and Price*, que se apoya en la combinación de ambas estrategias para reducir así reducir la complejidad de resolución del problema *Dial-a-Ride*. Como es natural, estos cuentan con la ventaja de resolver instancias más fáciles de resolver del problema por tener un menor número de restricciones (eliminación de filas) así como de hacerlo sobre versiones más pequeñas del problema (eliminación de columnas). Tal y como se ha podido comprobar en distintas publicaciones que comparan la utilización de los métodos de generación de filas o columnas de manera separada, frente a las correspondientes versiones combinadas, estas últimas obtienen un nivel de resultados equivalente requiriendo de un menor nivel de cómputo.

Tal y como se ha podido comprobar a lo largo del apartado, los métodos de resolución apoyados por técnicas basadas en ramificación son capaces de ofrecer ventajas relacionadas con la simplicación del problema completo a resolver a costa de requerir la resolución de múltiples versiones relajadas de distintas formas de dicho problema. Cuando el tamaño de la instancia del problema es tan grande que esta no puede ser resuelta "de una sola vez", estos métodos ofrecen una alternativa interesante por seguir ofreciendo garantías de optimalidad, lo cual en ciertos casos es un requisito necesario.

3.3. Métodos de Resolución basados en Heurísticas

Tal y como se ha explicado anteriormente, los problemas de optimización combinatoria se corresponden en la mayoría de ocasiones con problemas cuya complejidad desde el punto de vista de la resolución no es sencilla. Tal es así que muchos de estos se engloban dentro de la categoría de problemas NP, lo cual se puede resumir como aquellos para los cuales no existe una estrategia

de resolución exacta que alcance el resultado óptimo en tiempo polinómico. Esto implica que para instancias no demasiado grandes no sea factible alcanzar dicho resultado en un tiempo aceptable.

Para paliar esta problemática, surgen métodos de resolución denominados *estrategias heurísticas*. Estas se caracterizan por ser capaces de proporcionar resultados razonablemente cercanos a solución óptima empleando para ello una cantidad de recursos (generalmente tiempo de cómputo) significativamente menor a la que requerirían otras estrategias con ganarantías de optimalidad. Este es un compromiso aceptable cuando no es posible llegar a soluciones de otro modo. Por lo tanto, uno de los grandes objetivos de la investigación y el desarrollo de heurísticas consiste en implementar algoritmos que sean capaces de alcanzar soluciones lo más cercanas a la solución óptima, teniendo en cuenta las restricciones de tiempo para que estas puedan ser alcanzadas.

Desde una perspectiva de alto nivel, los métodos de resolución basados en heurísticas típicamente son clasificados en dos grandes categorías según la función que desempeñan, las cuales se conocen como *heurísticas de construcción* y *heurística de mejora*. A pesar de que su nombre es lo suficientemente explicativo, a continuación se detalla la función que representan los métodos pertenecientes a cada una de estas categorías:

- **Heurísticas de Construcción**: la función que desempeñan consiste en la generación de soluciones sin el apoyo de ningún estado previo. Es decir, recibiendo únicamente los valores de entrada necesarios para la formulación del problema, estas son capaces de generar soluciones del problema que satisfacen todas las restricciones impuestas por la formulación del mismo. En el caso de los problemas de rutas de vehículos, estas estrategias se apoyan comúnmente en algorimos de inserción inspirados en algoritmos greedy y, en algunas ocasiones con un cierto nivel de aleatoriedad. Por lo general, los resultados a los que es posible llegar utilizando únicamente estrategias de este tipo, no son lo suficientemente cercanas al resultado óptimo. Para resolver esta problemática, existen otras estrategias que tratan de paliar esta situación.
- **Heurísticas de Mejora**: debido al grado de dificultad necesario para tomar las decisiones más acertadas durante el proceso de construcción de soluciones, la categoría de *heurísticas de mejora* trata de mejorar la calidad una solución dada. Para ello, como es esperable, estas estrategias necesitan conocer tanto el conjunto de valores de entrada necesarios para la formulación del problema, como una solución inicial del mismo. A partir de estas dos partes, el modo de funcionamiento consiste en aplicar distintas modificaciones a la solución para así tratar de maximizar (o minimizar) el valor de la función objetivo, lo cual acerca dicha solución al valor óptimo deseado. A este modo de funcionamiento se le conoce como búsqueda local dentro del espacio de soluciones. En el caso de los problemas de rutas de vehículos, las estrategias en que se basan consisten en la aplicación de operadores tales como el intecambio de tareas (o viajes) entre distintos vehículos, cambio del orden dentro del mismo vehículo, etc. Por tanto, el objetivo se trata de buscar el conjunto de transformaciones de la solución

que permitan un mayor acercamiento al resultado esperado al final de la ejecución. Esta última afirmación se explicará más en detalle en el apartado 3.3.3

Una vez se han detallado las diferencias entre las dos categorías principales de estrategias de resolución basados en heurísticas, se está en condiciones suficientes como para explicar en detalle algunos de los algoritmos concretos a través de los cuales es posible alcanzar dichas soluciones. El resto del apartado se estructura de la siguiente forma: en el apartado 3.3.1 se detalla la estrategia basada en *fuerza bruta* (la cual como veremos no es aplicable en la práctica en la gran mayoría de casos). Seguidamente, en el apartado 3.3.2 está dedicado a la estrategia basada en *selección aleatoria*. Después se describirá en detalle la estrategia basada en *algoritmos greedy* (la cual es muy popular por su sencillez conceptual y relativamente buenos resultados). Posteriormente se discutirá el funcionamiento de la estrategia de *clarke and wright* y un método basado en *redes neuronales* en los apartados 3.3.4 y 3.3.5. Finalmente, se describirá la categoría de heurísticas conocidas como de *Búsqueda Local*, las cuales presentan una naturaleza diferente tal y como se comenta en el apartado correspondiente.

3.3.1. Fuerza Bruta

A pesar de que anteriormente se ha comentado que una de las grandes diferencias entre métodos de resolución exactos y métodos de resolución basados en heurísticas son las garantías de optimalidad, esta distinción es un tanto difusa cuando se tratan de clasificar las estrategias de fuerza bruta. Esto se debe a que, si bien es cierto que esta estrategia es capaz de garantizar resultados óptimos cuando termina, la estrategia de resolución se encuentra más próxima a los algoritmos heurísticos que a los métodos que hemos clasificado como exactos, puesto que estos se apoyan en mayor medida en la teoría de programación lineal, mientras que las estrategias heurísticas tienden a acercarse más a estrategias algorítmicas clásicas. Para una descripción más detallada y profunda del tema se recomienda consultar [Cor+09].

Una vez aclarada la peculiaridad de la estrategia de *fuerza bruta*, procedemos a describir en qué se basa: El modo de funcionamiento consiste en la enumeración de todas las posibles configuraciones, para después proceder a filtrar todas aquellas que sean factibles, y de entre estas, escoger aquella que maximize (o minimize) la función objetivo. La definición de esta estrategia en pseudo código se incluye en el algoritmo 3.1 (donde U representa el universo de posibles soluciones, $feasible$ se trata de una función que recibe una solución y se encarga de evaluar si es o no factible, F se compone de todas las soluciones factibles y por último, s se corresponde con la solución óptima, que en este caso se trata de un problema de minimización).

Sin embargo, tal y como se puede intuir, esta estrategia de resolución requiere de una cantidad de recursos que crece de manera exponencial respecto del tamaño de entrada. Por lo tanto, no es aplicable en la práctica sobre problemas de gran tamaño. Aún así, existen ciertos problemas para

Result: s
1 $F \leftarrow filter(feasible, U)$;
2 $s \leftarrow min\{F\}$;

Algoritmo 3.1: Estrategia de resolución basada en *fuerza bruta*.

los cuales el grado de complejidad es relativamente bajo o el tamaño de las instancias a resolver es pequeño, por lo que esta estrategia es aplicable. En dichos casos, esta es una alternativa interesante ya que, como se ha comentado anteriormente, los algoritmos basados en *fuerza bruta* proporcionan garantías de optimalidad.

Una de las ventajas de los métodos de resolución basado en fuerza bruta es la capacidad de poder aprovecharse de la estructura del problema concreto que se pretende resolver para así poder reducir el espacio de búsqueda de soluciones en gran medida. Esto se debe a que el espacio de posibles soluciones factibles es un subconjunto del espacio de posibles soluciones. Por tanto, si se consigue encontrar alguna estrategia que permita enfocarse únicamente en las soluciones factibles, los requerimientos computacionales serán mucho menores. Un ejemplo de ello podría aplicarse a una estrategia de generación de soluciones en problemas de rutas a partir de una estrategia de inserción al final de la ruta. Entonces, es fácil darse cuenta de que si durante el proceso de inserción de tareas en la ruta, se llega a un estado de infactibilidad, continuar el proceso de insertar nuevas tareas al final de la ruta, continuará generando soluciones infactibles. Por tanto, en este caso es posible reducir el espacio de búsqueda sin perder las garantías de optimalidad.

3.3.2. Selección Aleatoria

La siguiente estrategia heurística a comentar se presenta en muchos sentidos como una versión opuesta a la de *fuerza bruta*. Esta es la heurística basada en *selección aleatoria*, cuya principal característica es el apoyo en una componente de estocasticidad que pretende, a largo plazo, alcanzar resultados razonables utilizando para ello un reducido coste computacional. Entonces, tal y como se puede intuir esta estrategia es opuesta a la de *fuerza bruta* en dos sentidos: (1) la búsqueda en el espacio de soluciones no es exhaustiva, por lo que no se prueban todos los casos posibles (lo cual elimina las garantías de optimalidad) y (2) el coste computacional que conlleva generar soluciones factibles es mucho menor puesto que la decisión sobre qué camino explorar se delega en selección aleatoria.

Tal y como se puede apreciar en el algoritmo 3.2, la estrategia se basa en generar soluciones aleatorias del universo de posibles soluciones para después evaluar la factibilidad de la escogida en cada iteración y almacenarla en caso de que esta maximice (o minimice) el valor de la función objetivo. Esto se repite mientras se cumpla una determinada condición, la cual puede basarse en

un determinado número de iteraciones, una cantidad de tiempo o estrategias más elaboradas como un ratio de mejora respecto entre la última mejor solución obtenida y la anterior, etc.

```
   Result: s
1  s ← ∅;
2  while condition do
3  |  c ← random(U);
4  |  if !feasible(c) then
5  |  |  continue;
6  |  end
7  |  s ← min(s, c);
8  end
```

Algoritmo 3.2: Estrategia de resolución basada en *selección aleatoria*.

Tal y como se puede intuir, por si sola esta heurística no es todo lo eficiente como se espera para ser aplicable en la práctica puesto que, a pesar de la simplicidad que proporciona durante el proceso de generación y evaluación de soluciones, resulta muy improbable que llegue a alcanzar soluciones razonables sin que la estrategia se apoye en cierto conocimiento del problema. Sin embargo, tal y como se comentará más adelante, esta estrategia sirve como base (o componente) de otras mucho más elaboradas que consiguen alcanzar resultados mucho mejores en el largo plazo mediante la utilización de cierta componente aleatoria en alguna parte del proceso de optimización. Desde el punto de vista conceptual, hacer esto consigue añadir cierto ruido en el camino de búsqueda hacia la solución óptima que puede ser positivo dado que este reduce la complicación de "caer" en máximos (o mínimos) locales que en ciertas estrategias heurísticas complican en gran medida el proceso de optimización.

3.3.3. Greedy

Una de las estrategias más conocidas (por su buena relación entre sencillez y calidad de las soluciones) son las basadas en *algorithmos greedy*. Estos se corresponden con una de las principales categorías en las cuales se suelen dividir los algoritmos en la mayoría de libros de relacionados. La principal característica de este tipo de algoritmos es que cada una de las decisiones que toman se convierten en invariantes para los próximos pasos que lleve a cabo la estrategia en la búsqueda de la solución final. De ahí el nombre *greedy* (o voraz en español), que trata de remarcar la idea de que cada una de las decisiones que tomadas terminarán formando parte de la solución final. Nótese que esta condición conlleva dos consecuencias principales (1) la simplicidad tanto a nivel conceptual como conceptual al requerir únicamente del estado actual (obtenido a partir de las decisiones tomadas anteriormente y las alternativas actualmente posibles), y (2) la restricción de reevaluar decisiones anteriores que permitirían llegar a una solución más acercada con el conocimiento obtenido en el momento actual.

Sin embargo, existen ciertos problemas que poseen ciertas características que permiten que los algoritmos de naturaleza voraz sean capaces de llegar al resultado correcto (u óptimo dependiendo de la naturaleza del problema) de manera eficiente y a la vez sencilla. El ejemplo más notable es el de el *problema del cajero*. Dicho problema consiste en la elección sobre qué monedas utilizar de entre un conjunto dado de posibles valores (los cuales no tienen restricciones de cardinalidad) de tal manera que el número de monedas que sea mínimo. A modo de ejemplo, suponiendo el conjunto de valores posibles $\{1, 5, 10, 25, 100\}$, la asignación que minimiza el número de monedas para devolver el valor 34 es de $[25, 5, 1, 1, 1, 1]$. La estrategia para llegar a esta solución consiste en seleccionar siempre la moneda de mayor valor que no supere el resto necesario para llegar al valor deseado. En el algoritmo 3.3 se muestra el pseudo código de la estrategia, donde S representa la solución, v el valor deseado, C el conjunto de posibles monedas y c una variable temporal. Tal y como se puede apreciar, siguiendo esta estrategia se cumplen las condiciones requeridas po los *algoritmos de naturaleza voraz* ya que únicamente se tiene en cuenta el estado actual para tomar la siguiente decisión y no se modifican las decisiones tomadas anteriormente. Para más información acerca de las características de esta categoría algorítmica así como ejemplos más interesantes se recomienda consultar [Cor+09].

```
   Result: S
1  S ← [];
2  while v ≠ 0 do
3  |   c ← min_{v−c>0}{C};
4  |   v ← v − c;
5  |   S ← S + [c];
6  end
```

Algoritmo 3.3: Estrategia de resolución del *problema del cajero* basada en *algoritmo greedy*.

Desde el punto de vista de los problemas de optimización combinatoria, los *algoritmos de naturaleza voraz* han sido de gran utilidad para muchas situaciones. Sin embargo, tal y como se puede intuir, estos requieren que el problema sobre el que se van a aplicar cumpla la propiedad de ser concavos (una mejora sobre una solución arbitraria en la dirección de la solución óptima garantiza un menor coste en la función objetivo) para que estos puedan alcanzar la solución óptima. Por contra, esto ocurre en muy pocas ocasiones en la mayoría de problemas de optimización combinatoria. Si bien es cierto, hay casos para los cuales esta condición si se cumple (como el problema del cajero), pero en muchos otros (entre los que se encuentran los *problemas de rutas de vehículos*) esta condición no se satisface.

A pesar de ello, tal y como se ha indicado anteriormente, estos algoritmos son capaces de proporcionar resultados relativamente razonables utilizando una cantidad de recursos muy contenida por lo que cuando es tolerable cierto nivel de error respecto de la solución óptima, como es el caso

cuando se recurre a estrategias basadas en heurísticas, estos algoritmos proporcionan resultados muy interesantes. Como es natural, el modo en que estos se aplican es muy dependiente de la naturaleza del problema en cuestión. Sin embargo, en el ámbito de los *problemas de optimización combinatoria* es común que la estructura de los mismos tenga unas cierta forma: **Por lo general, los algoritmos greedy se construyen como un bucle condicional donde se controlan cuestiones relacionadas con el alcanzamiento de una solución final, ya sea debido a la falta de recursos u otros motivos. La acción que se repite en el bucle típicamente es la selección de la mejor alternativa de entre un conjunto de posibles opciones que modifican tanto el valor de la función de coste como el estado actual de una determinada manera.**

Entonces, un factor decisivo a la hora de aplicar *algoritmos de naturaleza voraz* se basa en elegir de manera adecuada el criterio de selección que se utilizará para elegir la siguiente alternativa que aplicar tras cada iteración del bucle condicional. En el siguiente apartado se describen algunos de los más utilizados en los problemas de optimización combinatoria.

3.3.3.1. Criterios de Selección

[TODO]

3.3.3.2. Randomized Greedy

A pesar de que la elección de un criterio de selección adecuado para el problema que se pretende resolver, existen ocasiones en que estos no son lo suficientemente "buenos" como se podría esperar. Por la propia naturaleza de los algoritmos de naturaleza voraz, en muchas ocasiones no es suficiente el contexto actual para escoger la decisión más acertada a largo plazo.

Para tratar de suavizar el peso de dicha decisión uno de los enfoques posibles es el de añadir un cierto grado de aleatoriedad en el momento de elegir el la mejor opción a realizar. Es decir, en lugar de seguir una estrategia que escoja siempre el la opción que maximize (o minimice) el valor del criterio de selección, se escoge aleatoriamente la siguiente opción de entre las mejores posibles. Como se ha comentado, la idea subyacente tras esta estrategia es que se asume que el criterio de selección escogido, a pesar de ser óptimo en el momento de su utilización (resultado parcial), probablemente no lo sea respecto del resultado completo. Entonces, la componente aleatoria trata de reducir dicho efecto generado por las "malas decisiones" tomadas.

