Algoritmos de Búsqueda Tabú
Aplicación en un problema de rutas

Abel Naya Forcano

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Directores del trabajo:
Herminia I. Calvete
Ángel R. Francés

Julio de 2016

Introducción

En su forma genérica, los problemas de rutas de vehículos consisten en hallar una ruta o rutas para una flota de vehículos para dar servicio a un conjunto de clientes minimizando, generalmente, el coste del transporte. Sus principales características son: la red de transporte, los arcos y nodos que los vehículos pueden recorrer, la flota de vehículos, las características de los vehículos utilizados en el proceso, así como las restricciones que puedan imponerse. Estas restricciones pueden afectar a la distancia recorrida o la carga máxima que pueden transportar los vehículos; al lugar en donde entregar la mercancía si cada cliente tiene varios posibles; el origen, o los orígenes si se permite que haya varios; el servicio, si se permite que la carga se pueda dividir o si existen horarios que cumplir; y las rutas solución, por ejemplo si éstas no deben superar una longitud máxima o si se permite que un vehículo realice varias de ellas. Este tipo de problemas de rutas de vehículos tiene multitud de variantes y modificaciones, desde las más sencillas hasta algunas que hoy en día siguen siendo materia de investigación. El coste computacional para resolver este tipo de problemas es elevado y, aunque existen métodos y algoritmos exactos, para problemas con un elevado número de nodos el tiempo de cálculo suele ser excesivo. Es por esto que los algoritmos metaheurísticos, que exploran el espacio de soluciones mediante algún método de búsqueda, se han utilizado para resolver este tipo de problemas, a menudo con resultados altamente satisfactorios.

Uno de estos procedimientos metaheurísticos son los llamados algoritmos de búsqueda tabú. Estos algoritmos tratan de guiar un proceso de búsqueda local mediante la utilización de estructuras de memoria, que almacenan determinados acontecimientos ocurridos a lo largo del proceso. La búsqueda local se basa en el concepto de vecinos de una solución, y va recorriendo el espacio de soluciones partiendo de una solución inicial y sustituyéndola por uno de sus vecinos iterativamente. Las estructuras de búsqueda tabú, en su forma mas básica, penalizan determinados vecinos para evitar que el proceso se estanque en un mínimo local.

El objetivo de este trabajo es estudiar el problema de rutas de vehículos y los algoritmos de búsqueda tabú, así como implementar un algoritmo de este tipo para resolver un problema clásico de rutas de vehículos. Esta memoria consta de tres capítulos. En el primer capítulo, se formula el problema clásico de rutas de vehículos y se presentan algunas variantes así como distintos tipos de algoritmos que se han propuesto en la literatura para su resolución. En el segundo capítulo, se presentan los conceptos fundamentales de los algoritmos de búsqueda tabú, así como algunas características más avanzadas. Por último, en el tercer capítulo, se detalla la implementación realizada del algoritmo, así como los resultados obtenidos tras ejecutarlo.

Summary

The aim of this proyect is to study the vehicle routing problem, to show which characteristics are usually taken into account to formulate it, and to present some algorithms which have been proposed in the literature to solve it. Then, we focus on tabu search algorithms and present a custom implementation of a tabu search algorithm designed to solve the capacitated vehicle routing problem.

Vehicle routing problems are combinatorial optimization and integer programming problems that consist on finding an optimal set of routes for a fleet of vehicles to traverse in order a group of customers. It was first introduced by Dantzig y Ramser (1959) and used to solve a petrol delivery problem. There exist a lot of variants, and they have many applications on industry.

One of the most studied and simple variant is the capacitated vehicle routing problem. It consists on the distribution of goods from a single depot to a set of customers. The fleet is assumed to be homogeneous, meaning that all the vehicles have the same capacity and operate at identical costs. Each vehicle starts at the depot, visits some customers and returns to the depot ending the route. When travelling between nodes, the vehicle incurs the travel cost of the arc between them. The objective is to minimize the travel costs of all the routes visiting all the customers and without exceeding the vehicle capacity.

In this work we present five formulations of the capacitated vehicle routing problem: the two-index formulation for directed graphs and for undirected graphs, a three-index formulation, an extensive formulation based on a set covering model and the capacity indexed formulation.

Some of the variants of the vehicle routing problems take into account network characteristics, for example if the customers need to be served on arcs instead of nodes or even both; route constraints, like limiting route length; letting a vehicle travel multiple routes or adding time window constraints. Also the fleet characteristics can be considered, sometimes the fleet is heterogeneous and vehicles have different capacity, costs or even depots. In general, the objective is to minimize route costs, but in some cases it is also needed to optimize the time, the length or the number of vehicles, and others.

There also exist different methods to solve these problems, and they are separated in two main groups, exact and heuristics algorithms, but there are also methods which combine both of them. Exact algorithms find the optimal solution testing implicitly or explicitly all the solutions. Heuristics try to find a good solution searching the topology of the space of solutions. Heuristic approaches cannot ensure that the best solution found is the optimal solution, but they are a good alternative when considering a high number of restrictions, because exact approaches often take a lot of computing time.

Some important exact algorithms are branch and bound, that incorporate relaxations and recursively split the search space removing the groups of solutions that cannot improve the current best solution; approaches based on column generation, where only some routes are considered and improved; branch and cut algorithms, an improvement to branch and bound procedures where restrictions are introduced to cut the search space; and branch and cut and price, which combines branch and cut with column generation. On the other hand, some important heuristics are local search, that travels the search space moving from one solution to another at each iteration like simulated annealing or tabu search; and population based heuristics, inspired by nature like ant colony optimization or genetic algorithms.

Tabu search heuristic is an approach used to guide a local search trying to avoid local optimums using memory and keeping track of events occurred in the past. These algorithms were first proposed by Glover (1977) and have been used to solve problems from many different areas like scheduling, logistics, planning, etc.

Local search heuristics travel the search space from the current solution to a near one. The near solutions are called neighbours. In general, the near solution results of modifying some of the current solution characteristics or variables. The best of these neighbours is set as the current solution, and the process is repeated until a stopping criterion is satisfied. During the execution of the algorithm, the best solution found is kept and returned when finalizing. Main basic tabu search mechanisms consist on keeping some properties of the solutions visited, or the movements that lead to these solutions, called tabu restrictions. When searching for the best neighbour, the solutions that match some of the tabu characteristics, are marked as tabu and they are not chosen unless an aspiration criterion removes the tabu status. The tabu restrictions are kept on a tabu structure for a limited number of iterations, then they are removed. This is often called short term memory. Some advanced tabu search mechanisms consist on keeping characteristics on a different structure without removing them. This is called long term memory. This information shows the characteristics or movements most used, and can be used to bias the search. Penalizing these movements will encourage the method to search unexplored regions; whether encouraging them will focus the search on the current area, these are intensification rules.

The algorithm taburoute, proposed by Gendreau *et al.* (1994), is a tabu search heuristic for capacitated vehicle routing problems on directed graphs. This algorithm uses basic and advanced features of tabu search. It starts generating some random solutions in such a way that they are separated on the search space. On each of these solutions a tabu search process is executed. Then, from the best solution found in all of them, the algorithm is run again with different parameters chosen to improve as much as possible the current solution. Finally, the algorithm is run one more time with intensification parameters, in order to ensure that we did not miss any good near solution. The search algorithm implements tabu search methods, checking all neighbours that are obtained by moving a vertex from its current route to another one. Removing and adding vertex to routes is performed using procedures from Gendreau *et al.* (1992) called Stringing and Unstringing, where instead of removing the vertex joining the open paths and adding it between two consecutive vertices a more efficient approach is used: it checks some different ways of modifying the route reorganising the vertices in the process, and the best of them is executed. This paper also describes other algorithms called Us, which optimizes a route removing and adding all vertices iteratively; and Genius, which creates a full route adding vertices using Stringing and optimizing with Us.

We have developed a custom implementation of the algorithm in Java using 23 different classes. Taburoute which contains the main algorithms. Genius which implements the algorithm with the same name. EstructuraTabu which contains the tabu structure used by the Search algorithm. Grafo which contains the data of the graph and the problem. Solucion which consists of a set of routes. Ruta which consists of a sequence of vertices and implements the Stringing and Unstringing procedures. And SolucionProxy and RutaProxy (and all subclasses) that represent the modifications of a solution and a route, respectively, and extends the SolucionAbstracta and RutaAbstracta classes. Proxi classes were chosen so that, instead of creating copies of the neighbour, the result of the modifications made to get the neighbour are evaluated, without actually generating the neighbour, and then the one chosen modifies the current solution.

Finally, the algorithm has been run using the same fourteen test problems from Christofides *et al.* (1979) used by the original article and the obtained results are shown. The code is also annexed for inspection.

Índice general

Capítulo 1

Problemas de rutas de vehículos

Los problemas de rutas de vehículos, *Vehicle Routing Problems* (VRP), engloban una familia de problemas de optimización que, genéricamente, tratan de encontrar las rutas que deben seguir un conjunto de vehículos desde un almacén central para visitar un cierto número de clientes con el menor coste posible. En particular, se trata de decidir qué vehículos recorren qué clientes y en qué orden, de forma de que todas las rutas efectuadas sean viables y el coste sea mínimo.

Esta descripción del problema es muy genérica, existiendo una gran cantidad de variantes. Aun así todos estos problemas comparten unas características similares, como la dificultad de encontrar una solución exacta en un tiempo razonable (no así soluciones aproximadas, que suelen requerir mucho menos tiempo) al tratarse de problemas P-duros, y la utilidad de sus aplicaciones en el mundo real.

Han pasado casi 60 años desde que la primera formulación fue descrita en 1959 por Dantzig y Ramser (1959) para resolver un problema de suministro de gasolina a estaciones de servicio. Desde entonces, se siguen publicando artículos y métodos de resolución debido principalmente a las ya mencionadas complejidad y utilidad práctica.

Una buena referencia sobre los problemas de rutas de vehículos son los dos libros de Toth y Vigo (2001, 2014) y el artículo de Laporte (2009).

1.1. Problemas de rutas de vehículos con capacidad

El problema de rutas de vehículos con capacidad, *Capacitated Vehicle Routing Problem* (CVRP), constituye el modelo más estudiado de entre los VRP. Trata de distribuir una mercancía desde un almacén central o centro de distribución hasta un conjunto de clientes. Se caracteriza porque se toma en consideración la capacidad de los vehículos, que puede ser o no igual para todos ellos. Por tanto, la suma de las demandas de los clientes en una ruta no puede ser mayor que la capacidad del vehículo que la recorre.

Denotaremos con 0 el centro de distribución y con $N = \{1,2,...,n\}$ el conjunto de los nodos que representan a los clientes. Consideramos la red $G = (V,A)$ o $G = (V,E)$ donde $V = \{0\} \cup N = \{0,1,\ldots,n\}$ es el conjunto de nodos y A/E es el conjunto de arcos o ejes, según sea la red dirigida o no dirigida. En el primer caso $A = \{a = (i,j) \in V \times V : i \neq j\}$. En el segundo caso $E = \{e = (i,j) = (j,i) : i,j \in V, i \neq j\}$.

Cada nodo $i \in N$ tiene asociada una demanda $q_i \geqslant 0$. Para facilitar la formulación del modelo, asociaremos con el centro de distribución una demanda $q_0 = 0$. Denotaremos el conjunto de vehículos como K suponiendo una flota homogénea, cada uno de ellos con capacidad Q. Supondremos conocido el coste c_{ij} asociado a que un vehículo haga el trayecto desde el nodo i al nodo j. Si al menos un par de nodos tienen coste asimétrico ($c_{ij} \neq c_{ji}$) entonces es necesario usar una red dirigida. En caso contrario una red no dirigida es suficiente. Tanto en un caso como en el otro, el número de arcos es $O(n^2)$ ($|E| = n(n+1)/2$ y $|A| = n(n+1)$).