Generalmente, para llevar a cabo la selección aleatoria se mantiene una lista ordenada con las k mejores opciones posibles (de acuerdo con el criterio de selección) para después utilizar algun generador aleatorio de valores de una distribución aleatoria discreta que finalmente eligen la alternativa que finalmente utilizar. Estas distribuciones pueden ser de distinta naturaleza, aunque

las más comunes son la la distribución uniforme discreta o alguna distribución armónica decreciente respecto del orden obtenido por el criterio de selección.

Como es esperable, estas estrategias tienden a mejorar los resultados obtenidos por los métodos greedy simples en promedio, pero obtienen peores resultado en el peor caso. Por lo tanto, su uso está muy extendido en metaheurísticas basadas en la generación de muchas soluciones. En concreto, este método se utiliza por las metaheurísticas *GRASP* tal y como se comentará en el apartado 3.4.1.

3.3.4. Clarke and Wright

Una de las estrategias heurísticas más conocidas para los problemas de optimización de rutas (y que es también aplicable al problema *Dial-a-Ride*) es la de ahorros (*Savings*) definida por Clarke y Wright en [CW64]. La estrategia esta definida asumiendo que no existen limitaciones en cuanto al número de vehículos disponibles para resolver el problema, por lo que esto es algo a tener en cuenta para su utilización. Sin embargo, es posible realizar pequeñas modificaciones que permitan su utilización con restricciones del número de vehículos. Una manera trivial de hacer esto es la de eliminar tantas rutas como sea necesario hasta que se cumplan las restricciones del número de vehículos sobre la solución generada por la heurísticas, aunque también se pueden aplicar otros enfoques como una penalización de costes elevada que limite el uso de más vehículos.

En cuanto al modo de funcionamiento de esta heurística, en un paso inicial genera una ruta desde almacen a cada tarea (en el caso del problema *Dial-a-Ride* esto consistirá en las dos paradas que requiere el viaje, la de recogida y la de entrega). Esto hace que sean en una situación inicial sean necesarios tantos vehículos como tareas a satisfacer. Posteriormente, se lleva a cabo un proceso iterativo de fusión de rutas mientras los costes de estas sean reducidos. Es decir, se calculan el coste total que genera cada par de rutas de manera independiente para posteriormente compararlo con el coste obtenido de combinar ambas rutas. En cada ciclo del algoritmo se escoge la opción que minimiza los costes y esta exploración se repite hasta que no haya opciones de mejora posibles. Es interesante remarcar que las estrategias de combinación de rutas no están limitados a la concatenación, sino que la estrategia se basa en la valoración de todos los posibles puntos en que una ruta se puede incluir en la otra. Esto hace que sea una estrategia de muy exhaustiva en el sentido de ser capaz de probar muchas situaciones.

Sin embargo, el motivo de la popularidad que ha recibido esta heurística durante tanto tiempo no solo se debe al método en si, sino a que además en [CW64] se describe la manera en que poder calcular el nivel de ahorro (de ahí el nombre de la estrategia) de la combinación de dos rutas sin tener que calcularlo los costes completos de ambas rutas y después hacer la diferencia respecto de la versión combinada. La estrategia en que se apoyan para llevar esto a cabo es la diferencia entre los costes de los tramos de inicio y fin de la ruta que se va a insertar frente a los costes de los

tramos de unión entre ambas rutas. Nótese que esta estrategia es altamente eficiente cuando las tareas tienen restricciones muy flexibles tanto desde el punto de vista de las ventanas temporales como de la capacidad del vehículo. En problemas más complejos como el *Dial-a-Ride* estas mejoras han de ser tomadas con prudencia ya que es necesario verificar la factibilidad de la ruta resultante.

3.3.5. Redes Neuronales

En los últimos años, el aumento de la capacidad computacional (sobre todo aplicable a operaciones de naturaleza matricial apoyado en gran parte por la evolución de las *unidades de computación gráficas (GPUs)*) a propiciado un incremento exponencial la cantidad de aplicaciones reales basadas en técnicas basadas en redes neuronales (cuyos requerimientos computacionales encajan perfectamente en dicho marco). Además de las aplicaciones prácticas, también ha surgido un gran incremento desde el punto de vista de la cantidad de artículos científicos relacionados con la materia. Entre otros temas, uno de los contextos en que se ha tratado de estudiar la factibilidad de este tipo de técnicas han sido los problemas de optimización combinatoria. Algunos de los más destacados son [Pot93; GO03; LJX04; MC09; Bel+16]. Tal y como se puede apreciar, estos trabajos tienen en común la utilización de enfoques basados en *redes neuronales recurrentes* así como *mapas auto-organizados*. A continuación se describen brevemente este tipo de arquitecturas:

1. **Redes Neuronales Recurrentes**: Consisten en una clase de redes neuronales con la característica de utilizar los pesos de ciertas neuronas de las capas internas como valores de entrada, creando una relación de recurrencia con la que se espera obtener la capacidad de tener un estado interno en la red. Es por ello que su utilización en el campo del análisis de lenguaje natural es muy interesante.

2. **Mapas Auto-Organizados**: Se trata de una arquitectura de red neuronal que se caracteriza por tener en cuenta la localización de las neuronas para el entrenamiento de la misma (que se organizan en forma de rejilla). Esto implica que los pesos de estas estén basados en la posición de las mismas. Es por ello que se su utilización en procesos de visualización de datos utilizados en tareas de clustering es muy interesante.

Tras haber descrito brevemente estas dos arquitecturas de redes neuronales es fácil darse cuenta de las ideas en las que se apoyan los artículos anteriormente citados. En el caso de las técnicas basadas en redes neuronales recurrentes se espera "descubrir" un patrón de resolción común inherente a un problema de rutas concreto y no visible de manera natural, que las capacidades recurrentes de la red sean capaz de captar. En el caso de las técnicas basadas en mapas auto-organizados, la idea es que (por la estructura en forma de mapa de la red) se descubran patrones de cercanía entre las distintas localizaciones a visitar durante la ruta, con las que después se puedan reconstruir caminos de coste reducido.

Dichas estrategias presentan un gran potencial sobre aquellos problemas que presenten las características comentadas anteriormente. Sin embargo, hasta la fecha dichas ideas se encuentran todavía en fases muy tempranas de investigación por lo que no son aplicables en situaciones reales (en [Bel+16] fueron necesarias 7 horas de computación para resolver una instancia del problema del viajero de 50 localizaciones). Sin embargo, si que forman parte de un interesante nicho de investigación que podría ser muy prometedor en el futuro.

3.3.6. Búsqueda Local

[TODO]

En este apartado se han descrito las principales estrategias heurísticas utilizadas en problemas de optimización de rutas, las cuales sirven de base para resolver las características concretas que se presentan en el problema *Dial-a-Ride*. Tal y como se comentó al principio del mismo, estas estrategias destacan por su relativa sencillez, tanto conceptual como computacional. Sin embargo, en muchas ocasiones no proporcionan resultados suficientemente buenos (desde el punto de vista de su distancia a la solución óptima) como se requiere. Para contrarestrar dicha situación surge una nueva clase de estrategias de optimización compuesta por otras estrategias más sencillas la cuál se discute en el próximo apartado.

3.4. Métodos de Resolución basados en Metaheurísticas

El concepto de *metaheurística* surge tras la necesidad de desarrollar estrategias de resolución apoyadas en otras (típicamente más sencillas) las cuales, en muchas ocasiones, no son capaces de proporcionar soluciones lo suficientemente satisfactorias por si solas a pesar de basarse en intuiciones coherentes para llegar a soluciones adecuadas pero que a su vez presentan alguna limitación que por si solas no les permite alcanzar el resultado deseado. En esta línea, la función de una *metaheurística* trata de tener en cuenta dichas limitaciones para así tomar las decisiones pertinentes que reduzcan dichos efectos no deseados.

A modo de ejemplo, este tipo de acciones podría ir desde limitar la voracidad de un algoritmo greedy para evitar caer en mínimos locales, hasta controlar el tiempo de ejecución de un método de resolución exacto, cuando el ratio de mejora relativa obtenido por este alcance una determinada cota mínima. Sin embargo, las *metaheurísticas* uno solo se limitan a este tipo de acciones posibles, sino que abarcan un gran número de posibilidades tal y como se verá a lo largo del apartado, la cual se orienta hacia las más populares en lo relacionado con el *problema Dial-a-Ride*. En primer lugar, en el apartado 3.4.1 se describirá la estrategia *GRASP*. Posteriormente, en el apartado 3.4.2 se describe la idea subyacente tras los métodos basados en *Simulated Anneling*. A continuación

se comenta la metaheurística de *Búsqueda Tabú*. Finalmente, se comentan las metaheurísticas basadas en *Busqueda de Vecindarios Variable*, *Búsqueda de Gran Vecindario* en los apartados 3.4.4 y 3.4.5 respectivamente. Finalmente, se comentan brevemente las *metaheurísticas basadas en poblaciones* en el apartado 3.4.6. Para una descripción más detallada acerca de las metaheurísticas más comunes, se recomienda consultar [BR03; GK06; BLS13; Ho+18].

3.4.1. GRASP

Una de las estrategias metaheurísticas más populares es la conocida como *GRASP (Greedy Randomized Adaptive Search) o búsqueda adaptativa aleatorizada voraz*. Esta metaheurística es muy simple conceptualmente pero a su vez muy potente a la hora de alcanzar resultados razonables. El modo de funcionamiento se basa en lo siguiente: se repiten de manera iterativa dos operaciones principales mientras no se cumpla una determinada condición de parada. Dichas operaciones consisten en la generación de una solución inicial, posiblemente basada en una estrategia *Greedy Aleatorizado*, aunque no necesariamente (existen estrategias basadas en la generación de soluciones iniciales totalmente aleatorias), y una segunda operación de mejora de dicha solución mediante una heurística de *Búsqueda Local*. La condición de parada, al igual que se ha indicado en otras ocasiones, puede venir dada por un determinado número de iteraciones sin soluciones mejores u otras estrategias de naturaleza similar.

```
   Result: best
1  while condition do
2  |  current ← initialSolution(U);
3  |  current ← localSearch(current);
4  |  best ← min{best, current};
5  end
```

Algoritmo 3.4: Estrategia de resolución basada en metaheurística *GRASP*.

En el algoritmo 3.4 se muestra una plantilla para la metaheurística *GRASP*, la cual sigue la misma estructura que se describió en el anterior párrafo. Tal y como se ha comentado, estas estrategias gozan de gran popularidad por su sencillez y buenos resultados. Además, entre otras, presenta la ventaja de poder ser paralelizada de manera muy sencilla dado que el contenido del bucle interno es independiente tras cada etapa (obviando la parte final de comparación con la mejor solución encontrada hasta el momento). Para más información acerca de esta metaheurística se recomienda consultar [RR16], dedicado integramente a su estudio y análisis.

3.4.2. Simulated Anneling

Las metaheurísticas basadas en *Simulated Anneling* se basan en la mejora continua de una solución inicial, siguiendo unos determinados criterios de parada (y parametrización de la etapa

de búsqueda local). Esto se contrapone con la estrategia *GRASP* descrita anteriormente, que se basaba en la generación de soluciones independientes para después elegir la mejor de todas ellas. En este caso, la estrategia consiste en la exploración del *vecindario de soluciones* de tal manera que la opción escogida no tenga por qué ser necesariamente la que más se aproxime a la mejor solución en un primer momento, llegando al punto incluso de poder elegir ciertas opciones que empeoren el la solución actual. La idea detrás de esta estrategia se apoya en la asunción de que en fases tempranas en la etapa de busqueda de la mejor solución, decidir una opción que mueva la solución actual a un punto más lejano de la solución óptima puede ser beneficioso a largo plazo para llegar a un punto más cercano al óptimo. Dicha capacidad de "tomar malas decisiones" se aplica con probabilidad decreciente respecto del tiempo de ejecución actual (ya que se asume que en fases más avanzadas de la ejecución dicho comportamiento no es tan beneficioso).

Dicha estrategia fue desarrollada tomando como apoyo el comportamiento de ciertos metales durante su proceso de fundición, que si se lleva a cabo bajo unas determinadas condiciones de enfriamiento es posible obtener un mayor grado de resistencia. Este proceso es conocido como *Simulated Anneling* (de ahí el nombre de la metaheurística). En el caso de la metaheurística, se aceptan soluciones negativas con cierta probabilidad dada por el resultado de la ecuación (3.1), donde t representa la temperatura (o valor decreciente respecto del número de iteraciones) y $f(s1)$ y $f(s2)$ los valores de la función objetivo para las soluciones $s1$ y $s2$ respectivamente. En el algoritmo 3.5 se incluye el pseudo-código de la metaheurística.

$$P(t, f(s1), f(s2)) = exp\left(-\frac{f(s1) - f(s2)}{t}\right) \tag{3.1}$$

```
   Result: best
1  while condition do
2  |  while annelingCondition do
3  |  |  current ← randomNeighbor(best);
4  |  |  if f(current) < f(best) or random(0,1) < P(t, f(current), f(best)) then
5  |  |  |  best ← current;
6  |  |  end
7  |  end
8  |  t ← decrease(t);
9  end
```

Algoritmo 3.5: Estrategia de resolución basada en metaheurística *Simulated Anneling*.

Esta metaheurística ha servido de base para la construcción de otras más avanzadas, como *Microcanonic annealing* basado en la propiedad de equilibrio de la termodinámica, *Threshold accepting method* basado en un threshold de decreciente o *Noising method*. También es fácil darse cuenta de que esta metaheurística puede utilizarse de manera hibrida junto con *GRASP* utilizando

Simulated Anneling como envoltorio sobre la operación de *Búsqueda Local*. Las técnicas que representan el estado del arte en el área de metaheurísticas de optimización actualmente se componen principalmente de estrategias hibridas como la descrita en este caso.

3.4.3. Tabu Search

La metaheurística conocida como *Tabu Search (o Búsqueda Tabú)* es una de las estrategias más conocidas y ampliamente utilizadas por sus buenos resultados. En concreto, en problemas de optimización de rutas existe una gran bibliografía al respecto demostrando sus buenos resultados, tanto para problemas más generales como para el *Dial-a-Ride* entre los que se encuentran [GHL94; CLM01; CL03].

La idea sobre la que se soportan las metaheurísticas basadas en *Tabu Search* consiste en la utilizació de una estructura de datos de memoria que almacena la lista (restringida) de últimas soluciones visitadas para así obviarlas en próximas iteraciones y no caer en ciclos (posiblemente infinitos) de exploración de soluciones ya visitadas. Generalmente, dicha memoria está restringida a un número máximo de soluciones, que se van descartando bajo una política *FIFO (First In, First Out)* lo cual incluye la posibilidad de visitar la misma solución varias veces con la esperanza de que mediante algún mecanismo aleatorio, el camino camino tomado en cada visita a la solución sea diferente. Respecto del tamaño de la memoria utilizada, el efecto que esto tiene en el proceso de optimización consiste en que memorias de menor tamaño tienden a intensificar la búsqueda sobre determinadas regiones del espacio mientras que memorias de gran tamaño tienden a realizar búsquedas más extensivas (pudiendo incluso desarrollar técnicas con memoria de tamaño variable, posiblemente decreciente).

```
   Result: current
1  memory ← ∅;
2  while condition do
3  |  current ← randomNeighbor(current, ignore = memory);
4  |  memory ← updateMemory(memory, current);
5  end
```

Algoritmo 3.6: Estrategia de resolución basada en metaheurística *Tabu Search*.

En el algoritmo 3.6 se incluye el pseudo-código que define la estructura común a las metaheurísticas basadas en búsqueda tabú. En cuanto a la intuición tras esta metaheurística, tal y como se puede apreciar, en ningún caso se restringe el movimiento hacia soluciones de "peor calidad" por lo que de esta manera se evita "caer" en mínimos locales durante el proceso de búsqueda de soluciones. Esta es una de las razones por las cuales esta metaheurística goza de tanta popularidad (sobre todo en problemas de *optimización de rutas* donde el efecto dichos mínimos locales dificulta

mucho el proceso). Por contra, esta metaheurística requiere de un mayor coste computacional, tanto a nivel de ejecución puesto que añade la necesidad de poder comparar si dos soluciones son o no iguales, lo cual en terminos de eficiencia no es trivial, como a nivel de espacio ya que requiere de mantener varias soluciones en memoria, lo cual es necesario tener en cuenta durante el proceso de implementación.

3.4.4. Variable Neighborhood Search

Una de las heurísticas más interesantes en problemas de optimización de rutas es la conocida como *Variable Neighborhood Search (o Búsqueda en Vecindarios Variables)* la cual se apoya en la idea de enfocar la optimización en mejorar distintos aspectos de la solución en cada fase del mismo para así terminar alcanzando una solución final de mejor calidad. Sin embargo, para poder describir esta estrategia en condiciones ideales lo primero es describir el concepto de *vecindario*.