Con estas notaciones, el CVRP queda definido de manera única por una red dirigida $G = (V,A,c_{ij},q_i)$ o no dirigida $G = (V,E,c_{ij},q_i)$, junto con el tamaño $|K|$ de la flota de vehículos y la capacidad Q de cada vehículo.

Una *ruta* consiste en una secuencia ordenada de nodos $r = (i_0, i_1, ..., i_s, i_{s+1})$ cuyos arcos son $i_0 \to i_1, i_1 \to i_2, \ldots, i_s \to i_{s+1}$, con $i_0 = i_{s+1} = 0$. Esta ruta representa la visita de los s clientes $S = \{i_1, ..., i_s\} \subseteq N$, donde S es un agrupamiento de clientes que llamaremos clúster. Esta ruta r conlleva un coste $c(r) = \sum_{p=0}^{s} c_{i_p, i_{p+1}}$ y es factible si se cumple la restricción de capacidad $q(S) := \sum_{i \in S} q_i \leqslant Q$ y los nodos son distintos $i_j \neq i_h \forall 1 \leqslant j < h \leqslant s$. En este caso, se dice que S es un clúster factible.

Una solución del CVRP consiste en un conjunto de $|K|$ rutas, una para cada vehículo $k \in K$. Una solución se dice factible si todas las rutas $r_1, r_2, \ldots, r_{|K|}$ son factibles y los clústeres $S_1, ..., S_{|K|}$ forman una partición de N. Por tanto, para resolver el problema se precisa llevar a cabo dos tareas simultáneas: La partición del conjunto de clientes en clústeres factibles y la resolución de un problema del viajante (*Traveling Salesman Problem* (TSP), Applegate *et al.* (2011)) asociado a $\{0\} \cup S_k, \forall k \in K$.

1.2. Formulaciones

En este apartado vamos a presentar cuatro formulaciones matemáticas que se han propuesto a lo largo de los años para el CVRP. Cada una presenta características propias que la hacen idónea para determinados algoritmos de resolución. Mediante modificaciones de estas formulaciones se tratarán posteriormente distintas variantes del VRP.

1.2.1. Notaciones básicas

Sea $S \subseteq V$ un conjunto arbitrario de nodos. Para redes no dirigidas, definimos $\delta(S) = \{(i,j) = (j,i) : i \in S, j \notin S\}$ como el conjunto de arcos con exactamente uno de los nodos en S. Para redes dirigidas, se definen $\delta^-(S) = \{(i,j) : i \notin S, j \in S\}$ y $\delta^+(S) = \{(i,j) : i \in S, j \notin S\}$, los conjuntos de arcos de entrada a S y de salida de S, respectivamente. Por otra parte, $A(S) = \{(i,j) \in A : i, j \in S\}$ es el conjunto de los arcos que conectan nodos de S y $r(S)$ denota el mínimo número de vehículos necesarios para satisfacer la demanda de todos los clientes en S.

1.2.2. Formulación con dos índices

Para el modelo del CVRP en una red dirigida, denotado VRP1, se definen las variables binarias x_{ij} que toman valor 1 si un vehículo recorre el arco $(i,j) \in A$, y 0 en caso contrario. El problema puede plantearse como:

$$
\begin{aligned}
\min_{x_{ij}} \quad & \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{sujeto a} \quad & \\
& \sum_{j \in \delta^+(i)} x_{ij} = 1 && \forall i \in N \\
& \sum_{i \in \delta^-(j)} x_{ij} = 1 && \forall j \in N \\
& \sum_{j \in \delta^+(0)} x_{0j} = |K| && \\
& \sum_{(i,j) \in \delta^+(S)} x_{ij} \geqslant r(S) && \forall S \subseteq N,\ S \neq \emptyset \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in A
\end{aligned}
$$

Esta formulación fue introducida por primera vez por Laporte *et al.* (1986).

Denotando por x el vector de variables x_{ij} y definiendo $x(I) = \sum_{(i,j) \in I} x_{ij}$ donde I es un subconjunto arbitrario de arcos, el modelo anterior se puede formular de forma condensada como:

$$
\begin{aligned}
\min_{x} \quad & c^T x \\
\text{sujeto a} \quad & \\
& x(\delta^+(i)) = 1 && \forall i \in N \\
& x(\delta^-(j)) = 1 && \forall j \in N \\
& x(\delta^+(0)) = |K| && \\
& x(\delta^+(S)) \geq r(S) && \forall S \subseteq N,\ S \neq \emptyset \\
& x_a \in \{0,1\} && \forall a \in A
\end{aligned}
$$

donde, si $a = (i, j)$, $x_a = x_{ij}$. El modelo correspondiente para una red no dirigida, denotado VRP2, donde ahora las variables se denotan por $x_e = x_{ij}$, $e = (i, j) \in E$ es, en notación condensada:

$$
\begin{aligned}
\min_{x} \quad & c^T x \\
\text{sujeto a} \quad & \\
& x(\delta(i)) = 2 && \forall i \in N \\
& x(\delta(0)) = 2|K| && \\
& x(\delta(S)) \geqslant 2r(S) && \forall S \subseteq N,\ S \neq \emptyset \\
& x_e \in \{0,1,2\} && \forall e \in \delta(0) \\
& x_e \in \{0,1\} && \forall e \in E \backslash \delta(0)
\end{aligned}
$$

Esta formulación fue presentada por Laporte *et al.* (1985).

1.2.3. Formulación con tres índices

La siguiente formulación, en la que las variables tienen tres índices, permite plantear el problema en una red dirigida $G = (V,A)$ en donde el origen 0 se reemplaza por dos nodos o y d que representan el comienzo y el final de las rutas. De esta manera, $V := N \cup \{o,d\}$ y $A := (V \setminus \{d\}) \times (V \setminus \{o\})$.

Como su nombre indica, la formulación con tres índices tiene variables binarias de la forma x_{ijk} que modelan el movimiento de los vehículos sobre los arcos. En otras palabras, $x_{ijk} = 1$ si y solo si el vehículo $k \in K$ recorre el arco $(i, j) \in A$. Al igual que antes, se denota por x_k el vector de componentes x_{ijk} y $x_k(I) = \sum_{(i,j)\in I} x_{ijk}$. Se define también la variable y_{ik} que indica si el vehículo k visita el nodo $i \in V$. En este caso, u_{ij} representa la carga del vehículo k justo antes de llegar al nodo i. La formulación del problema, que denotamos VRP3, es:

$$
\begin{aligned}
\min_{x,y} \quad & \sum_{k\in K} c^T x_k \\
\text{sujeto a} \quad & \\
& \sum_{k\in K} y_{ik} = 1 && \forall i \in N \\
& x_k(\delta^+(i)) - x_k(\delta^-(i)) = \begin{cases} 1, & i = o \\ 0, & i \in N \end{cases} && \forall i \in V \setminus \{d\},\ k \in K \\
& y_{ik} = x_k(\delta^+(i)) && \forall i \in V \setminus \{d\},\ k \in K \\
& y_{dk} = x_k(\delta^-(d)) && \forall k \in K \\
& u_{ik} - u_{jk} + Q x_{ijk} \leqslant Q - q_j && \forall (i,j) \in A,\ k \in K \\
& q_i \leqslant u_{ik} \leqslant Q && \forall i \in V,\ k \in K \\
& x = (x_k) \in \{0,1\}^{K\times A} \\
& y = (y_k) \in \{0,1\}^{K\times V}
\end{aligned}
$$

Esta formulación, de manera ligeramente diferente, fue descrita por primera vez por Golden *et al.* (1977).

1.2.4. Formulación basada en rutas

Esta formulación fue propuesta inicialmente por Balinski y Quandt (1964), y está basada en un modelo de cubrimiento de conjuntos (Set covering model) cuya idea es que si las rutas factibles se conocen, para resolver el problema basta determinar qué rutas se han de seleccionar. Sea Ω el conjunto de todas las rutas factibles del problema. Cada ruta $r \in \Omega$ es de la forma $r = (i_0, i_1, \ldots, i_s, i_{s+1})$ con $i_0 = o$ e $i_{s+1} = d$. Su coste asociado es $c_r = \sum_{j=0}^{s} c_{i_j, i_j+1}$. Se define el coeficiente binario a_{ir} que vale 1 si y solo si el nodo $i \in N$ se visita en la ruta r. Finalmente, se define la variable binaria λ_r que toma valor 1 si la ruta r es seleccionada y 0 si no lo es, siendo λ el vector de variables λ_r, y se denota por $\mathbb{1}$ el vector de dimensión el número de rutas con todas sus componentes iguales a 1. La formulación del problema, denotada VRP4, es:

$$
\begin{aligned}
\min_{\lambda} \quad & c^T \lambda \\
\text{sujeto a} \quad & \\
& \sum_{r\in\Omega} a_{ir}\lambda_r = 1 \quad \forall i \in N \\
& \mathbb{1}^T \lambda = |K| \\
& \lambda \in \{0,1\}^{\Omega}
\end{aligned}
$$

1.2.5. Formulación con la capacidad como índice

Esta formulación fue propuesta para modelos en redes dirigidas por Pecin *et al.* (2014). Se define la variable binaria x_a^q que toma el valor 1 si el arco $a = (i, j)$ pertenece a una ruta cuya demanda total a partir del nodo j inclusive es exactamente q, y 0 en caso contrario. Los arcos que vuelven al origen deben tener $q = 0$.

La formulación, que denotamos VRP5, del problema es:

$$
\begin{aligned}
\min_{x} \quad & \sum_{a\in A} c_a \sum_{q=0}^{Q} x_a^q \\
\text{sujeto a} \quad & \sum_{a\in\delta^-(\{i\})} \sum_{q=1}^{Q} x_a^q = 1 && \forall i \in N \\
& \sum_{q=1}^{Q} \sum_{i\in N} x_{0i}^q = |K| \\
& \sum_{a\in\delta^-(\{i\})} x_a^q - \sum_{a\in\delta^+(\{i\})} x_a^{q-q_i} = 0 && \forall i \in N,\ q = q_i,\ldots,Q \\
& x_a^q \in \{0,1\} && \forall a \in A,\ q = 1,\ldots,Q \\
& x_{(i,0)}^q = 0 && \forall i \in N,\ q = 1,\ldots,Q
\end{aligned}
$$

1.3. Variantes de problema

Los problemas de rutas de vehículos presentan, por sus aplicaciones prácticas, muchas variantes de distinto tipo, algunas de las cuales se enumeran a continuación.

1.3.1. Características de la red

En el planteamiento general del VRP se considera que los lugares en los que el producto ha de ser entregado están asociados a los nodos de la red existente. Existen, sin embargo, otras variantes en las que el servicio a realizar debe llevarse a cabo en los arcos de la red, los denominados *Arc Routing Problems* (ARP); o en nodos y arcos simultáneamente, los denominados *General Routing Problems* (GRP). Pueden citarse como ejemplos los problemas de rutas para vehículos de limpieza o inspección, el envío de correo postal y la recogida de basuras.

1.3.2. Requisitos de transporte

Envíos y recogidas. En general, se considera la distribución de mercancía desde un almacén central a unos clientes. Sin embargo, también se han considerado problemas de recogida, en los que la mercancía debe ser recogida en los nodos y llevada al almacén. Estos problemas son equivalentes, sin más que revertir las rutas.

Otras variantes combinan en las rutas la recogida y la entrega de mercancía simultáneamente. Si no se permiten reorganizaciones dentro de los vehículos, esto es, se debe entregar toda la mercancía antes de recoger la nueva, se denomina *VRP with Backhauls* (VRPB). Si por el contrario el vehículo permite la reorganización, el problema es un *Mixed VRPB* (MVRPB). En estos casos los clientes precisan entrega o recogida de mercancía, pero no ambas. Si se permite que los clientes recojan y entreguen mercancía y que además deba realizarse en una única visita, el problema pasa a denominarse *VRP with Simultaneous Pickup and Delivery* (VRPSPD). En caso de que se permita la carga/descarga en más de una visita, se denomina *VRP with Divisible Deliveries and Pickups* (VRPDDP).

Servicios alternativos. En esta variante del problema, un cliente puede tener varios lugares posibles de recogida. En los *Multi-Vehicle Covering Tour Problem* (MVCTP) quien proporciona el servicio elige uno de esos lugares para la entrega.

Transportes punto-a-punto. En este tipo de VRP, cada transporte consiste en el movimiento entre dos ubicaciones específicas de bienes *Pickup-and-Delivery Problem* (PDP) o personas *Dial-a-Ride Problem* (DARP)

Suministro reiterado. Esta variante modela situaciones en las que los clientes requieren los bienes cada cierto tiempo, pudiendo variar el día de entrega mientras no se queden sin stock, ya sea en días predefinidos: *Periodic VRP* (PVRP), o sin predefinir: *Inventory Routing Problem* (IRP).

Servicios divididos. En la variante *Split Delivery VRP* (SDVRP) la mercancía que un cliente concreto requiere no tiene por qué ser transportada por un único vehículo, por lo que cada demanda puede ser dividida en un número arbitrario de subdemandas, transportadas por diferentes vehículos.

Servicios compartidos y multimodales. En este caso la mercancía permanece indivisible, pero se transporta por diferentes vehículos utilizando puntos de transferencia intermedios, en donde se almacena temporalmente.

Rutas con beneficios y selección de servicios. A veces es imposible satisfacer todas las restricciones de transporte y algunas no pueden realizarse, en cuyo caso hay una penalización por no ser satisfechas. En el proceso de optimización, se debe elegir qué característica se pretende priorizar, normalmente la que mayor beneficio produzca. Si el servicio se centra en maximizar el beneficio obtenido (ingresos menos penalizaciones) se denomina *Profitable Tour Problem* (PTP). Si se impone como restricción necesaria una longitud máxima de las rutas, y el objetivo es maximizar el beneficio, se llama *Team Orienteering Problem* (TOP). Y si se debe obtener un beneficio mayor que cierta cantidad prefijada minimizando el coste de las rutas, entonces se denomina *Prize-Collecting VRP* (PCVRP).

Rutas dinámicas o estocásticas. En ocasiones existen condiciones del problema que no se pueden conocer previamente. En general, un problema se dice que es dinámico si se dispone de información relevante sobre las condiciones del sistema durante la operación y estocástico si pueden ser descritas por una función de probabilidad.

1.3.3. Restricciones de la ruta

Al formular el VRP se ha tenido en cuenta la restricción de capacidad de los vehículos sobre la ruta, pero existen otras muchas restricciones aplicables como el peso, espacio, volumen, indivisibilidad, rotación o apilado. También se consideran aquellas en las que los bienes se distribuyen de acuerdo con el orden de reparto, para evitar reorganizaciones en los vehículos.

Longitud de ruta. Limita la longitud total de una ruta, ya sea por consumo de recursos o por duración de ésta.

Usos múltiples de los vehículos. Generalmente se supone que cada vehículo realiza una única ruta en el problema. En los *VRP with Multiple use of vehicles* (VRPM) un mismo vehículo puede realizar varias rutas siempre que la suma de las duraciones de todas ellas satisfaga la restricción de duración y por tanto la solución sea factible.

Horarios. En los *VRP with Time Windows* (VRPTW) existen restricciones sobre el momento en el que comienza (o termina) el servicio a cada cliente.

1.3.4. Características de la flota

Aunque en general se supone que todos los vehículos son iguales y se comportan igual, en algunos casos se trabaja con vehículos con diferente capacidad, velocidad o coste. Estos problemas son los *Heterogeneous or mixed Fleet VRP* (HFVRP). En otros casos, los vehículos son homogéneos, pero tienen distintos orígenes, esto es, comienzan y terminan las rutas en diferentes lugares: *Multi(ple) Depot VRP* (MDVRP).

1.3.5. Restricciones globales

Además de las restricciones que afectan a cada ruta, consideradas en el apartado 1.3.3, se pueden considerar restricciones que afecten a cómo las rutas se combinan. Este es el caso de los *VRP with Multiple Synchronization constraints* (VRPMS), que permiten modelar, por ejemplo, si varios vehículos deben moverse a la vez por un recorrido común, si un destino debe ser visitado por varios vehículos en un orden prefijado, o si el consumo de recursos total en cada instante no debe sobrepasar un límite.

1.3.6. Objetivos

Aunque se ha definido el VRP como un problema en el que se minimiza el coste de las rutas, a veces se deben cumplir otro tipo de objetivos como, por ejemplo, minimizar el tiempo empleado, la duración del recorrido o el número de vehículos utilizados.

1.3.7. Otras extensiones

La consideración del VRP y otras actividades logísticas da lugar a problemas muy diferentes e interesantes. Por ejemplo, se puede considerar la consistencia de las rutas cuando el servicio se realiza repetida o regularmente. En los *Consistent VRP* (ConVRP) se requiere que el mismo conductor visite los mismos clientes en aproximadamente el mismo momento del día, lo que permite tener en cuenta la familiaridad de los conductores con la región y los clientes.

Todas estas variantes, y muchas otras más, constituyen la *Familia de los Problemas de Rutas de Vehículos*.

1.4. Métodos de resolución exactos

Los algoritmos exactos resuelven el problema hallando la mejor solución posible, generalmente comprobando todas las soluciones factibles, ya sea explícita o implícitamente. Esto hace que resulte prácticamente imposible la resolución en tiempo razonable cuando el problema tiene un gran número de nodos.

1.4.1. Algoritmos de ramificación y acotación

Los algoritmos de ramificación y acotación, *Branch-and-bound*, fueron los primeros algoritmos exactos efectivos para resolver el CVRP. A partir de las propuestas iniciales se han ido sofisticando hasta lograr un rendimiento alto. Estos algoritmos consisten en relajar el problema quitando o reduciendo alguna restricción, normalmente de integridad. La solución óptima de este problema relajado sirve de cota inferior (en caso de minimizar) de las soluciones del problema original. Si la solución no es factible para el problema original, se separa el problema en ramas restringiendo las relajaciones y se resuelven los problemas asociados a cada una de las ramas, generalmente empezando por aquella con menor coste. Debido a que la solución da una cota del valor de la función objetivo, si en una rama la cota no mejora alguna solución factible ya encontrada, esa rama se descarta (prune).

Los distintos algoritmos que se han ido proponiendo difieren, fundamentalmente, en la forma en la que se relaja el problema y en la cota que proporciona. Cuanto más ajustada sea la cota que se propone, más soluciones factibles dejarán de analizarse y, en consecuencia, antes se alcanzará la solución óptima.

Tipos de relajaciones. La relajación basada en asignación y emparejamiento consiste en convertir el problema en un *Problema de transporte* (TP) añadiendo $k-1$ copias del origen al grafo y asignándoles un coste entre ellas de $c'_{ij} = \gamma$, donde γ influencia el número de vehículos usados en la solución. Por ejemplo, $\gamma = \infty$ impone el uso de todos los vehículos y con $\gamma = 0$ se obtiene la mejor solución usando como mucho $|K|$ vehículos. Este problema se resuelve hallando un circuito de coste mínimo que pase por todos los nodos. Otra relajación consiste en imponer únicamente las condiciones de conectividad y hallar

un árbol recubridor mínimo imponiendo algunas condiciones de orden en los nodos. Estas restricciones tienen una baja calidad y solo permiten la solución óptima en pequeños problemas. Por ello también se suele realizar un enfoque Lagrangiano o aditivo, que consiste en comenzar imponiendo un subconjunto significativo de las restricciones y resolver el problema. Después se comprueban aquéllas que no se cumplen y se imponen, repitiendo el proceso hasta que todas se cumplan.

Ramificaciones. Una manera de ramificar un VRP consiste en separar en dos ramas imponiendo el uso de un arco determinado en una de ellas y en la otra no ($x_{ij} = 1$, $x_{ij} = 0$, respectivamente). Otra forma consiste en generar tantas ramas como nodos no visitados, imponiendo en cada una la inclusión del arco desde el último cliente visitado a uno de los clientes no visitados, y una rama extra excluyendo todos estos arcos.

1.4.2. Algoritmos basados en la formulación con rutas

Este tipo de algoritmos utiliza la formulación VRP4 presentada en 1.2.4 en donde se enumeraban todas las rutas posibles y se elegían cuáles formaban parte de la solución.

Esta formulación hace que el número de variables a utilizar sea muy grande. Por ello, se suele utilizar un procedimiento de *Generación de columnas* (CG) para resolver el problema. Se comienza resolviendo el problema con un subconjunto de rutas y se hallan las variables óptimas para ese subproblema. Posteriormente, se comprueba si existe una ruta mejor fuera del subconjunto de rutas considerado, mediante cálculos con el problema dual. Si existe tal ruta se añade al subconjunto y se repite el procedimiento. En caso contrario, la solución óptima del problema restringido es también la solución óptima del problema completo y se termina el algoritmo.

1.4.3. Algoritmos de ramificación y cortes

La estructura de este tipo de algoritmos es muy similar a la de los algoritmos de ramificación y acotación, pues a ella se le añade el procedimiento de cortes (cuts) que consiste en añadir restricciones al problema relajado que cortan partes de la región de factibilidad que con seguridad no contienen la solución óptima, o ayudan a su resolución (es un procedimiento muy usado en problemas generales de programación entera). Para este tipo de algoritmos se suele utilizar la formulación con dos índices VRP2, en donde existen variables binarias que indican si un arco es recorrido o no, la relajación entera que permite a las variables binarias tomar valores reales en el intervalo [0,1] y la ramificación en la que en una rama se impone visitar un arco y en la otra no.

Desigualdades válidas utilizadas. A lo largo de la historia de los algoritmos de ramificación y cortes se han introducido cortes o desigualdades válidas relacionadas con desigualdades que se han probado útiles en el problema del viajante. Se han considerado también cortes que tienen en cuenta una cota sobre el número mínimo de vehículos necesarios, o sobre la demanda de los clientes visitados, entre otros.

1.4.4. Algoritmos de ramificación, cortes y precios

Los algoritmos *Branch-and-cut-and-price* son la base de la mayoría de algoritmos más recientes, y se está probando que proporcionan los mejores resultados. Combinan la *Generación de columnas* (CG) con los algoritmos de ramificación y cortes (branch-and-cut). Se basan en la observación de que para grandes problemas con muchas variables binarias, la mayoría de ellas valen 0 y relajarlas, es decir dejar que tomen cualquier valor en el intervalo $[0,1]$, es innecesario en la mayoría de los casos.

1.5. Métodos de resolución heurísticos

Como ya se ha indicado, la complejidad de este tipo de problemas hace que los algoritmos exactos sean costosos y prácticamente imposibles de utilizar cuando el problema tiene un gran número de nodos. Es por esto que se han ido aplicando otros tipos de algoritmos que, aunque no garantizan la obtención de la mejor solución global, tienen una complejidad mucho menor y la solución encontrada se espera que difiera poco de la óptima.