En el contexto de espacios de soluciones un **vecindario** consiste en un conjunto de soluciones que posseen características comunes, o que pueden ser alcanzadas aplicando operaciones similares sobre una solución inicial dada. En el caso de los problemas de optimización de rutas un vecindario puede referirse al conjunto de posibles soluciones obtenidas tras aplicar la operación de permutar dos tareas contiguas de una ruta. Entonces, este es el vecindario generado por la aplicación de dicha operación sobre una solución. Nótese que este será disjunto respecto del generado por todas las soluciones obtenidas tras aplicar la operación de permutar dos tareas no contiguas entre si. Sin embargo, esto no tiene por qué ser así, puediendo haber vecindarios que se solapan entre si.

Una vez entendido el concepto de *vecindario*, comprender el comportamiento de esta metaheurística es muy sencillo. Se trata de seleccionar iterativamente una solución de cada vecindario escogida de forma aleatoria y seleccionarla como la actual mientras esta mejore la anterior, aplicando esta operación por cada vecindario mientras se sigan alcanzando mejore soluciones.

En el algoritmo 3.7 se muestra el pseudo-código de una posible implementación de la metaheurística *Variable Neighborhood Search*. Nótese que la ventaja de este enfoque consiste en enfocar la mejora en cada paso en una determinada propiedad inherente a cada vecindario. Esto puede ser incrementar el número de tareas, permutar tareas dentro de la ruta, permutar tareas entre rutas, etc.

3.4.5. Large Neighborhood Search

Hasta ahora, todas las metaheurística que se han analizado asumen la utilización de una heurística de construcción inicial, que es capaz de generar una solución por si misma. Sin embargo, existen otras alternativas en contraposición con dicha estrategia. Entre ellas se encuentra la metales conocida como *Large Neighborhood Search (o Búsqueda de Gran Vecindario)*. La idea subyacente tras

```
   Result: current
1  best ← initialSolution();
2  neighborhood ← initialNeigborhood();
3  while condition do
4  |  current ← randomNeighbor(best, neighborhood);
5  |  if f(current) < f(best) then
6  |  |  best ← current;
7  |  |  neighborhood ← initialNeigborhood();
8  |  else
9  |  |  neighborhood ← nextNeigborhood(neighborhood);
10 |  end
11 end
```

Algoritmo 3.7: Estrategia de resolución basada en metaheurística *Variable Neighborhood Search*.

dichas estrategia consiste en reaprovechar las soluciones obtenidas hasta el momento para apoyarse en variaciones de las mismas durante el proceso de contrucción de soluciones iniciales bajo la premisa de que estás tenderán a ser de "buena calidad" y el coste computacional de adaptarlas será menor que el de construir una nueva por completo.

En cuanto al nombre de la estrategia esta cobra sentido desde el punto de vista del camino generado por todas las soluciones visitadas, ya que en este caso en lugar de presentar conjuntos de soluciones aislados, la estructura está compuesta por un único camino de gran longitud y que (idealmente) abarca una gran superficie del espacio de soluciones.

Respecto de las operaciones aplicadas para generar nuevas soluciones iniciales, típicamente estas suelen basarse en heurísticas de eliminación de visitas (en el caso del problema *Dial-a-Ride* simultaneamente de origen y destino) y a su vez tratar de reañadirlas con heurísticas de inserción (posiblemente las mismas que las usadas para construir soluciones completas).

```
   Result: current
1  best ← initialSolution();
2  neighborhood ← initialNeigborhood();
3  while condition do
4  |  current ← initialFrom(best);
5  |  current ← localSearch(current);
6  |  best ← min{best, current};
7  end
```

Algoritmo 3.8: Estrategia de resolución basada en metaheurística *Large Neighborhood Search*.

En el algoritmo 3.8 se incluye el pseudo-código que representa la estructura básica de una implementación *Large Neighborhood Search*. Al igual que sucedía con otras estrategias descritas a lo largo del apartado, para el caso de esta también es posible desarrollar versiones adaptativas que conforme avanza el proceso de optimización reducen el grado de aleatoriedad (en este caso respecto de la generación de soluciones iniciales) para focalizarse más en la zona del espacio de soluciones conocido más cercano al óptimo deseado.

3.4.6. Metaheurísticas basadas en Poblaciones

Una de las técnicas que más ha gozado de popularidad en los últimos años es la de las *metaheurísticas basadas en poblaciones* por su mayor capacidad frente a situaciones como mínimos locales durante el proceso de optimización. Para entender cómo funcionan este tipo de estrategias, lo primer es ser capaces de diferenciarlas de las anteriormente descritas. Dichas metaheurísticas se engloban dentro de la categoría de *metaheurísticas basadas en trayectorias* dado que el modo de funcionamiento consiste en la generación de una solución en cada paso de la ejecución, seguido de una fase de evaluación de las soluciones pertenecientes a su correspondiente vecindario para posteriormente elegir una de las opciones para continuar con el camino (o trayectoria) de exploración. Nótese, que este camino puede estar basado en estructuras de árbol u otras estrategias más sofisticadas. Sin embargo, lo que caracteriza a este tipo de metaheurísticas es que estas se basan en "afinar" soluciones independientes (y posiblemente elegir la mejor de todas ellas).

Las *metaheurísticas basadas en poblaciones* se caracterizan por seguir una estrategia diferente: en lugar de mantener independencia entre soluciones, se utilizan transformaciones que aplican distintas combinaciones entre ellas, dando lugar a nuevas soluciones generadas a partir de la "fusión" de otras. Comúnmente, estas estrategias están basadas en la generación mediante algún método de construcción de soluciones aleatorias (y deseablemente ortogonales sobre el espacio de soluciones) para después generar nuevas soluciones a partir de posibles variaciones (apoyadas por metaheurísticas basadas en trayectorias) y combinaciones con otras soluciones. La idea subyacente tras esta idea es que a priori es muy complicado generar una solución que posea todas las características deseadas. Sin embargo, si que es posible que algunas de estas soluciones tengan ciertas de ellas. Por lo tanto, a través de transformación de estas soluciones iniciales en versiones combinadas de ellas es posible alcanzar soluciones finales que presenten un mayor número de características deseadas, y por tanto una mayor cercanía a la solución óptima deseada.

A modo de ejemplo, y para entender mejor cómo es posible implementar transformaciones que combinen varias soluciones sobre *problemas de optimización de rutas*, una manera muy simple e intuitiva puede ser la siguiente: asumiendo rutas con tareas exclusivas entre vehículos, se puede generar una nueva ruta como combinación de otras escogiendo para cada vehículo, la ruta con menor coste de entre todas las soluciones posibles, llegando de esta manera a una nueva solución con las *mejores características* de todas las anteriores. Como es natural, esta es una versión muy

simplicada acerca de cómo funcionan este tipo de transformaciones, ya que las instancias reales raramente cumplen dichas propiedades de exclusividad de tareas entre vehículos.

Para una descripción más detallada acerca de este tipo de estrategias y cómo aplicarlas en problemas como el *Dial-a-Ride* se recomienda consultar [JLB07]. Para profundizar más en este tipo de técnicas también se recomienda [BA03] donde el autor aplica *Algoritmos Genéticos* para resolver el problema *VRP* o [BM04] donde se aplica la metaheurística de *Optimización basada en Colonia de Hormigas*.

3.5. Conclusiones

A lo largo del capítulo se ha llevado a cabo una descripción acerca de las distintas estrategias de resolución más populares aplicables a problemas de rutas, entre los que se encuentra el *Dial-a-Ride*. Dicha descripción se ha llevado a cabo partiendo de las estrategias más clásicas como los métodos exactos basados en la modelización como un problema de programación lineal, continuando con técnicas que no proporcionan resultados con garantías de optimalidad, desde simples heurísticas, hasta estrategias más complicadas construidas a partir de estas denominadas metaheurísticas.

Antes de concluir con el capítulo, es necesario remarcar la existencia de otro tipo de estrategias sobre las cuales se han llevado a cabo distintas investigaciones en los últimos años, consistentes en la combinación de métodos exactos con otros basados en metaheurísticas. La idea subyacente tras este enfoque consiste en aprovechar la gran capacidad en cuanto a eficiencia de búsqueda que proporcionan los métodos tradicionales, pero a la vez apoyarse en distintas metaheurísticas que sean capaces de "equilibrar" la intensidad de uso de estos, e incluso combinarlos con otras técnicas más simples basadas en heurísticas (con un coste computacional mucho menor) de tal manera que se aprovechen las bondades correspondientes que cada estrategia proporciona.

Tal y como se ha comentado a lo largo del capítulo, estos métodos presentan distintas ventajas y desventajas que, dependiendo de la instancia concreta a resolver pueden ser más o menos perjudiciales. Entre estos factores se encuentran el tamaño del mismo así como los requisitos de exactitud. Por tanto, aplicar estos métodos sobre entornos o situaciones reales se convierte en una decisión relativa al contexto de aplicación donde, en lugar de utilizar la "estrategia ganadora", la tarea consiste en elegir la estrategia que mejor se ajuste a las necesidades, tanto desde el punto de vista de los costes computacionales (sobre todo a nivel de tiempo) como de los requisitos de exactitud tal y como se ha indicado previamente.

Capítulo 4

Implementación y Resultados

4.1. Introducción

Hasta ahora, el objetivo principal de este documento se ha orientado en la descripción y análisis del problema *Dial-a-Ride* desde una perspectiva principalmente teórica, incluyendo una descripción detallada centrada en la *Formulación del Problema* en el capítulo 2 así como un análisis sobre los *Métodos de Resolución* disponibles para el problema en el capítulo 3. A lo largo de dichos capítulos se ha podido apreciar tanto las dificultades inherentes al propio problema por las características propias que este presenta, tales como el tiempo máximo de duración de trayecto (calidad del servicio), así como otros factores relacionados con la complejidad computacional de resolver un problema de esta naturaleza (englobado en la clase de problemas *NP*).

Sin embargo, este capítulo representa un cambio de perspectiva del problema, estando centrado más en los detalles de implementación de una biblioteca bautizada como `jinete` y desarrollada sobre el lenguaje *Python*, cuyo enfoque consiste en proporcionar una serie de herramientas que sobre las cuales ser construir métodos de resolución de problemas de rutas. En una primera implementación, el desarrollo de dicha biblioteca se ha orientado principalmente hacia la resolución del problema *Dial-a-Ride* mediante una estrategía inspirada en la metaheurística *GRASP* descrita en el apartado 3.4.1. En el apartado 4.2 se incluye una descripción detallada acerca de la implementación realizada, así como las ideas en que ha inspirado la misma.

Para demostrar el funcionamiento de la implementación realizada, así como proporcionar una ejemplificación de su uso, se han utilizado una serie de instancias del problema disponibles en la web de manera pública, que han sido utilizados a modo de *benchmark* en una gran cantidad de documentos científicos de gran reconocimiento. El apartado 4.3 se destina a dicho cometido.

Por último, en el apartado 4.4 se expone una conclusión relacionada con el proceso de implementación de estrategias de resolución del problema *Dial-a-Ride*, indicando las principales debilidades de la implementación actual, así como sus posibles puntos de mejora.

4.2. Implementación

Tal y como se menciona en la introducción del capítulo, este apartado se dedica integramente a comentar la implementación realizada durante el desarrollo de este trabajo, donde se describen de manera detallada desde las decisiones tomadas de alto nivel (en el apartado 4.2.1) en la que se comentan aspectos como el lenguaje de implementación elegido o la metaheurística implementada de manera concreta, continuando con una descripción acerca de las decisiones tomadas en lo relacionado con un nivel de más bajo (en el apartado 4.2.2) en el que se describen los módulos más relevantes así como algunas de las decisiones algorítmicas tomadas. Además, a lo largo de la descripción se trata de presentar una breve discusión acerca de las distintas decisiones de diseño tomadas a lo largo del proceso, tratando de destacar las principales ventajas e inconvenientes de cada una de ellas.

4.2.1. Decisiones de Alto Nivel

El objetivo principal de la implementación realizada puede ser resumido en una frase: **Razonar en detalle la implementación de un sistema capaz de generar soluciones válidas (y lo más cercanas al valor óptimo posible) para el problema Dial-a-Ride**. Sin embargo, tal y como se puede apreciar, dicha frase es de carácter muy general y puede abarcar una gran cantidad de tareas a realizar, desde el estudio y adaptación de restricciones definidas como desigualdades en un modelo lineal para que estas después sean implementadas sobre uno de los sistemas comerciales más populares como `FICO Xpress Mosel` o `IBM CPLEX`, hasta el diseño de alguna metaheurística por completo en algún lenguaje como `C++`, `Rust` o `Python`. Tal y como se puede intuir, cualquiera de estas tareas puede llegar a ser por si sola motivo de una trabajo de tesis con el conveniente grado de profundización en la materia. Sin embargo, dado que en este caso el trabajo se ha llevado a cabo sobre el contexto de un *Trabajo de Fin de Grado* (con sus correspondientes limitaciones temporales), la implementación que se ha llevado a cabo consiste en una tarea intermedia entre estos dos caminos, a costa de un menor grado de profundidad en cada uno de ellos, pero que ha permitido conocer las complicaciones reales que la literatura relacioanda con el problema *Dial-a-Ride* pretende resolver para que las soluciones alcanzadas sean aplicables sobre situaciones reales y no queden en meros ejercicios teóricos.

Una de las decisiones importantes dede el punto de vista de los objetivos de la implementación es elegir el lenguaje sobre el cuál se va a llevar a cabo. El motivo de sobre la relevancia de dicha decisión desde el punto de vista del objetivo de la misma está intimamente relacionada con el alcance del trabajo. En este caso, los principales factores que han influido en dicha decisión han sido la capacidad de generar una implementación relativamente elaborada en un tiempo relativamente reducido, la experiencia en el propio lenguaje y el ecosistema de herramientas relacionadas compatibles con el mismo. Por contra, se han obviado otros factores como el grado de eficiencia del mismo (el cual es muy importante en este tipo de sistemas) pero que se ha dejado en un segundo

nivel por la naturaleza didáctica del mismo. Tras tener en cuenta todos estos factores, la decisión tomada ha sido la de elegir `Python 3` [Ros95] como lenguaje sobre el cual realizar la implementación. Este proporciona la capacidad de llevar a cabo implementaciones relativamente grandes en un periodo de tiempo inferior a la que se requeriría por otros lenguajes más verbosos como `Java` o `C++` además de proporcionar una sintaxis con una alta legibilidad, lo cual es una factor muy beneficioso en el contexto de los métodos de resolución de problemas como el *Dial-a-Ride*. Otro de los factores es la experiencia adquirida sobre este lenguaje, lo cual es un factor muy facilitador. Sin embargo, la elección escogida también presenta desventajas, sobre todo de escalabilidad a largo plazo y extensión de la misma desde un contexto didáctico hacia otro más aplicable a situaciones reales. Esto se debe a la ineficiencia que este lenguaje presenta frente a otros con una interfaz a un nivel más bajo, que (a costa de una mayor complejidad de desarrollo) son algunos órdenes de magnitud más rápidos. Este es uno de los factores que se comentarán en los próximos pasos, destacando la posibilidad de llevar a cabo una reimplementación de la biblioteca actual sobre el lenguaje `Rust`.

En relación con las ideas expuestas en los anteriores párrafos, y obviando la decisión sobre el lenguaje sobre el cuál se ha llevado a cabo la implementación, uno de los factores más importantes para entender la implementación realizada ha sido la naturaleza de la misma. En este caso, en lugar de tratar de llevar a cabo el desarrollo de una serie de ficheros que permitan resolver únicamente el problema *Dial-a-Ride*, sin tener en cuenta las posibles variantes que este pueda llegar a tener, así como las partes en común que distintos métodos de resolución compartan, en este caso se ha primado el análisis de dichas partes. De este modo se ha conseguido alcanza una estructura en forma de biblioteca, la cual se caracteriza por proporcionar una interfaz de utilización externa, sobre la cuál definir por ejemplo la manera de "cargar" el fichero de entrada que contiene la entrada del problema así como la estrategia de resolución que se desea utilizar e incluso el modo en que se exportarán los resultados. Por otra parte, se ha tratado de mantener oculta al exterior toda la complejidad inherente a este tipo de problemas de optimización, lo cuál es un punto a favor para ser utilizado por personas que no sean lo suficentemente expertas en la materia. En relación con la naturaleza en forma de biblioteca, también se ha tratado de priorizar la estructura de composición en los métodos de resolución. Es decir, en lugar de proporcionar una serie de implementaciones bien definidas, se ha pretendido proporcionar una interfaz extensible y que favorezca la composición. Esto se puede apreciar en situaciones como la definición (o eleeción) de los criterios de selección en ciertas estrategias de naturaleza *greedy* o la capacidad de componer de manera personalizada distintas metaheurísticas que requieren de fases de inserción llevadas a cabo por otras posibles heurísticas (o metaheurísticas). Cuando se ejemplifique su utilización esto se podrá apreciar de manera más clara.