1.5.1. Heurísticas constructivas

Las heurísticas constructivas se utilizan generalmente para construir soluciones iniciales usadas en algoritmos de búsqueda. Una de las más utilizadas es el algoritmo de Clarke and Wright. Este algoritmo comienza construyendo rutas de ida y vuelta $(0,i,0)$ para $i = 1,\ldots,n$ y, gradualmente, va uniendo dos rutas $(0,\ldots,i,0)$ y $(0,j,\ldots,0)$ en $(0,\ldots,i,j,\ldots,0)$ reduciendo el coste en $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ que puede ser calculado a priori. Inicialmente se tomaban las rutas sucesivamente, pero en la versión paralela del algoritmo se juntan aquellas rutas que proporcionan la mayor reducción en el coste hasta que ya no hay más uniones válidas.

1.5.2. Heurísticas clásicas

Las heurísticas clásicas realizan movimientos entre rutas, bien quitando η arcos y añadiendo otros η (siendo η un valor modificado dinámicamente), o bien cambiando η clientes consecutivos de ruta, entre otros. Esto hace que explorar de forma completa todos los vecinos, es decir probar todos los movimientos posibles, requiera $O(n^2|K|^2)$ operaciones. Por ello para grandes problemas se recomienda reducir la lista de movimientos, por ejemplo con el método *granular search* que considera restricciones geográficas para evitar movimientos entre nodos distantes.

1.5.3. Metaheurísticas de búsquedas locales

Estos métodos exploran el espacio de soluciones moviendo la solución actual x_t a alguna de sus vecinas $N(x_t)$, donde $N(x_t)$ representa el conjunto de soluciones que comparten ciertas características con x_t. Están diseñados para intentar escapar de óptimos locales y evitar ciclos. Denotaremos mediante $f(x_t)$ el coste asociado a la solución x_t.

Recocido simulado. Llamada en inglés *Simulated Annealing* (SA), consiste en elegir una solución aleatoria $x \in N(x_t)$. Si $f(x) \leq f(x_t)$ entonces $x_{t+1} = x$. En otro caso $x_{t+1} = x$ con probabilidad p_t y $x_{t+1} = x_t$ con probabilidad $1 - p_t$, donde $p_t = P(x,x_t,t)$ es generalmente una función decreciente de t. Se suele definir como $p_t = exp(-[f(x) - f(x_t)]/\theta_t)$ con θ_t una función decreciente de t.

Búsqueda determinista. También llamada *Deterministic Annealing* (DA) es equivalente al recocido simulado, salvo que en este caso la elección de x es determinista tomando $x_{t+1} = x$ si $f(x) \leq \sigma f(x^*)$ donde σ es un parámetro ligeramente mayor que 1 (por ejemplo 1,05) y x^* es la mejor solución encontrada hasta el momento.

Búsqueda tabú. Como veremos en el capítulo 2, los algoritmos de búsqueda tabú, *Tabu Search* (TS), recorren las soluciones moviéndose a la mejor de sus vecinas que sean no-tabú (se suelen considerar tabú aquellas que comparten características con las previamente visitadas). En el capítulo 3 veremos una implementación de este tipo de algoritmos en detalle para el problema CVRP.

Búsqueda local iterativa. Llamada en inglés *Iterated Local Search* (ILS), basa su implementación en la búsqueda local heurística: busca entre los vecinos utilizando algún mecanismo de búsqueda local, le aplica una pequeña perturbación a la solución encontrada, y repite el proceso hasta alcanzar algún criterio de parada. Esta perturbación debe ser diseñada con cuidado para no ser revertida por la búsqueda local, y al mismo tiempo mantener las propiedades de la solución actual.

Búsqueda en entornos variables. En inglés *Variable Neighborhood Search* (VNS), se basa en la idea de que una solución óptima local para un tipo de vecinos, no tiene porqué serlo considerando otros tipos de vecinos, mientras que la solución óptima global lo será independientemente de los vecinos utilizados. Dada una lista finita de algoritmos de búsqueda de vecinos, si encuentra una solución que mejora la actual, la toma y vuelve al primero de ellos. En caso de haber encontrado un óptimo local pasa a utilizar el siguiente algoritmo de búsqueda de vecinos. Una vez utilizados todos los algoritmos disponibles se termina el procedimiento.

1.5.4. Metaheurísticas basadas en poblaciones

A diferencia de los anteriores, éstos métodos están diseñados a partir del comportamiento de las poblaciones en la naturaleza.

Colonia de hormigas. En inglés *Ant Colony Optimization* (ACO), consiste en aplicar la información de decisiones anteriores para alterar la probabilidad de elegir candidatos. Se basa en el comportamiento de las colonias de hormigas y el método que siguen para encontrar bienes. Comienzan moviéndose aleatoriamente dejando feromonas a su paso que se evaporan gradualmente y que dependen de la calidad del bien encontrado. El resto de hormigas tienden a seguir los rastros con mayor nivel de feromonas. En el caso del VRP, los arcos que están en soluciones con menor coste reciben más feromonas, lo que incrementa su probabilidad de ser seleccionados en sucesivas soluciones.

Métodos evolutivos o genéticos. En inglés *Genetic Algorithms* (GA) o *Evolutionary Algorithms* (EA), imitan la selección natural. Comienzan con un conjunto de soluciones aleatorias, la población, y en cada iteración se seleccionan algunas aleatoriamente que se combinan para generar nuevas soluciones (hijos), que pueden a su vez ser modificadas aleatoriamente. De entre la población actual y los hijos, se selecciona la nueva población con algún criterio y se repite el proceso. Al igual que en la selección natural, el objetivo es la mejora de las sucesivas poblaciones. El procedimiento se repite hasta realizar un número prefijado de iteraciones o hasta que alguna solución de la población se considere satisfactoria.

1.6. Métodos Híbridos

En los apartados anteriores se han descrito algunos métodos que se han utilizado para resolver el VRP o algunas de sus variantes. No obstante, conviene señalar que la frontera entre ellos está cada vez más difuminada, y han surgido algoritmos híbridos que combinan dos o más algoritmos metaheurísticos, ya sean trabajando juntos o complementándose el uno al otro, o bien utilizan un método metaheurístico que incorpora la obtención de soluciones parciales de forma exacta.

Capítulo 2

Algoritmos de búsqueda tabú

La Búsqueda tabú, *Tabu Search* (TS), es un tipo de metaheurística que se utiliza para guiar un proceso de búsqueda local, utilizando estructuras y mecanismos diseñados para visitar regiones a las que de otra manera sería difícil acceder, así como para evitar que la búsqueda quede atrapada en un mínimo local.

Este tipo de algoritmos están basados en ideas propuestas por Glover (1977), que aplicó originalmente para resolver problemas de programación entera. A lo largo de los años estas ideas se han ido mejorando, y sofisticando, siendo utilizadas en la resolución de problemas de muy diversas áreas, entre las que cabe destacar: planificación, telecomunicaciones, computación en paralelo, transporte y diseño de redes, optimización de estructuras, optimización en grafos, aprendizaje y redes neuronales, optimización estocástica y continua, fabricación, análisis financiero, etc. En Glover (1990) y Glover y Laguna (2013) puede encontrarse una relación más completa de problemas y áreas en los que la búsqueda tabú se ha utilizado con éxito.

En general, estos algoritmos tratan de optimizar, maximizar o minimizar, el valor de una *función objetivo* $f(x)$, con $x \in X$ el vector de variables de decisión, donde X representa el conjunto de soluciones admisibles, factibles o no, del problema que se pretende resolver. En los procesos de búsqueda local, cada solución x tiene asociado un entorno $N(x) \subset X$ de soluciones vecinas, cada una de las cuales se obtiene a partir de x mediante una transformación elemental llamada *movimiento*.

Un algoritmo de búsqueda local trata de mejorar una solución dada, que es inicialmente considerada como la solución actual x_a de un proceso iterativo. En cada iteración de este proceso se explora el entorno de sus soluciones vecinas $N(x_a)$ o, equivalentemente, el conjunto de movimientos que conducen a ellas, remplazándola con la 'mejor' encontrada en relación a la función objetivo. Durante todo este proceso iterativo, la mejor solución encontrada se va almacenando y, cuando algún *criterio de parada* especificado se cumple, el algoritmo se detiene y la devuelve como resultado. Este criterio de parada puede ser, por ejemplo, haber realizado un determinado número de iteraciones bien totales o bien desde la última actualización de la mejor solución, entre otros.

Si solo se permite actualizar la solución actual con una vecina $x_v \in N(x_a)$ que mejora estrictamente la función objetivo, es decir, si $f(x_v) < f(x_a)$ en el caso de minimizar, el proceso se detendrá al alcanzar un mínimo local, que no necesariamente tiene porqué ser la solución óptima buscada. Para evitar esto, la búsqueda tabú recopila y posteriormente explota información sobre las soluciones visitadas en las iteraciones previas. En su forma más simple, ciertas propiedades de esas soluciones, que se determinan en función del problema, se califican como prohibidas (tabú) durante un cierto número de iteraciones, de forma que en cada iteración se escoge el mejor vecino no tabú aunque éste no mejore la solución actual. En formas más avanzadas se recoge también información sobre la frecuencia de cada propiedad, bien sea para penalizar aquellas que más han aparecido, con objeto de diversificar la búsqueda hacia nuevas áreas del espacio de soluciones, o para incentivarlas, con lo que se permite intensificar la búsqueda en una zona concreta.

2.1. Búsqueda tabú básica

En esta sección introducimos los principales conceptos de los algoritmos de búsqueda tabú, para cuya exposición nos hemos basado en Glover (1989); Laguna (1994); Gendreau (2003); Glover y Laguna (2013). Con objeto de facilitar su comprensión, ilustraremos con ejemplos algunos de ellos utilizando el siguiente problema de planificación de trabajos.

2.1.1. Problema de planificación o secuenciación de trabajos (job scheduling)

Se tiene un conjunto de trabajos $\{j : 1 \leqslant j \leqslant n\}$ que deben ser realizados en orden, cada uno de los cuales tiene un tiempo de realización p_j, una fecha límite de entrega d_j y una penalización w_j por superarla. El objetivo del problema es encontrar el orden en el que los trabajos deben ser realizados para minimizar el coste ocasionado por superar las fechas límite, es decir, encontrar aquella permutación de los trabajos que minimiza

$$T = \sum_{j=1}^{n} w_j [C_j - d_j]^+$$

donde $[x]^+ = \max\{0, x\}$ y $C_j = \sum_{i=1}^{j} p_j$ es el instante en el que el trabajo j se completa. El conjunto de soluciones X en este caso es el conjunto de las permutaciones de n elementos.

2.1.2. Vecindad y movimientos

Como hemos indicado, la búsqueda tabú guía un proceso de búsqueda local. Por tanto, el primer paso es definir para cada solución x el entorno $N(x) \subset X$ de sus soluciones vecinas. Esta es la parte más compleja del algoritmo y en ocasiones decidir qué debe ser vecino y qué no de manera errónea suele ser la causa de que el algoritmo sea poco eficiente. Una alternativa es la definición de una función distancia sobre X, en cuyo caso las soluciones vecinas de una dada serán las que estén a una distancia menor que un valor fijado, sin embargo es habitual definir las soluciones vecinas como aquellas que se obtienen modificando de alguna manera la expresión de la solución de partida. Llamaremos *movimiento* al conjunto de estas operaciones que hay que realizar sobre una solución para obtener otra diferente.

El tipo de movimientos que se pueden realizar depende de la estructura del problema y, en gran medida, de la forma de expresar las soluciones, así como de las variables utilizadas para representarlas. Algunas posibilidades son el cambio de valor de una determinada variable, añadir o eliminar un elemento de un conjunto, intercambiar la posición de dos elementos, etc. Si consideramos el problema de planificación de trabajos, cuyas soluciones son permutaciones de los elementos $1, \ldots, n$, podemos considerar que una solución es vecina de otra si dos de sus trabajos están intercambiados, es decir, las soluciones $(\ldots, i, \ldots, j, \ldots)$ y $(\ldots, j, \ldots, i, \ldots)$, con $i \neq j$, son vecinas. En este caso, un movimiento es la transposición de (i, j). Una segunda posibilidad sería considerar inserciones en lugar de permutaciones, es decir, tomar el elemento i y trasladarlo a la posición j, desplazando el resto de elementos según corresponda. Así, la solución $(\ldots, i, \ldots, j-1, j, j+1, \ldots)$ se transformaría en $(\ldots, j-1, j, i, j+1, \ldots)$.

En cada iteración de un algoritmo simple se deberían generar todas las soluciones vecinas $x_v \in N(x_a)$ de la solución actual x_a, y evaluar para cada una de ellas la función objetivo. La mejor de ellas, x_m, reemplazará a x_a en la siguiente iteración, incluso si $f(x_m) > f(x_a)$. La eficiencia de este subproceso es crítica para la del algoritmo completo, pues es lo que más veces se ejecuta. Por ello, suele ser sustituido por un recorrido del conjunto de movimientos que se pueden aplicar a la solución actual y por la evaluación del *atractivo* de cada uno de ellos. Esta alternativa ofrece varias ventajas. Por un lado, no obliga a generar las soluciones vecinas, por lo que solo es necesario mantener una única solución completa en todo momento, la solución actual, con el consiguiente ahorro de espacio y tiempo. Por otro, aunque la medida más obvia para evaluar el atractivo de un movimiento es la diferencia de los valores de la función objetivo $f(x_v) - f(x_a)$, se pueden utilizar otras alternativas en caso de que ésta no pueda ser calculada directa y eficientemente, por ejemplo con algún tipo de medida local.

2.1.3. Características y estructura tabú

Como hemos señalado, la búsqueda tabú recopila información para posteriormente utilizarla. La forma más básica de explotar esta información, que constituye el núcleo central de los algoritmos de búsqueda tabú, es prohibir que ciertos acontecimientos ocurran, al menos durante un cierto tiempo. Por ejemplo, tras abandonar un mínimo local debería prohibirse realizar en la siguiente iteración el movimiento inverso que nos ha sacado de él. Esta información que se recoge, y que debe ser adecuada para alcanzar este objetivo, está generalmente relacionada con atributos de los movimientos y de la expresión de las soluciones.

Se define como *restricción tabú* lo que se desea penalizar, por ejemplo aquellos movimientos que desplacen de nuevo los trabajos que una permutación (i, j) acaba de intercambiar. Y llamaremos *atributo tabú* a la información que se almacena en la estructura tabú y que es necesaria para determinar si un movimiento está o no restringido, en el ejemplo anterior almacenaríamos cada uno de los trabajos por separado i y j. Otros ejemplos, utilizando el problema de planificación, pueden ser: si definimos como movimientos la inserción de un trabajo j en la posición i desde la posición k, se puede definir como restricción tabú que el trabajo j pase a estar en la posición k, mediante el atributo tabú (j,k), con lo que estaríamos evitando que el trabajo j vuelva de nuevo a la posición k durante algunas iteraciones. Si por el contrario tomamos como movimientos las permutaciones de dos elementos (i, j), podemos definir como restricción tabú la propia permutación (i, j), en cuyo caso los atributos tabú almacenados serían las parejas (i, j).

La estructura tabú es el elemento principal de la búsqueda tabú. Consiste en un listado de *atributos tabú*, normalmente restricciones que han sido marcadas como tabú en las iteraciones anteriores a la actual. Su misión consiste en prohibir la visita de soluciones que han sido visitadas recientemente, o que tienen características muy similares a ellas. Por esto, el algoritmo descarta todos los movimientos considerados 'tabú' salvo que se cumpla un *criterio de aspiración*, que si se satisface permite revocar el estatus de tabú y entonces el movimiento puede ser elegido, incluso si es tabú. En la mayoría de los casos se toma como criterio de aspiración que la solución obtenida tras realizar el movimiento mejore la mejor solución obtenida hasta el momento.

A la hora de almacenar la lista de atributos tabú también existen distintos métodos. Se puede almacenar cada elemento junto a un entero, indicando el número de iteraciones que debe permanecer tabú al que llamaremos *permanencia* (tenure), y disminuir su valor en cada iteración hasta que llegue a cero, momento en el que se quita de la lista. Si se opta por este método, es preferible almacenar en su lugar el atributo tabú junto a la iteración en la que dejará de ser tabú, es decir la iteración actual más la permanencia, con lo que no hace falta actualizar toda la estructura cada vez, únicamente al marcar como tabú. Para comprobar si un atributo es tabú basta con comprobar si la iteración actual supera o no la iteración almacenada. Si tomamos como ejemplo de atributos tabú las permutaciones de trabajos, se puede utilizar un array de dos índices; y si optamos por marcar como tabú los trabajos por separado basta un array de un solo índice. Otra alternativa consiste en utilizar una cola de prioridad con tamaño fijo, a la cual se le van añadiendo los atributos junto a la iteración en la que se incluyeron (si el atributo ya existe se elimina y se añade de nuevo). En el momento en el que la lista supere el tamaño máximo permitido se elimina aquel con menos prioridad, es decir, el primero que se añadió. Los atributos tabú serán aquellos que que se encuentren en la lista, independientemente del número de iteraciones que hayan pasado desde que se incluyeron. Esta estructura es similar a una lista FIFO (first-in first-out) pero sin permitir duplicados.

2.1.4. Un algoritmo básico

En este apartado vamos a explicar como se realizaría una implementación básica de un algoritmo que utilice mecanismos de búsqueda tabú. Para ello disponemos de una variable global que implementa la estructura tabú y de dos funciones, $esTabu(m,it)$ y $hacerTabu(m,it)$, que operan sobre ella. La primera determina si el movimiento m es o no tabú en la iteración it; la segunda declara el movimiento m como tabú a partir de la iteración it. También se tienen dos funciones $criterioParada(it)$ y

criterioAspiracion(m) que comprueban si se debe detener el algoritmo en la iteración *it* y si el movimiento *m* debe ser elegido aunque sea tabú, respectivamente.

En primer lugar se inicializan las variables, la solución inicial pasa a ser tanto la solución actual como la mejor solución encontrada y el número de iteraciones se inicializa a cero. En esta fase ningún movimiento es considerado tabú. Durante un número determinado de iteraciones, mientras no se cumpla el criterio de parada, se realizan las siguientes etapas:

1. Se comprueban los movimientos que se pueden aplicar a la solución actual y se toma el mejor de ellos, recordando que aquellos considerados tabú se descartan salvo que cumplan el criterio de aspiración.
2. Una vez identificado el mejor movimiento, se le aplica a la solución actual, modificándola, y se actualiza la estructura tabú marcando este movimiento realizado como tabú.
3. Se comprueba si la solución obtenida es mejor que la mejor encontrada hasta el momento, en cuyo caso se almacena, y se aumenta el número de iteraciones.

```
BúsquedaTabú(entrada:solucionInicial; salida:mejorSolucion)

//inicialización
solucionActual <- solucionInicial
mejorSolucion <- solucionInicial
estructuraTabu <- vacía
it <- 0

//iteración
MIENTRAS NO criterioParada(it)
    mejorMovimiento <- null

    //evaluación
    PARA_CADA movimiento EN movimientos(solucionActual)
        SI valor(movimiento) < valor(mejorMovimiento) Y
         ( NO esTabu(movimiento,it) O criterioAspiracion(movimiento) )
            mejorMovimiento <- movimiento
        FIN_SI
    FIN_PARA_CADA

    //modificación
    solucionActual <- aplicar(mejorMovimiento,solucionActual)
    hacerTabu(mejorMovimiento,it)

    //actualización
    SI f(solucionActual) < f(mejorSolucion)
        mejorSolucion <- solucionActual
    FIN_SI
    it<-it+1

FIN_MIENTRAS
```

2.2. Búsqueda tabú avanzada

La forma de explotar la información recogida en la estructura tabú sólo permite tener en cuenta los movimientos realizados recientemente, en unas pocas iteraciones previas a la actual. Por este motivo

es habitual referirse a ella como memoria a corto plazo. Aunque en muchos problemas la estructura tabú es suficiente para obtener buenas soluciones, en problemas más complejos se requieren distintas estrategias y estructuras adicionales para obtener un mayor rendimiento. En esta sección haremos una revisión de algunas de las muchas que se han presentado conforme los algoritmos han ido creciendo en nivel de sofisticación.

La mayor parte de las estrategias avanzadas que se incorporan a la búsqueda tabú requieren almacenar información a largo plazo, para así disponer de datos sobre lo ocurrido desde el inicio del proceso. El tipo de información que se almacena en estas estructuras puede ser muy variada y depende del objetivo que se quiera alcanzar. Lo más habitual es registrar la frecuencia de determinados acontecimientos, por ejemplo el número de veces que cada movimiento ha sido escogido como el mejor, la frecuencia con la que al ejecutar cada movimiento se ha alcanzado un mínimo local o una mejor solución, etc. Esta información puede ser explotada generalmente de dos formas opuestas: las estrategias de diversificación y las de intensificación.

Las estrategias de *diversificación* están diseñadas específicamente para escapar de ciclos, es decir, repeticiones de forma continuada de un conjunto de soluciones, y dirigir la búsqueda hacia zonas todavía no exploradas. En el proceso pueden aparecer ciclos si éstos tienen un tamaño mayor que la permanencia de los atributos tabú o si se permite escoger vecinos con el mismo valor que la solución actual, algo que muchos autores no recomiendan realizar. Las estrategias de diversificación suelen ser muy útiles cuando existen buenas soluciones que sólo pueden ser visitadas cruzando barreras o 'montes' de la topología del espacio de soluciones.

La forma más sencilla de implementar una estrategia de diversificación es modificar la función que evalúa el atractivo de los movimientos, penalizando aquellos que se han realizado más frecuentemente. Otras implementaciones consisten en que, tras un cierto número de iteraciones, en lugar de realizar el proceso de selección de vecinos se le aplica el movimiento menos usado a la solución actual. En otras ocasiones incluso, se reinicia el proceso entero como si se ejecutara de nuevo todo el algoritmo con la mejor solución encontrada como solución inicial.

Las estrategias de *intensificación* tratan de modificar las reglas de elección de vecinos para favorecer los mejores movimientos, los que más frecuentemente han contribuido a mejorar la solución actual, o la aparición de ciertos patrones frecuentes en las soluciones que en algún momento fueron consideradas las mejores. Visto de otro modo, lo que se intenta conseguir con esta estrategia es penalizar aquellos que en el pasado no han dado lugar a buenas soluciones.

La implementación más habitual consiste, al igual que en las estrategias de diversificación, en modificar el atractivo de los movimientos, por ejemplo sumándole una cantidad inversamente proporcional al número de veces que el movimiento se ha realizado desde el inicio del algoritmo.