Una de las partes más importantes a la hora de estudiar *problemas de rutas* como el *Dial-a-Ride* consiste en el análisis del mismo desde un pusto de vista más teórico. Uno de los caminos

para llevar esto a cabo es el estudio de la formulación como *problema de programación lineal* donde, cómo se ha indicado en el capítulo 2, es necesario definir tanto la naturaleza del conjunto de variables de decisión y las restricciones que estas presentan entre si, como la función objetivo (en este caso de minimización). Para que dicho ejercicio de estudio teórico del problema sea completo, también es una buena práctica implementarlo en uno de los solvers de propósito general más populares como `CPLEX` o `Mosel` de tal manera que se puedan apreciar los resultados así como comprender las dificultades computacionales desde una perspectiva mucho más cercana. Sin embargo, en este trabajo se ha seguido un enfoque ligeramente distinto para tratar de mantener lo más unificada posible la implementación llevada a cabo en `Python` con la obtenidas por distintos solvers. Para ello, se ha utilizado una heramienta desarrollada por la *COIN-OR Foundation* cuyo cometido es el de proporcionar una interfaz de comunición entre `Python` y los solvers más conocidos, la cual se conoce como `PuLP` [MOD11]. Para desenpeñar su tarea, esta biblioteca proporciona una interfaz sobre la cual llevar a cabo la formulación del problema a partir de sentencias como `prob = LpProblem("myProblem", LpMinimize)` (para definir el problema), `x = LpVariable("x", 0, 3)` (para definir variables), `prob += x + y <= 2` (para definir restricciones) o `status = prob.solve()` (para proceder a la optimización del problema). Internamente, dicha biblioteca es capaz de comunicarse con una gran cantidad de solvers (en el apartado de resultados se incluye una breve comparativa entre estos). Posteriormente, se ha añadido un nivel de abstracción superior capaz de convertir los resultados de la formulación lineal en conceptos relacionados con los problemas de rutas como *Vehículo*, *Ruta* o *Viaje*. De esta manera es sencillo realizar comparaciones entre métodos de resolución basados en heurísticas frente a otros basados en métodos exactos, además de compartir la misma implementación tanto para la lectura de los datos como para exportación de los resultados.

Hasta ahora se ha comentado el objetivo de la implementación realizada desde el punto de vista de algunas de las decisiones de implementación tomadas a priori (tales como el lenguaje utilizado o la filosofía seguida durante el proceso), sin embargo, aún no se ha expuesto en detalle la problemática concreta que dicha implementación es capaz de resolver, ni la estrategia seguida para llevarlo a cabo. En cuanto a la problemática concreta que esta biblioteca trata de resolver, es fácil intuir (por ser el tema principal del trabajo) que se trata de ser capaz de generar soluciones factibles para el problema *Dial-a-Ride*. Por contra, la segunda parte de la cuestión requiere de una clarificación en mayor profundad: Por una parte, esta implementación es capaz de generar soluciones basándose en métodos de resolución de problemas lineales gracias a la formulación del problema *Dial-a-Ride* de dicha forma, tal y como se ha comentado en el párrafo anterior. Por otro lado, actualmente la biblioteca es capaz de generar soluciones siguiendo una estrategia metaheurística de naturaleza *GRASP*, lo cual implica, entre otros, la implementación de una algoritmo de inserción *Greedy*, una heurística de *Búsqueda Local* capaz de mejorar una solución previamente generada en este caso, por la estrategia *Greedy*, y la propia lógica de control de la estrategia *GRASP*, encargada de controlar si se cumplen las condiciones suficientes para llevar

a cabo un nuevo ciclo de mejora de la solución actual así como otros aspectos relacionados. Es importante remarcar que todos estos módulos pueden ser utilizados añadiendo distintas puntos de aleatoriedad (para así tratar de reducir el alcanzamiento de mínimos locales "sin salida"). De forma complementaria a la capacidad de añadir una módulo de aleatoriedad, también se incluye la capacidad de apoyarse en un "generaror de múltiples soluciones" que es capaz de integrarse en cualquier fase del proceso. Posteriormente se detallará más este módulo, el cual se encarga de orquestar la generación de soluciones con distintas módulos de aleatoriedad y posteriormente seleccionar cual de todas ellas debe continuar con el proceso de optimización.

A lo largo de este apartado, se ha llevado a cabo una descripción acerca de las distintas decisiones tomadas a lo largo del desarrollo de la implementación desde una perspectiva externa, exponiendo las razones y tratando de comentar otras alternativas a las finalmente escogidas. Estas han sido desde comentar el lenguaje utilizado, la naturaleza de la implementación, su conexión con los métodos de resolución basados en formulación como problema de programación lineal, hasta el alcance de la metaheurística implementada de manera demostrativa.

Por otro lado, es importante remarcar también las decisiones tomadas en lo relativo al desarrollo de software muy importante en casos como los métodos de resolución en *problemas de optimización combinatoria*, donde el gran peso de la módulo algorítmica (sobre todo en lo relacionado con *metaheurísticas*) es crucial para alcanzar los resultados deseables. Dicho razonamiento se llevará a cabo a lo largo del próximo apartado.

4.2.2. Decisiones de Bajo Nivel

En este apartado se pretende llevar a cabo una descripción desde un punto de vista más técnico y detallado sobre las decisiones tomadas a lo largo de la implementación de la biblioteca en cuestión. Para ello, se dedican distintos sub-apartados a cada uno de los factores más relevantes a tratar. En primer lugar, se ilustra la organización de la biblioteca, dividida en módulos en el apartado 4.2.2.1. A continuación, se comenta el modelo de datos definido para resolver el problema, el cual guarda una estrecha correlación con el definido en el capítulo 2 donde se lleva a cabo una breve descripción de la nomenclatura utilizada. Finalmente, se trata de profundizar con un mayor nivel de detalle en cada uno de los módulos anteriormente indicados a lo largo de los apartados 4.2.2.3 a 4.2.2.5.

4.2.2.1. Organización Basada en Módulos

La estructura en que se ha organizado la biblioteca se ha definido tratando de tener en cuenta la facilidad para ser extendida y ampliada con el paso del tiempo, de tal manera que sea sencillo incluir tanto nuevas estrategias de resolución más sofisticadas, como la capacidad de poder modelar otros problemas de rutas de vehículos de una manera sencilla. Para ello, se ha tratado de prestar

especial importancia al desacoplamiento entre los distintos elementos que la componen así como la decisión de dar un sentido único y bien definido a cada uno de ellos.

Estos conceptos se entienden de manera mucho más sencilla a través de un ejemplo: en la implementación llevada a cabo, el módulo encargado de proceder a la carga de datos se considera desacoplado del resto puesto que la única comunicación que soporta consiste en las peticiones para llevar a cabo la carga de datos, así como la de suministrar estos en un formato ya adaptado para que pueda ser utilizado por el resto de la biblioteca (el cual se comenta en el apartado 4.2.2.2). En cuanto al sentido único que este módulo presenta, dicha característica consiste en que es fácil comprender la función que desempeña, lo cual facilita el proceso futuro de crear nuevas versiones de dicho módulo para que, por ejemplo, en lugar de cargar los datos desde un fichero concreto alojado en la máquina en cuestión, este sea capaz de recibirlo desde una dirección específica de internet u otro tipo de estrategias de definición más complejas como la carga desde una base de datos. Dichas capacidades de extensión permiten integrar la implementación realizada en sistemas reales de una forma muy sencilla y mantenible.

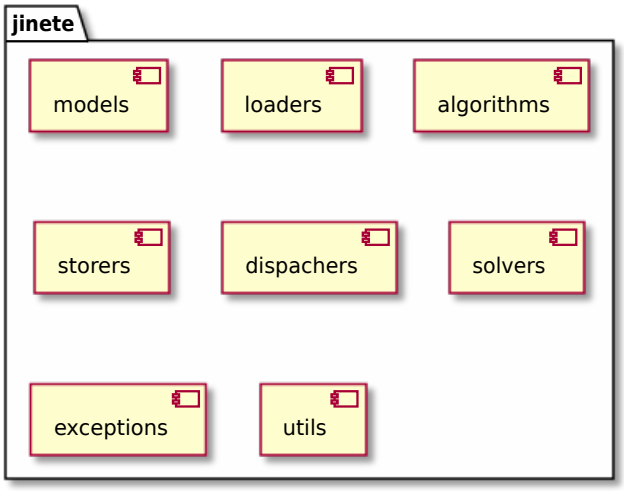


Figura 4.1: Diagrama basado en módulos de la biblioteca `jinete`

En cuanto a la organización completa de la biblioteca, esta se divide en 4 módulos principales, que además de apoyarse entre si, utilizan otros módulos de menor tamaño tal y como se comentará a lo largo del apartado. En la figura 4.1 se incluye una representación gráfica de estos. En cuanto a los módulos principales, a continuación se proporciona una breve descripción acerca de cada uno de ellos (el orden en que se describen se corresponde con el flujo natural en que estos se interrelacionan lo cual pretende servir para clarificar aún más su funcionamiento):

- `models`: Es el encargado de definir las entidades que representan los distintos elementos necesarios para modelar los distintos problemas de rutas, entre los que se encuentran clases como `Vehicle`, `Route` o `Stop`. Dichas entidades son las encargadas de poseer la información acerca de las restricciones del problema (como por ejemplo la ventana temporal en que

es factible llevar a cabo parada de recogida para un determinado viaje), pero además se encargan de recoger el estado tras cada etapa de ejecución del proceso de optimización (como por ejemplo el instante de tiempo en que se encuentra un determinado vehículo al haber realizado un secuencia concreta de viajes).Dichas entidades representan además un "contrato" entre el resto de módulos y submódulos de la biblioteca, lo cual es una de las herramientas que permite hacer más mantenible y ampliable la implementación realizada. Posteriormente se detalla de manera más precisa, pero a modo de ejemplo es fácil apreciar las ventajas de definir una entidad común `Route` que pueda ser utilizada por las distintas estrategias de optimización sin que estas requieran de otra característica adicional, lo que permite flexibilizar la capacidad de combinación y/o composición de dichas estrategias. En el apartado 4.2.2.2 se lleva a cabo una descripción más detallada acerca del módulo.

- `loaders`: Tal y como se ha comentado anteriormente a modo de ejemplo, la responsabilidad de este módulo consiste en ser capaz de cargar los datos desde una fuente de datos externa sobre la cual estos están definidos siguiendo un determinado formado posiblemente distinto del deseado, el cual consiste en las definidas por el módulo `models`. De esta manera, es posible aplicar indistintamente las distintas técnicas y herramientas implementadas en la biblioteca ignorando la estructura de los datos originales. Más adelante se describe de manera más detallada, sin embargo es interesante remarcar que en dicha tarea hay dos dimensiones principales: la "lectura" de la fuente de datos, la cual puede ser desde la carga de un fichero hasta el acceso a una base de datos o la realización de una petición `HTTP` a un sistema externo, del "procesamiento" del propio formato de los datos, que puede ser el definido por distintos autores o sistemas externos.En el apartado 4.2.2.3 se lleva a cabo una descripción más detallada acerca del módulo.

- `algorithms`: La responsabilidad de este módulo consiste en alojar las herramientas necesarias para poder llevar a cabo el proceso de optimización del problema en cuestión. Es decir, es quien contiene la implementación tanto de los algoritmos heurísticos tales como asignación voraz o búsqueda local, así como las metaheurísticas que se encargan de coordinar su ejecución. Además, contiene la lógica necesaria para ser capaz de resolver el problema a través de solvers externos de optimización lineal (métodos exactos). El modo de funcionamiento de estos algoritmos consiste en "rellenar" o "modificar" el estado de distintas entidades como `Route` o `Stop`, de tal manera que el formato siga siendo consiste para que puedan seguir siendo utilizadas tanto durante otras etapas del proceso de optimización como por el correspondiente módulo encargado de exportar dichos resultados. En el apartado 4.2.2.4 se lleva a cabo una descripción más detallada acerca del módulo.

- `storers`: Es el módulo encargado de almacenar los resultados obtenidos tras el proceso de optimización. este espera recibir una planificación compuesta por uno o más vehículos, la cual almacena de distintas formas según sea la entidad concreta elegida para el proceso. Nótese que en este caso el término "almacenar" se utiliza en un sentido más amplio, ya que se refiere

tanto al proceso de almacenar los resultados en un medio persistente como un fichero o una base de datos, como al proceso de mostrar un informe detallado en la terminal de salida o mostrar una representación gráfica de las rutas construidas. En el apartado 4.2.2.5 se lleva a cabo una descripción más detallada acerca del módulo.

Además de los módulos descritos anteriormente, la biblioteca implementada (`jinete`) incluye otros módulos de menor relevancia, pero igualmente importantes para el correcto funcionamiento de la misma. A continuación se incluye una breve descripción acerca de estos:

- `dispachers`: La labor que desempecha este módulo consiste en orquestar la comunicación entre `loaders`, `models` y `storers`. Como es natural, la estrategia más sencilla consiste en la comunicación unidireccional que carga una instancia del problema, aplica el proceso de optimización y después exporta los resultados en un medio persistente. Sin embargo, con la extensión de este módulo es posible llevar a cabo otras estrategias que permitan procesos de planificación en tiempo real entre otros.

- `solvers`: La responsabilidad que desempeña es la de simplificar el proceso de instanciación necesario para proceder con la optimización. El motivo es que para la mayoría de situaciones, no es neceario llegar a un nivel elevado de detalle para poder utilizar la biblioteca, por lo que este módulo se encarga de simplificar al máximo el proceso.

- `exceptions`: Contiene la definición acerca de las distintas situaciones de error que pueden darse durante el proceso de ejecución de la biblioteca, que después son comunicados a través de excepciones al usuario de la misma.

- `utils`: Se encarga de definir todas aquellas herramientas necesarias para el correcto funcionamiento de la biblioteca pero que no necesariamente se refieren al contexto de optimización combinatoria, tales como estructuras de datos o funciones auxiliares.

Una vez se ha llevado a cabo una descripción acerca de todos los módulos que forman la implementación realizada, en los siguientes apartados se procede a profundizar con un mayor grado de detalle en los más relevantes, entre los que se encuentran el modelo de datos utilizado, el proceso de carga de instancias, la optimización de la instancia del problema y finalmente, el proceso de almacenamiento de resultados. Por el contexto del trabajo, se presta especial detalle a los apartados del modelado y resolución del problema, indicando más brevemente los procesos de carga y almacenamiento de datos.

4.2.2.2. Modelo de Datos

Para poder resolver en condiciones adecuadas un problema de optimización combinatoria tan complejo como el *Dial-a-Ride*, donde es necesario tener en cuenta una gran cantidad de situaciones,

y a su vez, entender cómo han sido alcanzadas, un buen punto de partida es el de aclarar y uniformizar los conceptos que después serán utilizados por los métodos de resolución. En este caso, el punto de partida para llevar a cabo dicho proceso consiste en las entidades ya definidas previamente, en el apartado 2.3.1 donde se comentaron todos muchos de los terminos y palabras clave utilizadas para comentar el problema. Dichas definiciones han servido como punto de partida para definir las entidades de datos utilizadas a lo largo de la implementación realizada, las cuales en su gran mayoría se relacionan de manera biyectiva. Por lo tanto, el resto del apartado se dedica a indicar brevemente dicha adaptación de los conceptos en entidades de datos a la vez que se describe como estas se relacionan entre si. Además, en la figura 4.2 se incluye un diagrama que ilustra la descripción que se llevará a cabo a continuación.

Siguiendo el mismo orden que el llevado a cabo en el apartado 2.3.1, la primera entidad a comentar es `Position`, que como su propio nombre indica, se trat de la estructura de datos encargada de modelizar una posición en el espacio. Esto es, aquella que almacena las coordenadas que "apuntan" a un determinado punto del espacio. Como es natural, esta es utilizada por el resto de entidades con necesidades de ser localizables en el espacio. En este caso se corresponden con `Service` (actividad de recogida o entrega relacionada con un viaje) y `Stop` (acción de moverse hacia un determinado lugar llevada a cabo por un vehículo). Además, para mantener en un contenedor común todas las posiciones surge la entidad `Surface`.

La entidad `Surface` es la encargada de almacenar, proveer e interactuar con entidades `Position`. Es decir, es la encargada de realizar los cálculos de distancias entre distintos puntos del espacio. Por tanto, tiene la responsabilidad de definir la métrica utilizada en caso de que la distancias sean aritméticas (*Euclídea*, *Manhattan*, etc.) o de calcularlas apoyándose en un servicio cartográfico externo entre otros.

Siguiendo con las entidades que se encargan de definir las restricciones del problema (posteriormente se continúa con aquellas que recogen el estado y por tanto, el resultado de la optimización) es necesario comentar la labor que desempeña `Service`. Tal y como se ha dicho anteriormente, esta representa el requisito de llevar a cabo una determinada acción en una determinada posición del espacio. Es decir, es quien contiene información acerca de la ventana temporal disponible para visitar una determinada posición por un vehículo. Además, ofrece la posibilidad de indicar la duración requerida de dicho servicio. Esto es útil para modelar los tiempos de carga y descarga correspondientes. Sin embargo, `Service` no se utiliza únicamente para representar las tareas de carga y descarga de los viajes. Además, es utilizada para representar el concepto de tiempo de las ventanas temporales de inicio y fin de los vehículos. Es decir, es la manera de representar que un vehículo puede comenzar a operar durante un cierto intervalo de tiempo, y puede volver al almacén en otro determinado intervalo de tiempo. En caso de que ambos puntos sean el mismo, nótese que si dichas ventanas temporales (de comienzo y fin) son disjuntas entonces dicho vehículo

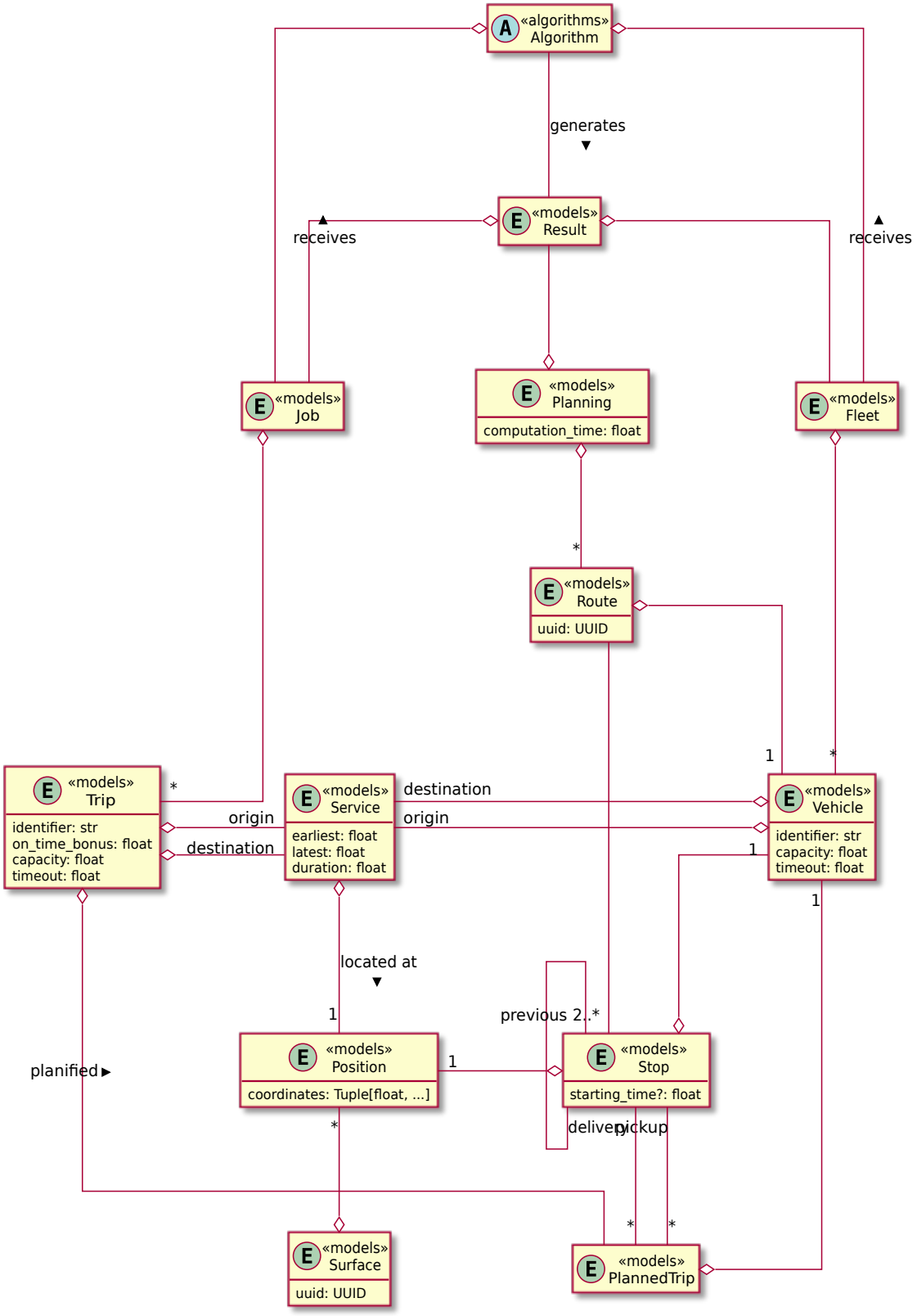


Figura 4.2: Modelo de datos simplificado de la biblioteca `jinete`.

estaría obligado a realizar al menos un viaje (lo que puede generar situaciones de infactibilidad de la solución del problema si no hay suficientes viajes disponibles).

Una de las entidades más importantes desde el punto de vista de la formulación del problema es `Trip`. Esta es la encargada de "conectar" los servicios de recogida y entrega entre si, de tal forma que no puedan ser realizados indistintamente o de manera parcial. Es decir, es quien impone la restricción de que primero se ha de realizar la tarea de visita y, en un tiempo no superior a la duración máxima de viaje, se debe realizar la entrega. Posteriormente, cuando se lleve a cabo la descripción de la entidad `Stop` se expondrá de manera más detallada cómo es el proceso para llevar esto a cabo. Además de lo indicado, esta es la entidad encargada de indicar la cantidad de espacio (carga) requerido en el vehículo.

La siguiente entidad a remarcar consiste en `Job`, cuyo cometido es actuar como un contenedor de viajes (objetos `Trip`) de tal manera que puedan ser consultados de manera sencilla por otras entidades del modelo de datos. En este caso, `Job` se corresponde con una visión estática del problema, es decir, no posee ningún conocimiento sobre qué viajes han sido realizados y cuáles no (dicha tarea es responsabilidad de otras entidades).Además de contener todos los viajes del problema, esta también se encarga de contener la entidad `Objective`, la cual se encarga de representar la función objetivo a optimizar. Esta puede ser de naturaleza muy variada dependiendo de la versión concreta del problema de rutas a resolver. En este trabajo, la más utilizada es la del problema *Dial-a-Ride* por razones obvias, la cual trata de minimizar la distancia total llevada a cabo por los vehículos disponibles.

Dejando de lado la representación de las necesidades a satisfacer, el siguiente paso es comentar los recursos disponibles para llevarlas a cabo. Como en todos los problemas de rutas, la entidad fundamental que actúa como recurso disponible es la del vehículo. En nuestro caso, dicha función la desempeña `Vehicle`. En este punto, hay una decisión de diseño importante a tomar, que consiste en la elección entre: (a) utilizar la entidad vehículo para representar tanto las restricciones que este presenta (tales como capacidad máxima, tiempo máximo de ruta, etc.) como el estado actual (secuencia de tareas o viajes completados, tiempo actual, capacidad actual, etc.) o, por contra, (b) dividir dicha entidad en dos, siendo una de ellas la encargada de representar las restricciones anteriormente citadas y otra la encargada de almacenar el estado actual. Como es obvio, cada una una de estas alternativas posee distintas ventajas y desventajas, siendo la primera de ellas más simple pero menos versátil y la segunda, a costa de un mayor grado de complejidad, permite abstraer mejor los conceptos facilitando la implementación de estrategias de optimización avanzadas que requieran de mantener distintas versiones de un mismo vehículo de manera simultanea minimizando la duplicidad de información (y los posibles errores que puedan derivar de dicha situación). En este caso, se ha escogido la segunda opción, por lo que la entidad `Vehicle` es encargada únicamente de modelizar las restricciones de un vehículo sin mantener constancia de su estado.

Al igual que se ha indicado para el caso de `Position` y `Job`, es necesario añadir una entidad adiccional que actúe como contenedor de estas, siendo `Surface` y `Job` respectivamente en los

anteriores casos. Por lo tanto, el contenedor de endidades `Vehicle` se ha denominado `Fleet` y, al igual que ocurre con `Job`, su único cometido consiste en proveer de las correspondientes entidades a quien lo requiera. Nótese que el problema *Dial-a-Ride* asume una cantidad finita de vehículos, por lo que esto deberá ser acorde con el funcionamient de `Fleet`. Sin embargo, existen otros problemas como el *Pickup and Delivery Problem* en que no hay una cota máxima del número de vehículos (a costa de que todos sean iguales entre sí).

Una vez comentado el significado de todas las entidades de naturaleza estáticas del modelo de datos, el siguiente paso es proceder al razonamiento de las entidades dinámicas, encargadas de almacenar el estado de la optimización. Estas se caracterizan por estar construidas apoyándose en gran medida sobre las anteriores, además de presentar un mayor grado de dificultad tanto en su implementación como explicación por su naturaleza cambiante.

Siguiendo el orden del apartado 2.3.1, la siguiente entidad a comentar es `Stop`, cuya función principal es la de representar la realización de un determinado servicio, por un determinado vehículo un estado concreto (posteriormente se indicará cómo se lleva a cabo dicha representación del estado). La entidad `Stop` se relaciona con el resto de la siguiente manera: con la entidad `Vehicle` a la cual se refiere, con la entidad `Position` sobre la cual se localiza, con la entidad `PlannedTrip` (que se comenta más adelante) sobre la cuál es capaz de relacionarse tanto con el viaje y los servicios requeridos, como con la parada relacionada (de entrega futura si esta es de recogia, o de recogida previa si esta es de entrega). De esta manera es posible verificar si el viaje completo se ha llevado a cabo desde una de las paradas, que no necesariamente tienen por qué ser contiguas entre sí. Adiccionalmente, la entidad `Stop` incluye otra relación adiccional consigo misma, que representa la anterior parada. De esta manera, todas las paradas de un mismo vehículo están relacionadas entre sí como una lista enlazada. La ventaja de este enfoque es que los cálculos tanto de tiempo como de capacidad actuales, pueden ser calculados desde la propia entidad `Stop` y la versión anterior de esta generandose una *relación de recurrencia de primer orden.* Por último, esta entidad se relaciona con `Route`, que como veremos más adelante actúa como contenedor de paradas.

La siguiente entidad a comentar es `PlannedTrip`, que como su propio nombre indica se refiere a la realización de un viaje. Esta entidad guarda una *relación fuerte* con las dos paradas necesarias para completar el viaje (la de recogida y la de entrega). Además, esta relacionada con la entidad `Vehicle` correspondiente.

Anteriormente se ha indicado que la decisión tomada acerca de cómo representar las restricciones fijas de los vehículos así como su estado actual ha consistido en dividirlo en dos entidades. Ya hemos descrito `Vehicle`, que es la responsable de definir las restricciones. Sin embargo, todavía no hemos indicado cómo se almacena el estado. Para ello, se ha decidido utilizar la entidad `Route`, cuyo cometido es almacenar la secuencia de paradas (entidades `Stop`), y a su vez mantener una relación

a través de la cual sea posible recuperar las restricciones de la misma (`Vehicle`). La ventaja que esta estrategia presenta consiste en la "facilidad" para poder mantener varias instancias de la entidad `Route` referidas al mismo vehículo y al mismo tiempo para que las comparaciones entre estas (para elegir la más beneficiosa para el resultado final) sean mucho más sencillas. Esto es un factor importante en la resolución de este tipo de problemas donde la carga computacional derivada de la evaluación combinatoria es muy elevada.

Antes de finalizar, es necesario comentar en qué consiste la entidad `Planning`, la cuál actúa como un contenedor de entidades `Route`. Tal y como se ha comentado antes, estas se refieren a la secuencia de paradas para cada vehículo. Nótese que deben satisfacer la restricción de que un vehículo tan solo tenga una ruta a realizar por razones obvias. Por último, se añade una entidad más denominada `Result`, que se encarga de poner en común la flota de vehículos, el conjunto de viajes a realizar, y la asignación generada mediante las entidades `Fleet`, `Job` y `Planning` respectivamente. Lo interesante de esta entidad es que es la utilizada como valor de entrada y salida para muchos de los métodos de resolución implementados (entidades `Algorithm`), lo cual permite que estos puedan ser intercambiados y combinados entre sí fácilmente.

Tal y como se puede apreciar a lo largo del apartado, el modelado del problema de manera adecuada se ha correspondido con una tarea de gran peso para la implementación realizada. A lo largo de la descripción, se ha tratado de comentar la taxonomia definida manteniendo el equilibrio adecuado entre detalle y extensión del mismo, dejando de lado algunos detalles técnicos de menor relevancia para la visión general del modelo de datos.

4.2.2.3. Carga de Datos

El proceso de carga de datos en la implementación llevada a cabo consiste más concretamente en la lectura de los valores concretos que definen las diferentes instancias del problema a resolver, los cuales consisten en el número de viajes requeridos así como las características concretas de cada uno de ellos, o los vehículos disponibles para satisfacer dichos viajes, con sus corresponientes limitaciones tanto de tiempo como de carga. Este apartado se dedica a describir cómo es llevado a cabo dicho proceso desde la perspectiva del flujo de los datos, es decir, desde que se realiza la llamada que comienza el proceso de carga, hasta que están construidos por completo los elementos necesarios para llevar a cabo el proceso de optimización.

Tal y como se ha comentado anteriormente, el módulo destinado a dicha tarea se ha denominado `loaders`. En este se ha definido una interfaz `Loader`, la cual sirva para definir la estructura que deben tener todas las implementaciones destinadas a la carga de datos. Esta proporciona acceso a las correspondientes instancias `Surface`, `Fleet` y `Job` que definen el espacio geométrico, la flota de vehículos y los viajes a realizar respectivamente, tal y como se indicó en el anterior apartado. El modo en que se ha definido dicha interfaz permite que esta pueda cargar los datos de manera

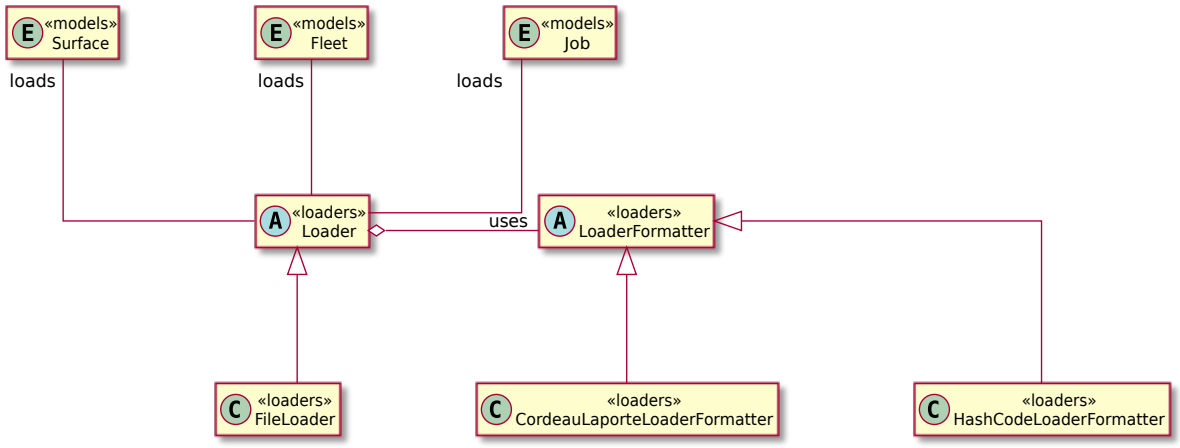


Figura 4.3: Diagrama de clases del módulo `loaders` perteneciente a la biblioteca `jinete`.

lazy (o bajo demanda), es decir, que la carga no se lleve a cabo hasta el momento inmediatamente anterior a su utilización.

La idea sobre la interfaz `Loader` es que las entidades que la implementen se encargen de proveer de acceso a los datos en crudo, pero sin profundizar en el formato de estos. Actualmente, se ha llevado a cabo la implementación `FileLoader`, que se encarga de cargar el contenido de un fichero. Una extensión natural podría ser la de implementar un `URLLoader` que se encargase de cargar el contenido alojado en una dirección web. Sin embargo, ninguna de estas implementaciones tiene la responsabilidad de "entender" el formato de estos datos y transformarlos en las entidades anteriormente citadas. Para ello se ha definido el concepto de `LoaderFormatter`, cuyo objetivo es el de construir las correspondientes entidades a partir de los datos. En este caso, la implementación principal es la de `CordeauLaporteLoaderFormatter`, que se encarga de generar las corespondientes instancias a partir del formato de definición de las instancias del problema definido por Cordeau y Laporte para las instancias usadas como *Benchmark* en [CL03]. El diagrama de clases que ilustra la manera en que se interrelacionan las clases definidas en el módulo `loaders` se incluye en la figura 4.3.

A modo de ejemplo, a continuacion se comenta la secuencia de acciones que son llevadas a cabo para la carga de datos utilizando la implementación `CordeauLaporteLoaderFormatter`. Como ejemplificación acerca del formato en cuestión, a través del siguiente enlace se puede consultar la instancia `a2-16.txt` utilizada en el benchmark del paper anteriormente citado: `http://fc.isima.fr/~lacomme/Maxime/ELSapproachDARP/instances/a2-16.txt`. En la primera línea se incluyen el número de vehículos disponibles, el número de paradas a satisfacer (incluyendo recogida y entrega, por lo que el número de viajes es la mitad), el tiempo máximo de viaje por vehículo, el número de viajes que pueden compartir un vehículo de manera simultánea y la duracción máxima de viaje (contabilizado desde que se deja el punto de recogida hasta que se alcanza el punto de entrega, es decir, únicamente es contabilizado el tiempo de trayecto para esta restricción).