Finalmente, otro tipo de estrategia de intensificación consiste en almacenar una lista de soluciones de élite, ya sean soluciones suficientemente separadas, esto es, que el número de movimientos para pasar de una a otra sea grande, o aquellas que en algún momento fueron la mejor solución encontrada. Después de que el criterio de parada detenga el algoritmo, se borra la memoria a largo plazo y se reinicia el proceso desde la mejor solución de esta lista, eliminándola de ella. Cuando la lista se queda vacía, o tras un número fijado de iteraciones, el algoritmo se detiene.

Independientemente de las estrategias de diversificación e intensificación que puedan implementarse con la memoria a largo plazo, y frecuentemente de forma simultánea, se introducen alteraciones en el esquema básico del algoritmo que en muchos problemas mejoran su rendimiento. Destacamos, por su sencillez e impacto, dos de ellas.

Si se trabaja con una estructura tabú con tiempo de permanencia, es usual considerar un valor variable. Cada vez que se añade un nuevo atributo a la estructura, su tiempo de permanencia se puede escoger aleatoriamente dentro de un intervalo fijado o bien como una función de la valoración del movimiento.

También es usual, especialmente cuando el entorno $N(x_a)$ de vecinos de la solución actual es muy numeroso, trabajar con un subconjunto de movimientos tentativos a explorar. La elección de este subconjunto puede ser aleatoria, con tamaño prefijado, o en función de la estructura del problema. Como ejemplo de esta segunda opción, en el problema de planificación de trabajos se pueden considerar úni-

camente aquellos movimientos que adelanten la finalización de un trabajo cuya fecha límite haya sido superada en la solución actual. Otra alternativa más simple, que se puede implementar junto a las anteriores, consiste en finalizar la comprobación de movimientos al obtener uno que mejore la solución actual. Estas tácticas aceleran globalmente la ejecución del algoritmo, pero pueden dejar inexplorados movimientos que conduzcan a mejores soluciones, por lo que es recomendable complementarlos con alguna estrategia de intensificación.

Capítulo 3

Un algoritmo de búsqueda tabú para el problema de rutas de vehículos

En este capítulo presentamos el algoritmo de búsqueda tabú TABUROUTE descrito en el artículo de Gendreau *et al.* (1994), así como una implementación propia realizada en el lenguaje de programación Java. Este algoritmo utiliza la metaheurística de búsqueda tabú para resolver una variante del problema CVRP asimétrico con tiempos de servicio, que presentamos en la sección 1.1. Más concretamente, el problema que TABUROUTE resuelve es el siguiente:

Sea $G = (V,A)$ una red dirigida, donde $V = \{v_0, \ldots, v_n\}$ es el conjunto de nodos y $A = \{(v_i, v_j) : i \neq j\}$ es el conjunto de arcos. El nodo v_0 denota el origen, del que parten $m \leqslant \bar{m}$ vehículos idénticos ($\bar{m}$ valor constante), y el resto son clientes. Cada nodo tiene asociado una demanda no negativa q_i ($q_0 = 0$), y un tiempo de servicio δ_i ($\delta_0 = 0$). Con cada arco (v_i, v_j) hay asociado un tiempo de transporte c_{ij}. El objetivo del problema es en encontrar un conjunto de rutas de mínimo coste de forma que todas ellas comiencen y terminen en el origen, todos los clientes sean visitados una y sólo una vez por algún vehículo y se cumplan las siguientes restricciones adicionales: 1) La suma total de las demandas de los clientes de una ruta no puede superar la capacidad de los vehículos Q, que se supone constante e igual para todos ellos. 2) La longitud de la ruta, es decir la suma de los tiempos de servicio de los clientes más los tiempos de transporte de los arcos recorridos, no puede superar un valor fijado L.

3.1. Algoritmo

En esta sección se da una visión general del algoritmo sin realizar ninguna consideración sobre su posible implementación. Para ello usaremos la siguiente notación. Una solución S es un conjunto de m rutas $R_1, \ldots, R_m$, donde $m \in [1, \bar{m}]$, y cada ruta es una lista ordenada de nodos $R = (v_0, v_{r_1}, \ldots, v_0)$, tales que cada cliente v_i, $i = 1 \ldots n$, es visitado por una y solo una ruta. Cuando un nodo v es visitado en una ruta R, se dice que v pertenece a la ruta R y escribiremos $v \in R$. Del mismo modo escribiremos $(v_i, v_j) \in R$ si los nodos v_i y v_j son visitados consecutivamente y en ese orden en R, es decir, si el arco es recorrido.

Estas rutas pueden ser factibles o no respecto de las restricciones de capacidad y longitud. Sobre el conjunto de todas las rutas definimos dos funciones objetivo:

$$F_1(S) = \sum_{r=1}^{m} \sum_{(v_i,v_j)\in R_r} c_{ij}$$

$$F_2(S) = F_1(S) + \alpha \sum_{r=1}^{m} \left[\left(\sum_{v_i \in R_r} q_i \right) - Q \right]^+ + \beta \sum_{r=1}^{m} \left[\left(\sum_{(v_i,v_j)\in R_r} c_{ij} + \sum_{v_i \in R_r} \delta_i \right) - L \right]^+$$

donde α y β son dos parámetros reales positivos y $[x]^+ = \max\{0, x\}$. Obsérvese que las funciones F_1 y F_2 coinciden para las soluciones factibles. De hecho, la función F_1 es la función objetivo que

se pretende minimizar, mientras que la función F_2 contiene dos términos adicionales que penalizan el exceso de capacidad y longitud, respectivamente, en las soluciones no factibles.

El algoritmo TABUROUTE recorre el conjunto de todas las soluciones, tanto las factibles como las que no lo son, intentando mejorar en cada iteración la solución actual, que denotaremos por S. En todo momento del algoritmo las expresiones F_1^* y F_2^* contendrán, respectivamente, el menor valor de F_1 y de F_2 encontrado hasta el momento. Del mismo modo S^* almacena la mejor solución factible encontrada, y $\tilde{S}^*$ la mejor solución, factible o no.

3.1.1. Algoritmos principales

Procedemos a explicar con detalle el propósito de los dos algoritmos principales. El algoritmo TABUROUTE es el algoritmo principal, que se encarga de dirigir el proceso, y realiza llamadas a SEARCH que se encarga de mejorar una solución dada mediante estrategias de búsqueda tabú. Ambos procedimientos están descritos en el artículo de Gendreau *et al.* (1994) y se apoyan en los procedimientos Stringing, Unstringing, US y GENIUS, presentes en el artículo de Gendreau *et al.* (1992), para realizar operaciones con rutas de manera eficiente. Todos estos algoritmos tienen parámetros de entrada que se explicarán en sus respectivos apartados.

TABUROUTE Parámetro de entrada: λ.

Este algoritmo comienza inicializando las variables globales que afectarán a todo el proceso de búsqueda. Se asigna el valor 1 a α y β, mientras que a F_1^* y F_2^* se le asigna infinito. Después, en una primera fase, se busca una primera solución mediante la generación de λ soluciones en el espacio de búsqueda. Para ello se realizan λ iteraciones del siguiente proceso:

1. Se elije un nodo aleatorio v_i.
2. A partir de la secuencia de nodos $(v_0, v_i, \ldots, v_n, v_1, v_{i-1})$, se genera un camino que pase por todos ellos y que es solución del TSP, utilizando GENIUS. El camino obtenido será de la forma $(v_0, v_{p_1}, \ldots, v_{p_n})$.
3. Empezando en v_0 se generan una solución S con un máximo de $\bar{m}$ rutas. La primera ruta contendrá los primeros nodos del camino $(v_0, v_{p_1}, \ldots, v_{p_{i-1}}, v_0)$ hasta aquel v_{p_i} cuya inclusión haga que la ruta deje de ser factible. El proceso se repite comenzando por v_{p_i} hasta que todos los nodos han sido utilizados (la solución será factible), o hasta que $\bar{m} - 1$ rutas se han generado, en cuyo caso los nodos restantes se añaden a la última ruta (la solución puede ser no factible).
4. Finalmente se llama a SEARCH con un conjunto de parámetros P_1 para mejorar esta solución, actualizando los valores de F_1, F_2, S^* y $\tilde{S}^*$ cuando sea necesario.

Una vez realizadas las λ iteraciones, se toma la mejor solución encontrada en todas ellas, que corresponde con S^* si F_1^* es finito, es decir se ha encontrado una solución factible, o con $\tilde{S}^*$ en caso contrario. A esta solución se le aplica de nuevo SEARCH con distintos parámetros P_2. Es en esta fase donde el algoritmo ocupa la mayor parte del tiempo y en la que, normalmente, se suele encontrar la mejor solución del proceso completo.

Por último se realiza una tercera llamada a SEARCH usando de nuevo la mejor solución encontrada (S^* o $\tilde{S}^*$ según corresponda). En esta última fase se utilizan parámetros P_3 elegidos para llevar a cabo una estrategia de intensificación, con la que se realiza una búsqueda exhaustiva en entornos cercanos. Para terminar, el algoritmo devuelve la solución almacenada en S^* si F_1^* es finito. En caso contrario no se ha encontrado ninguna solución factible.

SEARCH Parámetros de entrada $P = (W, q, p_1, p_2, \theta_{\min}, \theta_{\max}, g, h, n_{\max})$.

El procedimiento SEARCH es el que implementa realmente la búsqueda tabú. Dada una solución inicial se encarga de mejorarla aplicando las técnicas de búsqueda tabú, utilizando como movimiento la extracción de un nodo v de su ruta R_r y su reinserción en otra distinta R_s. La forma de evaluar el

atractivo de estos movimientos es calcular la diferencia entre el valor de las funciones objetivo antes y después de realizar la modificación. Tras realizar un movimiento se marca como tabú la pareja (v, R_r), con lo que evitamos que el nodo eliminado vuelva a ser reinsertado en la misma ruta durante algunas iteraciones.

Este procedimiento tiene como entrada un gran número de parámetros: W es un subconjunto de $V \setminus \{v_0\}$ que contiene los nodos que se permitirán mover de su ruta actual; q es el número de nodos de W que se utilizarán como candidatos; al insertar un nodo en una ruta diferente, ésta debe contener al menos uno de sus p_1 vecinos más cercanos; p_2 es el parámetro utilizado en `GENI`; $\theta_{\min}$ y $\theta_{\max}$ son los límites del intervalo del valor de la permanencia; g es un factor de escala que modifica la función de valoración de movimientos; h es la frecuencia con la que los valores de α y β son actualizados; y $n_{\max}$ es el máximo número de iteraciones que se permiten desde la última mejora en la función objetivo.

En primer lugar se inicializa $t = 1$ la variable que contiene el número de iteraciones realizadas. Dado que el número de movimientos posibles es muy elevado, al comienzo de cada iteración se construye una lista de movimientos potenciales. Para ello, primero se seleccionan aleatoriamente q nodos de W, y para cada nodo v se evalúa el coste de eliminarlo de su ruta actual R_r e insertarlo en otra ruta R_s que debe contener alguno de sus p_1 vecinos más cercanos, o en una ruta vacía si $m < \bar{m}$. Para cada uno de estos movimientos se repite el siguiente procedimiento:

1. Se calcula el coste de eliminar v de R_r mediante `Unstringing` con parámetro p_1 y de reinsertarlo en R_s mediante `Stringing` con parámetro p_2. Se obtiene la solución S'
2. Si el movimiento es tabú, se descarta salvo que $F_1(S') < F_1^*$, si S' es factible, o $F_2(S') < F_2^*$ si es no factible.
3. Si no ha sido descartado, se le asigna un valor $F(S') = F_2(S')$ si $F_2(S') < F_2(S)$, o $F(S') = F_2(S') + \Delta_{\max} \sqrt{m} \cdot g \cdot f_v$ en caso contrario, donde $\Delta_{\max}$ es la mayor diferencia observada en valor absoluto entre el valor de $F_2(S)$ obtenido en dos iteraciones sucesivas. f_v el número de veces que el nodo v ha sido movido, dividido por t. De esta manera estamos aplicando una estrategia de diversificación, penalizando los movimientos que más se han ejecutado si la solución no mejora la actual.