Seguidamente se incluye una secuencia con todas las paradas a satisfacer, donde la primera mitad se refiere a las recogidas mientras que la segunda mitad representa las entregas. Por tanto, suponiendo n viajes, la parada i representa la recogida y la parada $i+n$ la entrega (por lo que, como es obvio, existen $2n$ paradas). Cada parada se define de la siguiente forma: un identificador de parada, las coordenadas del punto donde satisfacer la parada. En este caso, están definidas sobre un espacio euclídeo bidimensional por lo que las distancias son calculada aplicando la norma dos (o $d(x,y) = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$). Seguidamente, se indica la duracción del tiempo de carga/descarga. A continuación, se incluye la capacidad requerida por el viaje. Nótese que esta es opuesta en la recogida y entrega, siendo la primera positiva y la segunda negativa. Esto facilita la integración de la instancia en formulaciones como problema de programación lineal. Finalmente se incluyen las ventanas temporales en que satisfacer cada parada. En este caso, algunas de ellas son abiertas y otras cerradas, dependiendo de si el viaje es de tipo *inbound* (recoger en un intervalo prefijado) u *outbound* (entregar en un intervalo prefijado).

Una vez comentado el formato de entrada, el siguiente paso es entender cómo este es transformado en las entidades definidas en el modelo de datos. El primer paso es la construcción del espacio geométrico, es decir, la instancia `Surface`. Esta es común entre la flota de vehículos y el conjunto de viajes a realizar por lo que es necesario que sea el primer elemento a construir. Dicha implementación incluye un método `get_or_create_position(coordinates: Tuple[float]) -> Position` por lo que el proceso es muy sencillo al no requerir de una lectura inicial de todas las coordenadas. A continuación se construye la secuencia de vehículos, la cual requiere únicamente de la primera línea del fichero de datos, creando posteriormente la instancia `Fleet`. Finalmente, se procede a la creacción de la entidad `Job`, compuesta por el conjunto de viajes (o instancias `Trip`). Para construir estas se procede a la iteracción de las restantes filas siguiendo el enfoque indicado anteriormente $(i, i+n)$ para enlazar tanto la parada de recogida como la de entrega. Una vez construidas las correspondientes instancias de `Surface`, `Fleet` y `Job` ya se está en condiciones necesarias para proceder a la fase de optimización del problema, la cual se describe en el siguiente apartado.

4.2.2.4. Optimización del Problema

El módulo `algorithms` representa una de las partes más interesentas (a la vez que críticas) de la implementación llevada a cabo. La lógica que implementa es la encarga de llevar a cabo la resolución de problemas en la biblioteca `jinete` por lo que la explicación acerca de cómo esta funciona es especialmente interesante. Tal y como se ha indicado previamente a lo largo del apartado dedicado a la descripción de la carga de datos, responsabilidad del módulo `loaders`, en el cual se ha definido una interfaz denominada `Loader`, en este caso se ha hecho lo correspondiente definiendo la interfaz `Algorithm`. Dicha interfaz espera recibir como entidades de entrada una instancia `Fleet` y otra `Job`, que representan las características de la flota de vehículos y el conjunto de viajes a realizar respectivamente. Nótese que en este caso no se requiere de una instancia `Surface`, lo cual es debido a que esta viene dada de manera implícita sobre las posiciones (`Position`) que componen

el resto de entidades. Para proceder con la ejecución, la interfaz `Algorithm` ofrece un método `optimize() -> Result`, que como resultado genera una nueva entidad `Result`, la cual contiene la planificación generada para la flota de vehículos y conjunto de viajes dados. Esto se comenta de manera más detallada en el apartado dedicado al almacenamiento de resultados, lo cual es responsabilidad del módulo `storers`.

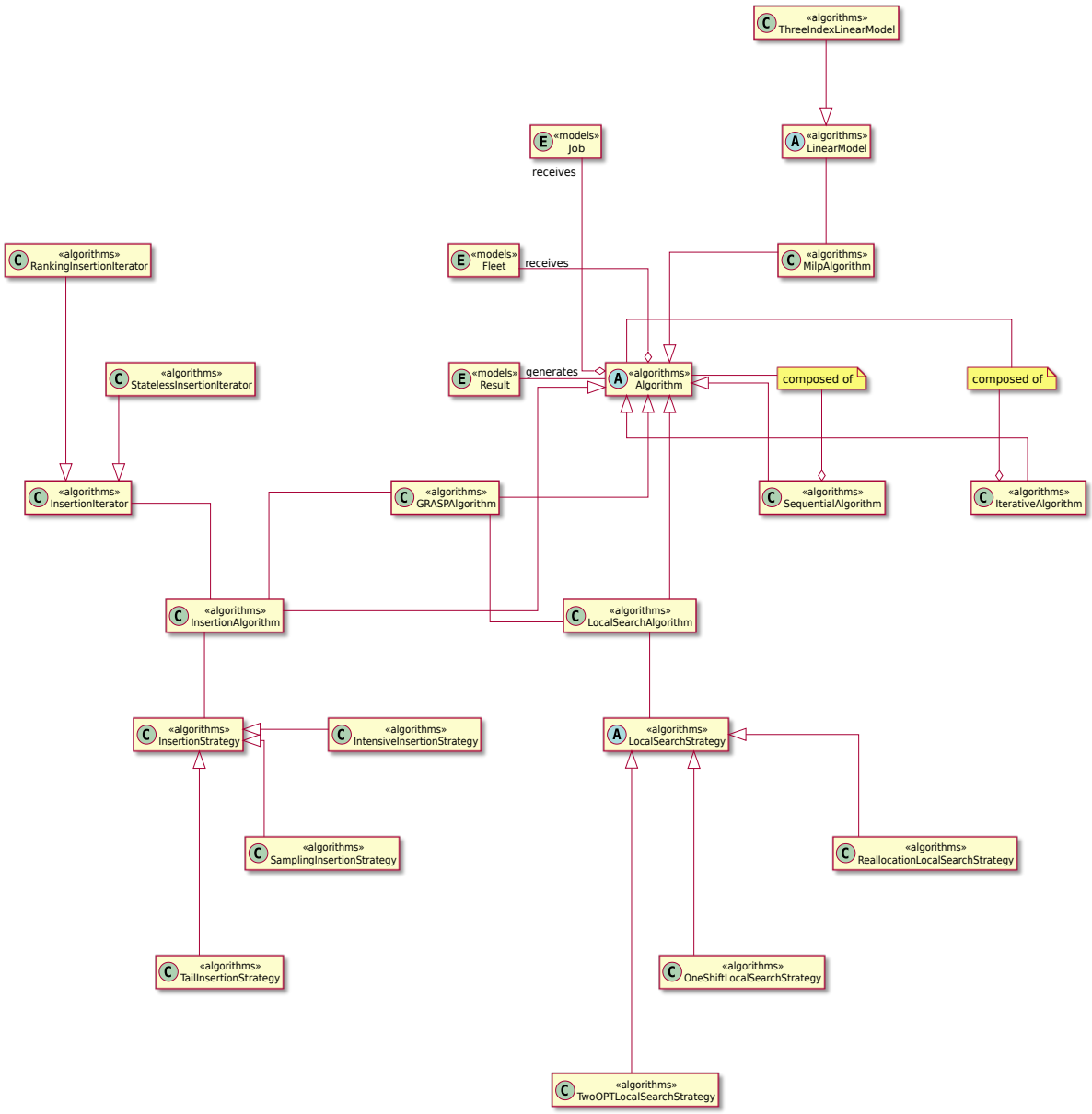


Figura 4.4: Diagrama de clases del módulo `algorithms` perteneciente a la biblioteca `jinete`.

Una vez entendido el modo de funcionamiento definido por la interfaz `Algorithm`, es posible dejar de lado la manera en que se intercomunican las distintas implementaciones, tanto entre sí como con otras entidades externas para poder enfocar la descripción en la funcionalidad que desempeñan cada una de ellas. Para tratar de simplificar el razonamiento acerca de cómo se relacionan, en la figura 4.4 se incluye el diagrama de clases del módulo `algorithms`. Sin embargo, este no proporciona una

descripción acerca de la función que desempeña cada una de las subimplementaciones más allá del nombre que estas tienen. Por lo tanto, a continuación se proporciona una descripción más detallada acerca de estas:

- `Algorithm`: Actúa como la interfaz que define la manera de utilizar las distintas subimplementaciones, tanto desde el punto de vista de los parámetros de entrada mínimos (dependiendo de la subimplementación estas pueden requerir ciertos parámetros opcionales), como del valor obtenido tras su ejecución, la cual se lleva a cabo mediante la llamada al método `optimize()` tal y como se indicó previamente.

- `InsertionAlgorithm`: La funcionalidad que desempeña esta subimplementación consiste en insertar nuevos viajes a rutas previamente existentes, así como crearlas en caso de que el correspondiente vehículo se encontrase en un modo ocioso (sin haber sido utilizado hasta el momento). Para proceder a la inserción de nuevos viajes en las distintas rutas disponibles, es necesario definir la forma en que estos serán emparejados respectivamente. Para ello, lo primero es definir la manera de recorrer todas estas combinaciones disponibles, que dependiendo del tamaño del problema y la configuración llevada a cabo, pueden llegar a ser inabarcables. Puesto que el algoritmo de inserción funciona de manera iterativa tratando de añadir todos los viajes que pueda mientras sea posible, parece interesante definir la manera en que iterar los pares `(Route, PlannedTrip)`. En este caso es importante remarcar que las combinaciones han de llevarse a cabo sobre viajes planeados, de tal forma que se tenga en cuenta el hueco en que se pretenden insertas las correspondientes paradas de recogida y entrega. Esto es debido a que dicho factor afectará de alguna forma tanto al coste como al estado resultante de la ruta (de ahí que el número de casos a evaluar se convierta en elevado). Como es natural, la iteración de estos pares se lleva a cabo de tal manera que primero se comprueben aquellos cuya valoración (o beneficio) sea mayor según el criterio correspondinte:

  - `RouteCriterion`: Se compone por una clase abstracta que espera recibir como entrada la dirección de optimización (maximización o minimización) y expone métodos de ordenación y cálculo de los k mejores. Para poder aprovecharse de estos es necesario implementar el método |scoring(route: Route) -¿float—, que se encarga de calcular la puntuación de una determinada ruta.
  - `EarliestLastDepartureTimeRouteCriterion`: Se trata de un criterio de minimización y calcula la puntuación de la ruta como el tiempo final de la ruta. Por tanto, se valora más positivamente la ruta que antes haya terminado.
  - `ShortestAveragePlannerTripDurationCriterion`: Se trata de un criterio basado en minimización y la puntuación es calculada como el promedio de la duración de cada viaje de la ruta. Es decir, la diferencia de tiempos entre entrega y recogida.
  - `ShortestTimeRouteCriterion`: Es un criterio de minimización del tiempo de ruta. Es decir, el tiempo que pasa desde que el vehículo sale del almacén hasta que este regresa.

- `LongestTimeRouteCriterion`: Es un criterio de maximización del tiempo de ruta. Es decir, es el opuesto del anterior y la intuición que este presenta es la de tratar de aprovechar al máximo el vehículo escogido.
- `LongestUtilTimeRouteCriterion`: Se trata de un criterio de maximización que contabiliza la distancia recorrida por el vehículo con carga. La intuición tras este criterio consiste en reducir el tiempo en que el vehículo viaja vacío.

Una vez comentadas las distintas alternativas disponibles para valorar una ruta, el siguiente paso es definir los modos en que puede iterarse a través de la secuencia generada por dicho orden.

- `InsertionIterator`: Define la interfaz de iteración de las rutas recibiendo como entrada tanto el conjunto de rutas como el criterio de valoración a utilizar.
- `RankingInsertionIterator`: Mantiene un estado interno a lo largo de la ejecución del algoritmo de insercción, de tal manera que tras cada cambio de estado se recalculan únicamente las valoraciones que hayan sufrido cambios para posteriormente ser reordenadas. Puesto que el número de pares puede crecer de manera exponencial, existe la posibilidad de generar un ranking truncado donde se eligen aleatoriamente $k < n$ viajes donde n representa el número total de viajes, para después ir añadiendo cada vez un nuevo viaje de entre los $n - k$ restantes. De esta manera es posible controlar la eficiencia computacional de la implementación a costa de obtener resultados ligeramente peores.
- `StatelessInsertionIterator`: Se trata de una versión que no mantiene el estado. Tras cada nueva iteración recalcula la valoración para todos los pares de rutas y viajes planificados, escogiendo el que minimice (o maximice) el criterio escogido.

Una vez que se ha definido tanto el criterio de valoración de las rutas como la forma de iterar sobre los pares ya solo queda una pieza para poder completar la descripción del algoritmo de insección, la cuál es la propia estrategia de inserción. En este caso, el concepto de estrategia se refiere a la forma en que es posible insertar nuevos viajes en las rutas y como se ha comentado anteriormente, existen distintas alternativas:

- `InsertionStrategy`: Define la interfaz de insercción, la cual consiste en un método `compute(route: Route, trip: Trip) -> List[Route]` donde el resultado es una lista de rutas generadas a partir de la enviada como parámetro, solo que con el nuevo viaje añadido.
- `IntensiveInsertionStrategy`: La estrategia intensiva consiste en probar todos los pares (i, j) posibles para insertar las paradas de recogida y entrega sobre la secuencia de paradas ya existentes. Esta estrategia es la que mejores resultados presenta, sin embargo, el número de rutas a evaluar puede ascender a $m \cdot log(m)$ donde m es el número de paradas de la ruta, en el "peor" caso.

- `TailInsertionStrategy`: La estrategia de inserción de cola consiste en tratar de insertar el viaje al final de la ruta. En este caso tan solo se crea una nueva ruta por lo que, en términos de eficiencia computacional, es la mejor de todas. Sin embargo, en este caso no se aprovecha la posibilidad de *compartir vehículo* por varios viajes de manera simultanea. Este método es interesante para generar soluciones iniciales de manera eficiente que después puedan ser mejoradas por otras (meta-)heurísticas más avanzadas.
- `SamplingInsertionStrategy`: Para tratar de equilibrar la estrategia de inserción intensiva (de alto coste computacional pero buenos resultados) con la de cola (de reducido coste computacional pero generalmente resultados mucho peores), se ha incluye una estratgia basada en sampleo de un número determinado de pares (i, j) a evaluar, de tal manera que la insercción sea sobre un espacio de búsqueda más amplio pero a la vez se mantenga asequible sobre problemas de gran tamaño.

- `LocalSearchAlgorithm`: Para la implementación de técnicas de búsqueda local en la biblioteca `jinete`, se ha llevado a cabo la implementación de este algoritmo. Para que pueda ser utilizado, este espera recibir además de los parámetros de flota y conjunto de viajes, una solución inicial creada previamente por otro de los algoritmos. Por ejemplo, el `InsertionAlgorithm` indicado anteriormente. En cuanto a la salida esperada, este genera una nueva instancia `Result`. En este caso, la instancia se genera llevando a cabo distintas variaciones sobre la recibida como parámetro de entrada. Una vez definida la forma en que funciona el algoritmo de búsqueda local, lo siguiente es comentar cuáles son las transformaciones que este puede llevar a cabo. Estas se definen como estrategias de búsqueda local las cuales se han implementado, al igual que otras partes de la biblioteca, mediante una intefaz que define la forma de uso, así como subimplementaciones que incluyen la lógica correspondiente. A continuación se lleva a cabo una descripción de estas:

  - `LocalSearchStrategy`: Se corresponde con la interfaz que define cómo es posible utilizarlas. En este caso, el parámetro de entrada es una entidad `Result`, que a través de la implementación del método `improve() -> Result` describe la forma en que se pueden crear distintas estrategias de búsqueda local.
  - `OneShiftLocalSearchStrategy`: Se trata de una estrategia de búsqueda local que se aplica a nivel de cada ruta. El modo de funcionamiento es el siguiente: trata de mover cada parada 1 posición hacia atrás respecto de la secuencia de paradas y hace el cambio permanente cuando se llega a una reducción de coste.
  - `TwoOPTLocalSearchStrategy`: Consiste en la implementación del método *2-OPT* de búsqueda local, que como es natural, se implementa a nivel de ruta. Tal y como se ha indicado en el apartado 3.3.6, esta trata de "deshacer nudos" en la ruta mediante inversión del orden de paradas del segmento (i, j) donde $1 \leq i < j \leq m$ y m representa

el número total de paradas de la ruta. Sin embargo, esta estrategia no proporciona resultados muy beneficiosos en problemas como el *Dial-a-Ride*.

- `ReallocationLocalSearchStrategy`: Se trata de una estrategia de búsqueda local entre rutas, es decir, a nivel de planificación. Esta estrategia consiste en elegir un viaje asignado a un determinado vehículo y tratar de añadirlo en la ruta de otro. El cambio se hace efectivo si hay una reducción de coste.

- `SequentialAlgorithm`: Se trata de una subimplementación que actúa como envoltorio para otras. Tal y como se ha definido la interfaz `Algorithm` así como la forma en que esta es utilizada por otras entidades, la cual consiste en una única llamada a la subimplementación correspondiente, esta no permite la concatenación de distintos métodos de resolución de manera nativa. Es por ello que se ha creado este emboltorio, cuyo objetivo es el de suministrar dicha funcionalidad. A modo de ejemplo, la definición de un algoritmo secuencial sencillo puede consistir en dos pasos: 1. la creación de una solución inicial a partir de `InsertionAlgorithm` y, la mejora de esta a partir de algún método de búsqueda local proporcionado por `LocalSearchAlgorithm`. Entonces, la manera de llevar esto a cabo en la biblioteca `jinete` es a partir de `SequentialAlgorithm`.