El movimiento que proporciona el menor valor de F es identificado. Llamaremos $\bar{S}$ a la solución que produce, que no necesariamente se implementa pues puede ser ventajoso intentar mejorar la solución S mediante el procedimiento `US`. Esta alternativa se realiza si se cumplen las siguientes condiciones: a) $F_2(\bar{S}) > F_2(S)$, b) S es factible, c) `US` no se ha utilizado en la iteración anterior. Si alguna de estas condiciones no se cumple, se sustituye $S = \bar{S}$.

Si se ha realizado el movimiento, reinsertar v en R_r se declara como tabú hasta la iteración $t + \theta$, donde $\theta \in [\theta_{\min}, \theta_{\max}]$ entero tomado aleatoriamente.

Se actualizan las variables F_1^*, F_2^*, S^*, $\tilde{S}^*$, $\Delta_{\max}$, m y f_v; y si t es múltiplo de h se actualizan también α y β de la siguiente manera: se comprueba la factibilidad de las h soluciones anteriores, si todas ellas han sido factibles respecto a la capacidad se ajusta $\alpha = \frac{\alpha}{2}$, en caso de que todas hayan sido no factibles se ajusta en su lugar por $\alpha = 2\alpha$. Se realiza el procedimiento análogo con β, esta vez comprobando la factibilidad respecto a la duración de la ruta.

Para finalizar, si tanto F_1^* como F_2^* no se han reducido en las $n_{\max}$ iteraciones previas (criterio de parada) el algoritmo termina. En otro caso se inicia una nueva iteración.

3.1.2. Algoritmos auxiliares

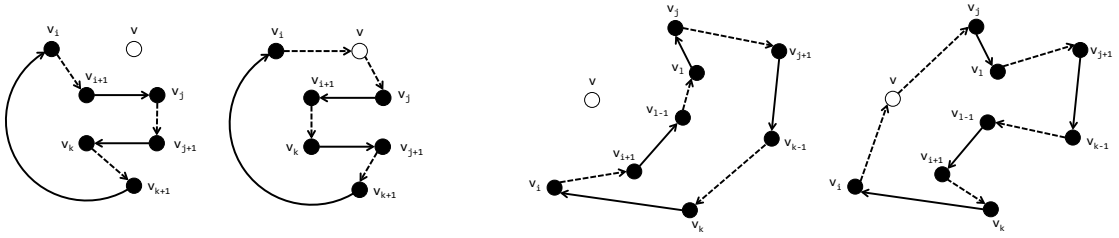
Estos algoritmos, desarrollados por Gendreau *et al.* (1992), han sido diseñados para resolver el problema TSP. `Stringing` y `Unstringing` se encargan, respectivamente, de insertar y eliminar nodos de rutas, `US` trata de optimizar una ruta y `GENIUS` es un procedimiento para generar rutas a partir de una lista de nodos.

Stringing Parámetro de entrada: p.

Este procedimiento trata de insertar un nodo v en una ruta minimizando el coste. Esto se consigue probando unas pocas alternativas particulares y eligiendo la mejor, en lugar de insertarlo entre dos nodos consecutivos directamente. Existen dos tipos de inserciones, además de considerar ambos sentidos de recorrido de la ruta.

En la primera, tipo I, se eligen tres nodos v_i, v_j y v_k, donde v_k se encuentra entre v_j y v_i (para un sentido concreto de la ruta) y se realizan las siguientes modificaciones: se eliminan los arcos (v_i, v_{i+1}), (v_j, v_{j+1}) y (v_k, v_{k+1}); se añaden los arcos (v_i, v), (v, v_j), (v_{i+1}, v_k) y (v_{j+1}, v_{k+1}); y se invierte el recorrido de los caminos $(v_{i+1}, \ldots, v_j)$ y $(v_{j+1}, \ldots, v_k)$.

Para la segunda, tipo II, se elige un nodo extra v_l entre v_i y v_j modificando la ruta de la siguiente manera: se eliminan los arcos (v_i, v_{i+1}), (v_{l-1}, v_l), (v_j, v_{j+1}) y (v_{k-1}, v_k); se añaden los arcos (v_i, v), (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}) y (v_{i+1}, v_k); y se invierte el recorrido de los caminos $(v_{i+1}, \ldots, v_{l-1})$ y $(v_l, \ldots, v_j)$.



(a) Tipo I

(b) Tipo II

Figura 3.1: Stringing

Como probar todas las combinaciones para una ruta con n nodos requiere orden $O(n^4)$, el algoritmo tiene como entrada un parámetro p pequeño y se toman únicamente los p-nodos más cercanos, aquellos cuyas aristas (v, w) tienen el menor coste. En particular, llamando $N_p(v)$ a la lista de los p-vecinos más cercanos a v, se toma $v_i, v_j \in N_p(v)$, $v_k \in N_p(v_{i+1})$ y $v_l \in N_p(v_{j+1})$.

Unstringing Parámetro de entrada: p.

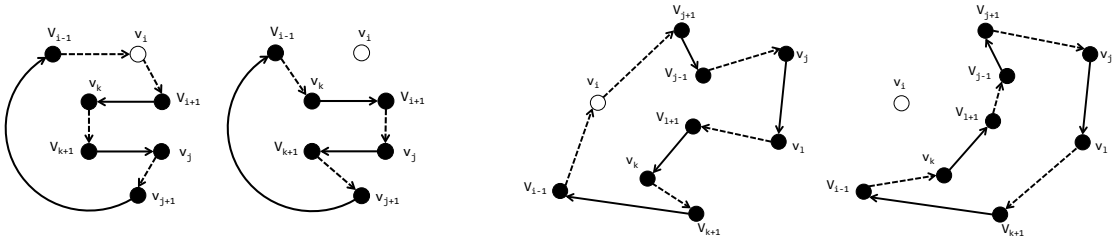
Este procedimiento trata de eliminar un nodo v_i de una ruta minimizando el coste. Lo realiza de manera análoga a `Stringing`, en este caso eliminándolo y probando unas pocas alternativas de reordenar la ruta, en lugar de quitar el nodo directamente. Al igual que en `Stringing` existen dos formas de realizar la comprobación, y además hay que considerar ambos sentidos de recorrido de la ruta.

Para el tipo I se eligen $v_j \in N_p(v_{i+1})$ y $v_k \in N_p(v_{i-1})$ entre v_{i+1} y v_{j-1}, modificando la ruta de la siguiente manera: se eliminan los arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_k, v_{k+1}) y (v_j, v_{j+1}); se añaden los arcos (v_{i-1}, v_k), (v_{i+1}, v_j) y (v_{k+1}, v_{j+1}); y se invierte el recorrido de los caminos $(v_{i+1}, \ldots, v_k)$ y $(v_{k+1}, \ldots, v_j)$.

Para el tipo II se toma en su lugar v_k entre v_{j+1} y v_{i-2} y además $v_l \in N_p(v_{k+1})$ entre v_j y v_{k-1} con las siguientes modificaciones: se eliminan los arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_{j-1}, v_j), (v_l, v_{l+1}) y (v_k, v_{k+1}); se añaden los arcos (v_{i-1}, v_k), (v_{l+1}, v_{j-1}), (v_{i+1}, v_j) y (v_l, v_{k+1}); y se invierte el recorrido de los caminos $(v_{i+1}, \ldots, v_{j-1})$ y $(v_{l+1}, \ldots, v_k)$.

US

Este procedimiento trata de optimizar el coste de una ruta reordenando sus nodos mediante la aplicación secuencial de los algoritmos `Unstringing` y `Stringing` a sus nodos. A pesar de que ambos algoritmos son similares y aparentemente opuestos, la ruta obtenida tras eliminar e insertar un nodo no



(a) Tipo I

(b) Tipo II

Figura 3.2: Unstringing

es necesariamente la de partida. US aprovecha esta característica para mejorar una ruta realizando el procedimiento a todos los nodos uno a uno, pero como en ocasiones el quitar y añadir un nodo empeora el valor de la función objetivo, el algoritmo almacena la mejor ruta encontrada en cada momento. Cada vez que al quitar y añadir un nodo la ruta mejora, ésta se almacena y se comienza el proceso de nuevo desde el primer nodo. Si se le ha realizado el procedimiento a todos los nodos sin obtener ninguna mejora, el algoritmo termina devolviendo la mejor ruta guardada.

GENI y GENIUS Parámetro de entrada: lista de nodos.

Ambos procedimientos resuelven el problema TSP, generando una ruta de longitud mínima insertando los nodos de entrada de forma secuencial, en una ruta inicialmente vacía, mediante el algoritmo `Stringing`. La diferencia entre ambos es que `GENIUS` tiene un paso adicional: le aplica `US` a la ruta tras la inserción de cada uno de los nodos, antes de pasar al siguiente.

3.1.3. Valores de los parámetros

Los algoritmos utilizados admiten como entrada una serie de parámetros. En nuestra implementación hemos utilizado los mismos valores que se describen en el artículo de Gendreau *et al.* (1994), que detallamos a continuación:

El único parámetro de `TABUROUTE` es el número λ de soluciones iniciales que se utilizarán. Tal y como se indica en el artículo, es beneficioso usar un valor mayor que 1 y tan grande como $\frac{\sqrt{n}}{2}$. Hemos tomado $\lambda = \frac{\sqrt{n}}{2}$.

El subalgoritmo `SEARCH` tiene 9 parámetros de entrada $P = (W, q, p_1, p_2, \theta_{\min}, \theta_{\max}, g, h, n_{\max})$ y se llama en tres ocasiones diferentes, con parámetros P_1, P_2 y P_3. Si no se especifica lo contrario, se usa el mismo valor en las tres ocasiones.

W es el conjunto de q nodos que se permitirán mover durante la búsqueda. En P_1 y P_2 hemos tomado $W = V \setminus \{v_0\}$ y $q = 5m$ para asegurarnos de que al menos se selecciona un nodo de cada ruta. En P_3, intensificación, se toman los $\frac{n}{2}$ nodos con mayor f_v, y $q = |W|$. De esta manera se toman aquellos nodos que más se han movido durante todo el algoritmo, y por tanto es más probable que den lugar a una mejora de la solución.

Los parámetros p_1 y p_2 son los parámetros usados en los algoritmos `Unstringing` y `Stringing` respectivamente. Tal y como indica el artículo, tomar $p_2 = 5$ proporciona un equilibrio entre tiempo y efectividad. Se define p_1 como $p_1 = \max\{k, p_2\}$ donde k es el número de nodos de la ruta que contiene el nodo que se va a eliminar.

Los valores de $\theta_{\min}$ y $\theta_{\max}$ se utilizan como extremos del intervalo que indica el número de iteraciones que un movimiento se mantiene tabú. Tal y como indican en Reeves (1993), se han tomado $\theta_{\min} = 5$ y $\theta_{\max} = 10$.

El parámetro g se utiliza como parámetro de escala para la penalización de movimientos usados frecuentemente. En nuestro caso $g = 0{,}01$.