- `IterativeAlgorithm`: Al igual que `SequentialAlgorithm` proporciona la funcionalidad de poder concatenar distintos algoritmos proporcionando un envoltorio, la manera de proporcionar iteratividad es a partir de `IterativeAlgorithm`. En este caso, el término *iteratividad* se refiere a la ejecución repetida de un mismo algoritmo durante un número prefijado de veces, o una condición de parada dada, escogiendo la mejor solución en términos de coste de entre todas las obtenidas en cada ciclo de iteracción.

- `GRASPAlgorithm`: Consiste en la implementación básica de la metaheurística *GRASP*, la cual se describe en detalle en el apartado 3.4.1. Tal y como se puede apreciar, gracias a la implementación de todas las distintas "piezas" que componen esta metaheurística como componentes independientes que pueden ser compuestas entre sí, la implementación de la misma es extremadamente sencilla. En concreto, esta está formada por la combinación de una instancia `IterativeAlgorithm`, que repite la ejecución de un `SequentialAlgorithm` compuesto por los dos pasos anteriormente utilizados como ejemplo: `InsertionAlgorithm` y `LocalSearchAlgorithm`. Sin embargo, la ventaja de este enfoque de "construcción" de algoritmos no es únicamente la de permitir la implementación de la metaheurística *GRASP* de una manera sencilla, si no las capacidades de reutilización que sus partes tienen para poder implementar metaheurísticas mucho más avanzadas en próximos pasos del proyecto.

- `MilpAlgorithm`: Se trata de la implementación que se comunica con los solvers de resolución de problemas de programación mixta lineal y entera. Al igual que el resto de implementaciones, esta recibe como parámetros las entidades `Fleet` y `Job` que representan la instancia concreta del problema y se encarga de generar la formulación lineal del mismo para después

transmitirla al correspondiente solver externo. Para llevar a cabo el proceso de formulación, se apoya en la utilización del correspondiente modelo lineal:

- `LinearModel`: Define la interfaz definida para transformar las correspondientes entidades `Fleet` y `Job` en restricciones del problema a resolver, de tal manera que estas sean comprensibles por las dependencias externas que permiten su ejecución.
- `ThreeIndexLinearModel`: Se trata de la implementación de la formulación definida sobre un modelo de 3 índices para el problema *Dial-a-Ride*.

A lo largo del apartado se ha llevado a cabo una descripción acerca de los distintos componentes definidos sobre la interfaz `Algorithm` en la biblioteca `jinete`. A través de dicha descripción se ha tratado de justificar la utilidad obtenida por un mayor grado de esfuerzo en el diseño del módulo, lo cual permite aumentar el grado de reutilización de los componentes de este. Por lo tanto, una vez descrita la manera en que los problemas de optimización son resuelto, el siguiente paso es comentar cómo se lleva a cabo el proceso de almacenamiento de los resultados obtenidos, para lo cual se destina el próximo apartado.

4.2.2.5. Exportación de Resultados

Tal y como se ha comentado a lo largo del apartado anterior, el resultado de la ejecución de los métodos de resolución del problema, que toman como entrada las instancias `Surface`, `Fleet` y `Job` que definen el espacio geométrico, la flota de vehículos disponibles y el conjunto de viajes a realizar respectivamente y generan como salida una instancia de tipo `Result`, la cual contiene la correspondiente planificación (representada por una instancia `Planning` tal y como se indicó en el apartado dedicado al modelo de datos). Entonces, la funcionalidad que desempeña el módulo `storers`, en el cual se centra este apartado, consiste en el almacenamiento, visualización y/o exportación de los resultados de la optimización, dados como una instancia `Result`.

A pesar de la aparente redundancia entre las entidades `Result` y `Planning`, hacer la división entre ambas es una decisión relacionada con la necesidad de reducir la responsabilidad que tiene cada una de ellas en el modelo de datos. En concreto, la entidad `Planning` se encarga de actuar como un contenedor de rutas de vehículos (instancias `Route`), mientras que `Result` se encarga de "conectar" la definición de la instancia del problema, dada por la flota de vehículos (`Fleet`) y el conjunto de viajes (`Job`), con la realización concreta de una asignación (`Planning`) generada por el correspondiente método de resolución. Dicha conexión entre definición del problema y asignación obtenida es necesaria de manera conjunta, entre otras cosas, para las tareas de exportación de resultados.

Una vez llevada a cabo la diferenciación entre `Result` y `Planning`, el siguiente paso es describir el modo de funcionamiento del módulo `loaders` desde una perspectiva de alto nivel, para poste-

riormente aclarar los detalles más interesantes de este. En este caso, el enfoque seguido para la implementación ha sido similar al utilizado en los módulos `loaders` y `algorithms`, es decir, se define una interfaz que define el modo de utilización de las distintas implementaciones incluidas en el módulo para tratar de maximizar la reusabilidad entre sí.

La interfaz que define el modo de utilización del módulo se ha denominado `Storer`, que espera recibir como entrada una instancia `Result` e incluye un método `store()` que inicia la ejecución. Además, al igual que ocurría en el caso de la carga de datos, en este caso también se ha añadido el concepto de `StorerFormatter`, cuyo cometido es el de definir el formato de los datos de salida, sobre el cual se apoyan las correspondientes implementaciones para formatear los resultados. En la figura 4.5 se incluye una ilustración del diagrama de clases simplificado que representa el contenido del módulo.

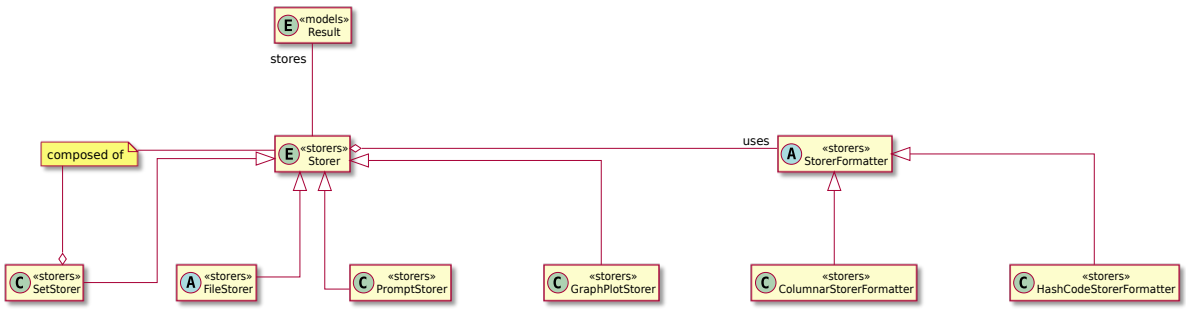


Figura 4.5: Diagrama de clases del módulo `storers` perteneciente a la biblioteca `jinete`.

En cuanto a las implementaciones concretas de estas interfaz, en este caso se han incluido las siguientes:

- `PromptStorer`: El comportamiento que desempeña es el de mostrar por la salida estándar (línea de comandos) el resultado a almacenar, muy útil en tareas de desarrollo y validación.
- `FileStorer`: Se encarga de almacenar los resultados generados por el correspondiente `StorerFormatter` en una fichero del sistema lo cual proporciona persistencia para su posterior a otros sistemas.
- `GraphPlotStorer`: Se encarga de generar una representación en forma de grafo de la solución generada, el cuál muestra en una ventana independiente del sistema. Es posible configurarlo de tal manera que dicha representación sea exportada a un fichero en caso de que el sistema donde se ejecute la biblioteca no tenga capacidades gráficas.
- `StorerSet`: La utilidad de este `Storer` consiste en actuar como contenedor de otros. Puesto que la manera en que se ha definido la interfaz de utilización del módulo no permite la asignación de más de un `Storer` para una misma solución en el `Dispacher` secuencial, entonces

es necesario crear una abstracción que permita obtener dicho comportamiento. En este caso, se ha llevado a cabo a través del `StorerSet`, que recibe como entrada un conjunto de implementaciones `Storer` y se encarga de propagar la llamada `store()` a todos ellos con el mismo `Result`. De esta manera es posible por ejemplo, mostrar una representación gráfica del resultado obtenido, a la vez que se almacena en un fichero el resultado formateado.

En cuanto a la manera de definir el formato de salida (cuando proceda), esto se lleva a cabo a través de la interfaz `StorerFormatter`, que se encarga de transformar una instancia `Result` en una cadena de caracteres. En la implementación actual se incluyen dos versiones:

- `ColumnarStorerFormatter`: Consiste en una versión en forma de columna que trata de incluir la mayor cantidad de información posible para poder entender el estado concreto de cada vehículo en cada punto concreto de la ejecución, muy útil sobre todo para tareas de análisis.
- `HashCodeStorerFormatter`: Consiste en un formato de salida muy simplificado y definido para una variación del problema *Dial-a-Ride* utilizada como problema a resolver en una de las ediciones del concurso *HashCode* llevado a cabo por la compañía *Google*.

Una vez finalizada la descripción acerca del modo de funcionamiento del módulo `storers` ya se puede tener una idea alto nivel acerca de cómo funciona la biblioteca implementada. En los futuros apartados se lleva a cabo un análisis acerca de los resultados obtenidos que demuestran la validez de la misma, así como su grado de eficiencia respecto de las soluciones óptimas obtenidas en trabajos de otros autores.

4.3. Resultados

Tras haber llevado a cabo una descripción acerca de la implementación realizada, la cuál tal y como se ha indicado, consiste en una biblioteca desarrollada sobre el lenguaje `Python` cuyo nombre es `jinete`, el siguiente paso es comentar el procedimiento llevado a cabo para evaluar la validez y el correcto funcionamiento de la misma. Además, se espera que este apartado sirva como guía de uso de esta, incluyendo en todo momento las sentencias utilizadas tanto para iniciar la ejecución como para definir la configuración utilizada, de tal manera que el proceso sea totalmente reproducible si así se requiere.

El resto del apartado se organiza de la siguiente manera: en el apartado 4.3.1 se indica la manera en que se ha llevado a cabo el proceso de experimentación, indicando las instancias del problema *Dial-a-Ride* escogidas para ello, la configuración exacta utilizada para la ejecución de la biblioteca y otros detalles como el tiempo máximo de ejecución. Seguidamente, en el apartado 4.3.2 se incluyen las tablas que muestran los resultados obtenidos tras la experimentación. Finalmente,

en el apartado 4.3.3 se lleva a cabo un análisis sobre dichos resultados, tratando de identificar ciertos patrones comunes así como posibles puntos de mejora.

4.3.1. Definición del experimento

Tal y como se ha indicado previamente, el objetivo de este apartado consiste en especificar cómo se ha llevado a cabo el proceso de obtención de resultados para demostrar la validez de la implementación realizada. Para ello, en primer lugar se indican los conjuntos de instancias utilizadas. Seguidamente se incluyen los scripts utilizados para comenzar la ejecución y finalmente se comentan otros factores como el tiempo máximo de ejecución para cada una de ellas (el cual es un factor a tener en cuenta en este tipo de problemas por razones obvias).

Las instancias utilizadas para el proceso de benchmarking o validación de resultados provienen de dos de los trabajos más populares relativos al problema *Dial-a-Ride*. En primer lugar, se han utilizado las instancias creadas en [CL03], las cuales están compuestas por 20 casos generados aleatoriamente y tienen tamaños desde los 3 hasta los 10 vehículos para satisfacer desde 24 a 144 viajes. Estás se identifican de la siguiente manera: `R1a.txt`, ..., `R10a.txt`, `R1b.txt`, ...,`R10b.txt`. El segundo trabajo del cual se han extraido las instancias utilizadas para el experimento es [RCL07], compuesto por dos grupos de 21 casos cada uno, haciendo un total de 42. Estas también han sido generadas de manera aleatoria, con tamaños desde 3 vehículos y 16 viajes hasta 8 vehículos y 96 viajes. En este caso las instancias se identifican de la siguiente forma: `a2-16.txt`, ..., `a8-96.txt`, `b2-16.txt`, ...,`b8-96.txt`. El motivo por el cual se han escogido estos dos grupos de instancias frente a otros ha sido la popularidad que tienen dentro del nicho de estudio del problema *Dial-a-Ride*, habiendo sido utilizados en numerosos trabajos desde su publicación inicial.

En cuanto a cómo se ha llevado a cabo la ejecución desde la perspectiva de la utilización de la implementación llevada a cabo (`jinete`), en las figuras 4.6 y 4.7 se incluyen los fragmentos de código utilizados. Antes de empezar a comentar el objetivo de ambos, así como las sentencias más relevantes de cada uno de ellos, es necesario comentar los requisitos sobre el entorno de ejecución: En este sentido, el primer requisito es tener instalada una versión de `Python` mayor o igual a 3.7 para que todas las dependencias funcionen correctamente. En cuanto a las dependencias dentro del lenguaje, en este caso tan solo es necesario instalar la última versión de la biblioteca implementada, la cuál se puede instalar a partir del código fuente. Sin embargo, al haberse desarrollado como una biblioteca open source, es tan sencillo como ejecutar el siguiente comando:

```
pip3 install jinete
```

Una vez configurado el entorno de ejecución ya se está en condiciones suficientes como para poder llevar a cabo la ejecución por lo que se procede a comentar la función que desempeña cada uno de los scripts.

El fragmento de código contenido en el fichero `launch.py`, que se muestra en la figura 4.6, es el encargado de "orquestar" la ejecución del experimento. Es decir, es el encargo de llevar a cabo las ejecuciones concretas para cada instancia, encargandose de detectar cuando estas terminan para así lanzar la siguiente. Para hacer esto, se ha decidido que cada instancia sea ejecutada en un único proceso, permitiendo la ejecución de múltiples instancias al mismo tiempo si la máquina en cuestión es capaz de ello. En cuanto al contenido del fragmento de código, las líneas 15 a 17 son las encargadas de buscar los ficheros que contienen las definiciones de cada instancia y proceder a la ejecución en paralelo, las cuales son transmitidas a `ProcessPoolExecutor`, encargado de controlar el paralelismo actuando mediante un protocolo de cola definido como 1 productor, k consumidores donde k es el número de procesadores de la máquina. Cuando llega el turno pertinente, la función `run_one(file_path: Path)` es ejecutada, encargandose de llevar a cabo una llamada al comando `python3 solve.py $FILE_PATH`. El resto de las sentencias son las encargadas de controlar los posibles errores, así como guardar los posibles mensajes de `logging` generados durante la ejecución.

Una vez comentado el fragmento de código encargado de llevar a cabo el proceso de control y lanzamiento de todas las instancias incluidas en el experimento, lo siguiente es comentar el segundo fragmento de código: Este es quién lleva a cabo el proceso de optimización para cada instancia, es decir, es el que realiza la llamada que se encarga de iniciar la ejecución de la biblioteca `jinete`. Dicho fragmento de código se incluye en el fichero `solve.py`, cuyo contenido se muestra en la figura 4.7.

A continuación se procede a comentar las sentencias más destacadas del fragmento de código: En primer lugar, es necesario comentar la línea 7, que a pesar de su extremada sencillez, es la que da acceso a la implementación llevada a cabo. Lo siguiente a comentar son las líneas 16 a 38, donde se construye el objeto `jit.Solver`. Tal y como se comentó previamente, este actúa como un envoltorio que permite configurar de una manera relativamente sencilla los parámetros de ejecución. Lo primero es indicar al `loader`, la ruta del fichero que contiene la instancia. En este caso, no se indica que el problema sigue el formato definido por *Cordeau-Laporte* dado que este es el usado por defecto.

Seguidamente, se indica el algorithmo a utilizar, el cual, en este caso es `jit.GRASPAlgorithm`. Para entender mejor cómo funciona dicha metahuerística, se recomienda consultar apartado 3.4.1. Por defecto, este se configura utilizando un método de inserción (`jit.InsertionAlgorithm`) que utiliza una estrategia de muestreo de *huecos* (`jit.SamplingInsertionStrategy`), junto con un iterador ranking (`jit.RankingInsertionIterator`). El criterio utilizado es el de añadir aquel viaje que minimice el tiempo de finalización de la ruta (`jit.EarliestLastDepartureTimeRouteCriterion`). Respecto de la heurística de búsqueda local, la estrategia seguida consiste en un repetir de manera iterativa (`jit.IterativeAlgorithm`) la secuencia (`jit.SequentialAlgorithm`) de dos pasos basada en mover un viaje de vehículo (`jit.ReallocationLocalSearchStrategy`) y poste-

riormente tratar de añadir los viajes que todavía se hubiesen podido incluir en la planificación ((`jit.InsertionAlgorithm`)). Dicha operación se lleva a cabo mientras se hayan obtenido mejoras respecto de la iteración anterior. En cuanto a la configuración utilizada para la solución inicial en la metaheurística *GRASP*, se permite una única ejecución y la selección del siguiente viaje a añadir se lleva a cabo de manera aleatoria entre los dos mejores candidatos. En cuanto al número de soluciones que se generan con la metaheurística, se permiten un máximo de 5, para posteriormente seleccionar la mejor de todas ellas.

```python
import logging
import traceback
from concurrent import futures
from pathlib import Path
from subprocess import check_output, TimeoutExpired, STDOUT

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

DATASETS_PATH = Path(__file__).absolute().parent
INSTANCE_TIMEOUT = 7200


def main():
    with futures.ProcessPoolExecutor() as executor:
        for file_path in DATASETS_PATH.glob("*.txt"):
            executor.submit(run_one, file_path)


def run_one(file_path):
    try:
        command = ["python3", "solve.py", str(file_path.absolute())]
        print(f'COMMAND: "{" ".join(command)}"')
        try:
            output = check_output(
                command,
                timeout=INSTANCE_TIMEOUT,
                stderr=STDOUT,
                cwd=Path(__file__).parent
            ).decode()
        except TimeoutExpired as exc:
            print(exc)
            output = exc.output.decode()
        with (file_path.parent / f"{file_path.name}.log").open("w") as file:
            file.write(output)
    except Exception as exc:
        traceback.print_exc()
        raise exc


if __name__ == '__main__':
    main()
```

Figura 4.6: Fragmento de código contenido en el fichero `launch.py`.

Por último, se incluye la configuración del `storer`. En este caso se utiliza un `jit.SetStorer` puesto que se quiere almacenar la solución con varias implementaciones, las cuales generan la solución como: texto plano en la línea de comando (`jit.PromptStorer`), un fichero externo

```python
import logging
import sys
import traceback
from functools import partial
from pathlib import Path

import jinete as jit

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def main():
    logger.info("Starting...")
    file_path = Path(sys.argv[1])
    solver = jit.Solver(
        loader_kwargs={"file_path": file_path},
        algorithm=jit.GraspAlgorithm,
        algorithm_kwargs={
            "first_solution_kwargs": {
                "episodes": 1,
                "randomized_size": 2,
            },
            "episodes": 5,
        },
        storer=jit.StorerSet,
        storer_kwargs={
            "storer_cls_set": {
                jit.PromptStorer,
                partial(
                    jit.GraphPlotStorer, file_path=file_path.parent / f"{file_path.name}.png"
                ),
                partial(
                    jit.FileStorer, file_path=file_path.parent / f"{file_path.name}.output"
                ),
            }
        },
    )
    result = solver.solve()  # noqa
    logger.info("Finished...")


if __name__ == "__main__":
    try:
        main()
    except Exception as exc:
        traceback.print_exc()
        raise exc
```

Figura 4.7: Fragmento de código contenido en el fichero `solve.py`.

(`jit.FileStorer`) y una imagen del grafo donde las posiciones se representan como nodos que se conectan entre si por las aristas generadas por las correspondientes rutas (`jit.GraphPlotStorer`).

En cuanto a las restricciones de ejecución, necesarias en este tipo de problemas donde la explosión combinatoria hace impractible la exploración de una gran cantidad del espacio de búsqueda. En este caso se ha decidido dedicar **2 horas de cómputo por cada instancia** del problema lo cual es un factor a tener en cuenta en la comparación de los resultados obtenidos, sobre todo en los resultados obtenidos en instancias de gran tamaño. Respecto del sistema utilizado para llevar a cabo el experimento, se ha utilizado una máquina con **16GB de RAM** y un procesador **Intel I7-3520M de 2 núcleos y 2.9GHz**.

A lo largo del apartado se han comentado tanto la procedencia de las instancias utilizadas para el proceso de experimentación, como la configuración de la biblioteca `jinete` escogida para llevar a cabo la resolución de las mismas, como las restricciones fijadas a nivel de la capacidad de cómputo destinada para la resolución de cada una de las instancias. Una vez comentados todos estos puntos, ya se está en condiciones suficientes para presentar los resultados obtenidos, los cuales se incluyen en el siguiente apartado, para posteriormente realizar un análisis sobre estos en el apartado 4.3.3.

4.3.2. Tablas de resultados

El objetivo de este apartado consiste únicamente en agrupar las tablas de resultados obtenidas. Tal y como se ha indicado previamente, el experimento llevado a cabo se ha basado en la ejecución de 3 grupos de instancias extraidas de [CL03; RCL07], los cuales contienen importantes aportaciones relativas al problema *Dial-a-Ride*. En este punto es importante remarcar que los resultados se han obtenido tras aplicar una metaheurística *GRASP*. Sin embargo, también se ha tratado de aplicar la resolución basada en la formulación de 3 índices como problema de *programación entera*. Sin embargo, tan solo se han obtenido resultados para las instancias de menor tamaño (16 viajes) por lo que se ha decidido no incluir en los resultados.

En cuanto al contenido, para mantener el orden del documento se ha decidido presentar los resultados en una tabla por cada grupo de problemas, es decir, las cuadros 4.1 a 4.3 se refieren a los grupos de problemas R, A y B respectivamente. Todas ellas siguen el mismo formato, el cual se comenta a continuación:

- **Id.**: Identificador de la instancia del problema.
- **Vehículos**: Número de vehículos disponibles.
- **Viajes**: Número de viajes a satisfacer.
- **Óptimo**: Mejor valor conocido.

- **Obtenido**: Valor obtenido tras 2 horas de ejecución.
- **Dif.**: Diferencia porcentual entre el mejor valor conocido y el obtenido.
- **Servicio**: Nivel de servicio alcanzado.
- **Timeout**: Valor booleano indicando si la ejecución ha termiando antes de las 2 horas o, por el contrario, ha terminado debido a haberlas superado.

Una vez descritas las columnas, a continuación se incluyen las tablas de resultados, las cuales son analizadas posteriormente.

Id.	Vehículos	Viajes	Óptimo	Obtenido	Dif.	Servicio	Timeout
R1a	3	24	190.02	-	-	-	-
R2a	5	48	301.34	-	-	-	-
R3a	7	72	532		-	-	-
R4a	9	96	626.93	-	-	-	-
R5a	11	120	570.25	-	-	-	-
R6a	13	144	785.26	-	-	-	-
R7a	4	36	291.71	-	-	-	-
R8a	6	72	487.84	-	-	-	-
R9a	8	108	658.31	-	-	-	-
R10a	10	144	851.82	-	-	-	-
R1b	3	24	164.46	-	-	-	-
R2b	5	48	295.66	-	-	-	-
R3b	7	72	484.83	-	-	-	-
R4b	9	96	529.33	-	-	-	-
R5b	11	120	577.29	-	-	-	-
R6b	13	144	730.67	-	-	-	-
R7b	4	36	248.21	-	-	-	-
R8b	6	72	458.73	-	-	-	-
R9b	8	108	593.49	-	-	-	-
R10b	10	144	785.68	-	-	-	-

Tabla 4.1: Resultados obtenidos tras 2 horas de cómputo mediante la metahurística *GRASP* de las instancias del grupo *R*.

Id.	Vehículos	Viajes	Óptimo	Obtenido	Dif.	Servicio	Timeout
a2-16	2	16	294.25	294.25	0 %	100 %	No
a2-20	2	20	344.83	-	-	-	-
a2-24	2	24	431.12	-	-	-	-
a3-24	3	24	344.83	-	-	-	-
a3-30	3	30	494.85	-	-	-	-
a3-36	3	36	583.19	-	-	-	-
a4-32	4	32	485.5	-	-	-	-
a4-40	4	40	557.69	-	-	-	-
a4-48	4	48	668.82	-	-	-	-
a5-40	5	40	498.41	-	-	-	-
a5-50	5	50	686.62	-	-	-	-
a5-60	5	60	808.42	-	-	-	-
a6-48	6	48	604.12	-	-	-	-
a6-60	6	60	819.25	-	-	-	-
a6-72	6	72	916.05	-	-	-	-
a7-56	7	56	724.04	-	-	-	-
a7-70	7	70	889.12	-	-	-	-
a7-84	7	84	1033.37	-	-	-	-
a8-64	8	64	747.46	-	-	-	-
a8-80	8	80	945.73	-	-	-	-
a8-96	8	96	1229.7	-	-	-	-

Tabla 4.2: Resultados obtenidos tras 2 horas de cómputo mediante la metahurística *GRASP* de las instancias del grupo *A*.

Id.	Vehículos	Viajes	Óptimo	Obtenido	Dif.	Servicio	Timeout
b2-16	2	16	309.41	-	-	-	-
b2-20	2	20	332.64	-	-	-	-
b2-24	2	24	444.71	-	-	-	-
b3-24	3	24	394.51	-	-	-	-
b3-30	3	30	531.44	-	-	-	-
b3-36	3	36	603.79	-	-	-	-
b4-32	4	32	494.82	-	-	-	-
b4-40	4	40	656.63	-	-	-	-
b4-48	4	48	673.81	-	-	-	-
b5-40	5	40	613.72	-	-	-	-
b5-50	5	50	761.4	-	-	-	-
b5-60	5	60	902.04	-	-	-	-
b6-48	6	48	714.83	-	-	-	-
b6-60	6	60	860.07	-	-	-	-
b6-72	6	72	978.47	-	-	-	-
b7-56	7	56	823.97	-	-	-	-
b7-70	7	70	912.62	-	-	-	-
b7-84	7	84	1203.37	-	-	-	-
b8-64	8	64	839.89	-	-	-	-
b8-80	8	80	1036.34	-	-	-	-
b8-96	8	96	1185.55	-	-	-	-

Tabla 4.3: Resultados obtenidos tras 2 horas de cómputo mediante la metahurística *GRASP* de las instancias del grupo *B*.

4.3.3. Análisis de resultados

[TODO]

[TODO]

[TODO]

[TODO]

[TODO]

[TODO]

[TODO]

4.4. Conclusiones

La implementación de estrategias de resolución para *problemas de optimización combinatoria* representa es una de las áreas englobadas en la *Investigación Operativa* que más cercana se encontra con las dificultades reales derivadas de resolver este tipo de problemas. Sin embargo, a su vez representa un área muy interesante, tanto desde el punto de vista de la necesidad de conocer en detalle tanto los problemas que se pretenden resolver como los métodos de resolución más populares. Además de ello, es necesario tener en cuenta todas las limitaciones que surgen al tratar de implementar dichos métodos de tal manera que estos pasen de ser ideas teóricas a materializarse como instrucciones en un sistema real.

A lo largo de este capítulo se ha tratado de poner en práctica todos aquellos conocimientos teóricos llevando a cabo la implementación de una biblioteca escrita en el lenguaje `Python` y bautizada como `jinete`, la cuál se ha tratado de llevar a cabo tratando de maximizar su mantenibilidad futura de tal manera que sea sencillo continuar con un constante proceso de mejora en el futuro. Entre otras cosas, en el correspondiente apartado del capítulo, se ha tratado de transmitir de la manera más clara posible por qué componentes está formada, así como las ideas más relevantes en las cuales se apoya. Para simplificar el razonamiento, dicha explicación se ha divido en dos partes: una de alto nivel donde se han comentado decisiones desde una perspectiva externa y otra de bajo nivel donde se han discutido decisiones más cercanas al proceso de implementación.

En cuanto a los resultados obtenidos, es fácil darse cuenta de que la implementación realizada aún no es lo suficientemente madura como para poder enfrentarse a problemas de una envergadura real, tanto por la necesidad de dedicar tiempo en mejorar ciertos aspectos que reduzcan el tiempo de cómputo de algunas partes críticas, como desde la perspectiva de las metaheurísticas implementadas hasta el momento (a pesar de que *GRASP* proporciona unos resultados muy favorables, actualmente existen técnicas mucho más avanzadas que con los mismos recursos computacionales son capaces de proporcionar resultados mejores).

A pesar de todo lo comentado, el proceso de implementación de soluciones para este tipo de problemas es una tarea muy enriquecedora que ayuda a entender mejor las dificultades reales de las cuales hablan los autores más relevantes en sus trabajos. Además, ha servido como un buen punto de partida para la biblioteca `jinete` que quizás, en un futuro, continúe ampliandose.

Capítulo 5

Conclusiones Generales y Próximos Pasos

5.1. Conclusiones Generales

[TODO]

5.2. Próximos Pasos

[TODO]

Bibliografía

[BA03] Barrie M Baker y MA Ayechew. «A genetic algorithm for the vehicle routing problem». En: *Computers & Operations Research* 30.5 (2003), págs. 787-800.

[Bel+16] Irwan Bello y col. «Neural combinatorial optimization with reinforcement learning». En: *arXiv preprint arXiv:1611.09940* (2016).

[Bel54] Richard Bellman. *The theory of dynamic programming*. Inf. téc. Rand corp santa monica ca, 1954.

[BLS13] Ilhem Boussaid, Julien Lepagnot y Patrick Siarry. «A survey on optimization metaheuristics». En: *Information sciences* 237 (2013), págs. 82-117.

[BM04] John E Bell y Patrick R McMullen. «Ant colony optimization techniques for the vehicle routing problem». En: *Advanced engineering informatics* 18.1 (2004), págs. 41-48.

[BPS03] J Christopher Beck, Patrick Prosser y Evgeny Selensky. «Vehicle routing and job shop scheduling: What's the difference?» En: *ICAPS*. 2003, págs. 267-276.

[BR03] Christian Blum y Andrea Roli. «Metaheuristics in combinatorial optimization: Overview and conceptual comparison». En: *ACM computing surveys (CSUR)* 35.3 (2003), págs. 268-308.

[CL03] Jean-François Cordeau y Gilbert Laporte. «A tabu search heuristic for the static multi-vehicle dial-a-ride problem». En: *Transportation Research Part B: Methodological* 37.6 (2003), págs. 579-594.

[CL07] Jean-François Cordeau y Gilbert Laporte. «The dial-a-ride problem: models and algorithms». En: *Annals of operations research* 153.1 (2007), págs. 29-46.

[CLM01] Jean-François Cordeau, Gilbert Laporte y Anne Mercier. «A unified tabu search heuristic for vehicle routing problems with time windows». En: *Journal of the Operational research society* 52.8 (2001), págs. 928-936.

[Cor+09] Thomas H Cormen y col. *Introduction to algorithms*. MIT press, 2009.

[Cor06] Jean-François Cordeau. «A branch-and-cut algorithm for the dial-a-ride problem». En: *Operations Research* 54.3 (2006), págs. 573-586.

[CW64] Geoff Clarke y John W Wright. «Scheduling of vehicles from a central depot to a number of delivery points». En: *Operations research* 12.4 (1964), págs. 568-581.

[DL91] Martin Desrochers y Gilbert Laporte. «Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints». En: *Operations Research Letters* 10.1 (1991), págs. 27-36.

[GHL94] Michel Gendreau, Alain Hertz y Gilbert Laporte. «A tabu search heuristic for the vehicle routing problem». En: *Management science* 40.10 (1994), págs. 1276-1290.

[GK06] Fred W Glover y Gary A Kochenberger. *Handbook of metaheuristics*. Vol. 57. Springer Science & Business Media, 2006.

[GO03] Hassan Ghaziri e Ibrahim H Osman. «A neural network algorithm for the traveling salesman problem with backhauls». En: *Computers & Industrial Engineering* 44.2 (2003), págs. 267-281.

[Ho+18] Sin C Ho y col. «A survey of dial-a-ride problems: Literature review and recent developments». En: *Transportation Research Part B: Methodological* 111 (2018), págs. 395-421.

[JLB07] Rene Munch Jorgensen, Jesper Larsen y Kristin Berg Bergvinsdottir. «Solving the dial-a-ride problem using genetic algorithms». En: *Journal of the operational research society* 58.10 (2007), págs. 1321-1331.

[KM70] Victor Klee y George J Minty. *How good is the simplex algorithm*. Inf. téc. WASHINGTON UNIV SEATTLE DEPT OF MATHEMATICS, 1970.

[LJX04] Kwong-Sak Leung, Hui-Dong Jin y Zong-Ben Xu. «An expanding self-organizing neural network for the traveling salesman problem». En: *Neurocomputing* 62 (2004), págs. 267-292.

[MC09] Thiago AS Masutti y Leandro N de Castro. «A self-organizing neural network using ideas from the immune system to solve the traveling salesman problem». En: *Information Sciences* 179.10 (2009), págs. 1454-1468.

[MOD11] Stuart Mitchell, Michael OSullivan y Iain Dunning. «PuLP: a linear programming toolkit for python». En: *The University of Auckland, Auckland, New Zealand* (2011).

[MTZ60] Clair E Miller, Albert W Tucker y Richard A Zemlin. «Integer programming formulation of traveling salesman problems». En: *Journal of the ACM (JACM)* 7.4 (1960), págs. 326-329.

[PDH08] Sophie N Parragh, Karl F Doerner y Richard F Hartl. «A survey on pickup and delivery problems». En: *Journal für Betriebswirtschaft* 58.1 (2008), págs. 21-51.

[Pot93] Jean-Yves Potvin. «State-of-the-art survey—the traveling salesman problem: A neural network perspective». En: *ORSA Journal on Computing* 5.4 (1993), págs. 328-348.

[RCL07] Stefan Ropke, Jean-François Cordeau y Gilbert Laporte. «Models and branch-and-cut algorithms for pickup and delivery problems with time windows». En: *Networks: An International Journal* 49.4 (2007), págs. 258-272.

[Ros95] Guido Rossum. «Python reference manual». En: (1995).

[RR16] Mauricio GC Resende y Celso C Ribeiro. *Optimization by GRASP*. Springer, 2016.

[TV02] Paolo Toth y Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[WN14] Laurence A Wolsey y George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.