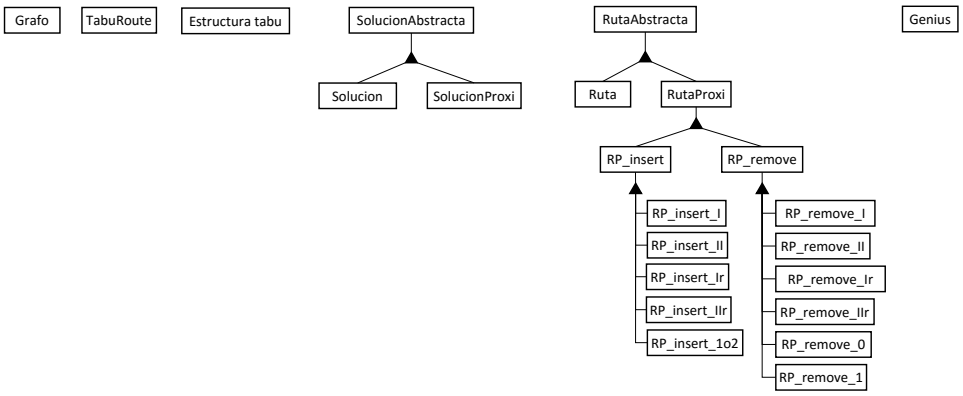
Para el valor de h, que indica el número de iteraciones que deben pasar para cambiar el valor de α y β si las h soluciones previas han sido todas factibles o todas no factibles, hemos tomado $h = 10$.

Por último, el valor de $n_{\max}$ indica el número de iteraciones que deben suceder sin haber obtenido una mejora para que el algoritmo se detenga. El tiempo de ejecución del algoritmo está directamente relacionado con este parámetro, si es muy bajo algunas soluciones buenas no serán visitadas, y si es muy alto el algoritmo trabajará durante más tiempo sin obtener ninguna mejora. Para la implementación hemos tomado $n_{\max} = n$ en P_1 y P_3, y $n_{\max} = 50n$ en P_2 (diversificación), pues es la parte más importante del algoritmo y donde la búsqueda debe ser más exhaustiva.

3.2. Implementación

Hemos realizado una implementación en el lenguaje de programación Java de los algoritmos anteriores para resolver el problema CVRP sobre grafos dirigidos. Esto nos ha permitido realizar una aproximación orientada a objetos y representar con relativa sencillez las rutas y las soluciones, así como la inserción o la eliminación de un nodo en una ruta antes de que estas operaciones de realicen de forma efectiva. Concretamente hemos utilizado el patrón de programación Proxi para este propósito. No obstante, nuestra implementación no es totalmente orientada a objetos: para asegurar una mayor eficiencia algunas clases, en particular `Grafo` y `TabuRoute` contienen variables globales de acceso público.

En el anexo de este trabajo se encuentra el código completo. En los párrafos siguientes presentamos los detalles más relevantes de las 23 clases que se han implementado, así como su estructura.



Grafo: Representa un problema CVRP, esto es, el número de nodos del grafo, la matriz de adyacencia, el tiempo de servicio y la demanda de cada nodo, la capacidad de los vehículos, el número de vehículos disponibles y la longitud máxima de cada ruta. También contiene el método utilizado para leer el fichero que contiene estos valores. Todos los métodos y variables de esta clase son estáticos, lo que impide que el algoritmo pueda ser ejecutado para varios problemas simultáneamente.

Además, para cada nodo v_i se construye y almacena una lista de los nodos v_j, con $i \neq j$, ordenada de menor a mayor mediante $\min\{c_{ij}, c_{ji}\}$, esto es, aquellos nodos más cercanos en ambos sentidos de recorrido. Esta estructura es utilizada para determinar en qué rutas puede ser insertado un nodo usando `Stringing`, pues en lugar de comprobar si una ruta concreta contiene uno de sus p-nodos mas cercanos, se toman directamente las rutas que contienen estos p-nodos.

TabuRoute: Contiene los algoritmos `TABUROUTE` y `SEARCH`, y es el encargado de iniciar todo el proceso y almacenar las variables F_1^*, F_2^*, S^* y $\tilde{S}^*$, así como α y β. Al igual que en Grafo, todos los métodos y variables son estáticos.

EstructuraTabu: Representa la estructura de memoria que almacena los atributos tabú, parejas (nodo,ruta), junto a la iteración a partir de la cual dicho atributo dejará de ser tabú.

RutaAbstracta: Es una clase abstracta que sirve de base para una jerarquía de clases que permiten representar tanto rutas 'reales', con la lista de nodos, como rutas 'virtuales' resultado de realizar modificaciones, inserciones y eliminaciones de nodos, a rutas reales sin generar un nuevo objeto. Contiene dos clases internas a las que tienen acceso sus clases derivadas: Nodo y NodoComparador. Los objetos de la primera representan la pertenencia de un nodo a una ruta concreta, de modo que una ruta se puede implementar como una lista circular doblemente enlazada de estos objetos. La segunda es una implementación de la interfaz Comparator de Java que nos permitirá mantener una lista ordenada de nodos.

El motivo por el que se ha elegido hacer una implementación propia del tipo abstracto de datos 'lista circular doblemente enlazada', en lugar de usar alguna de las clases disponibles en el lenguaje Java, es asegurar el acceso directo a los nodos para realizar eficientemente las operaciones de eliminación e inserción.

Ruta: Representa una ruta 'real', una secuencia ordenada de nodos que empieza y termina en el origen. Esta es la clase más compleja del proyecto pues aparte de contener las funciones básicas de una ruta, como comprobar si es factible y hallar su coste, también contiene los algoritmos `Stringing` y `Unstringing`.

Además, al igual que en la clase Grafo, para cada uno de los nodos v_i del problema, se construyen dos listas de nodos de la ruta $v_j \in R$, con $i \neq j$, ordenadas por c_{ij} y c_{ji} respectivamente. Esto se utiliza en los algoritmos `Stringing` y `Unstringing` para hallar los vecindarios $N_p(v)$, que son los p primeros elementos de una u otra lista.

RutaProxi: Esta clase representa una modificación de una ruta existente. RutaProxi es una clase abstracta implementada en 11 clases específicas que representan cada tipo de inserción y eliminación. Cada una de estas clases contiene información sobre la ruta original, el movimiento que debe ser realizado, y los valores de F_1 y F_2 de la ruta que se obtendría tras la aplicación del procedimiento, calculados a partir de la diferencia con respecto a la ruta original. En ningún momento se generan las rutas resultado, únicamente se modifica la ruta original cuando se requiere, momento a partir del cual el objeto queda inservible y lanza una excepción si se intenta usar.

SolucionAbstracta: Al igual que RutaAbstracta, esta clase abstracta sirve de base a una jerarquía, formada por las clases Solucion y SolucionProxi, que implementan el patrón de programación Proxi. Mientras que los objetos de la clase Solucion representan una solución concreta, los de SolucionProxi representan un movimiento completo sobre ella, tal y como se comenta a continuación. En este caso no ha sido necesario implementar ninguna estructura adicional.

Solucion: Representa una solución concreta, un conjunto de rutas. También contiene una referencia a una ruta siempre vacía (sólo el origen) para hacer más sencilla la comprobación de los candidatos.

SolucionProxi: Representa la modificación de una solución, en la que un nodo determinado se elimina de la ruta que lo contiene y se añade a otra ruta diferente. Equivale a la misma estructura que RutaProxi, pero como en este caso la modificación es única (un nodo se elimina de una ruta y se añade en otra), esta clase realiza la modificación, lanzando una excepción si un objeto se intenta usar tras haber modificado la solución de partida.

Genius: Contiene el algoritmo `GENIUS`. Dada una lista de nodos devuelve otra lista indicando el orden de los nodos que tendrían si se generase una ruta mediante `GENIUS`.

3.3. Resultados y análisis

Para comprobar la eficacia del algoritmo implementado, hemos utilizado los problemas del capítulo 11 del libro de Christofides *et al.* (1979). Estos problemas son los utilizados en el artículo del algoritmo TABUROUTE de Gendreau *et al.* (1994), cuyos resultados hemos comparado. Estos problemas contienen entre 50 y 199 nodos, sin incluir el origen. Los problemas 1-5 y 11-12 tienen únicamente restricciones de capacidad, siendo los problemas 6-10 y 13-14 los mismos, respectivamente, añadiendo restricciones de longitud. En los problemas 1-10 los nodos están dispersos en el plano, mientras que en 11-14 se encuentran agrupados.

Lo hemos ejecutado sobre un ordenador Intel Core i7-5500U CPU, 2401 Mhz, 7.2Gflops, utilizando Java 1.8 (jdk1.8.0_92). Los parámetros utilizados han sido siempre los mismos, mencionados en 3.1.3. El valor de la distancia recorrida ha sido redondeado a 2 decimales y en las operaciones internas se ha trabajado con 4 decimales.

Las siguientes tablas muestran las soluciones obtenidas en una única ejecución del algoritmo:

Problema número 1

rutas													ciudades visitadas	$Q = 160$ capacidad	$L = 999999,00$ tiempo
0	46	5	49	10	39	33	45	15	44	37	12	0	11	160	99,25
0	11	2	29	21	16	50	34	30	9	38	0		10	159	99,33
0	18	13	41	40	19	42	17	4	47	0			9	157	109,06
0	6	14	25	24	43	7	23	48	27	0			9	152	98,45
0	8	26	31	28	3	36	35	20	22	1	32	0	11	149	118,52
						k=5							n=50	777	524,61

Problema número 2

rutas													ciudades visitadas	$Q = 140$ capacidad	$L = 999999,00$ tiempo
0	57	15	37	20	70	60	71	69	36	5	29	0	11	139	114,76
0	16	23	56	41	64	42	43	63	0				8	134	108,59
0	72	39	9	25	55	31	10	58	0				8	139	110,55
0	53	11	66	65	38	0							5	129	77,16
0	30	48	47	21	61	22	1	73	0				8	140	92,72
0	68	2	74	28	62	33	6	0					7	139	62,25
0	17	40	12	26	67	75	0						6	137	43,41
0	7	35	14	59	19	54	13	27	0				8	140	97,20
0	51	49	24	18	50	32	44	3	0				8	135	89,66
0	34	46	8	52	45	4	0						6	132	47,39
						k=10							n=75	1364	843,68

Problema número 3

rutas																	ciudades visitadas	$Q = 200$ capacidad	$L = 999999,00$ tiempo
0	6	99	61	16	86	38	44	14	43	42	87	13	0				12	194	111,50
0	94	95	97	92	37	98	100	91	85	93	59	96	0				12	199	59,35
0	89	18	83	60	5	84	17	45	8	46	36	47	48	82	7	52	0	16	200	124,38
0	31	10	32	90	63	64	49	19	11	62	88	0						11	175	124,65
0	76	77	3	78	34	35	71	65	66	20	30	70	1	69	27	0		15	192	123,46
0	50	51	9	81	33	79	29	24	68	80	12	28	0					12	188	98,97
0	4	56	23	67	39	25	55	54	26	0								9	153	107,17
0	58	2	57	15	41	22	75	74	72	73	21	40	53	0				13	157	83,10
									k=8									n=100	1458	832,57

Problema número 4

rutas																	ciudades visitadas	$Q = 200$ capacidad	$L = 999999,00$ tiempo
0	53	58	137	2	115	57	144	87	97	92	59	95	117	13	0			14	195	66,00
0	93	85	91	141	44	140	38	14	119	100	37	98	0					12	196	93,15
0	27	69	122	30	128	131	32	90	63	126	108	10	31	127	0			14	200	88,51
0	111	50	102	33	81	120	9	103	51	1	132	0						11	191	74,38
0	96	104	99	61	16	86	113	17	84	5	118	60	0					12	198	81,45
0	105	40	21	73	72	74	75	133	22	41	145	15	43	142	42	0		15	190	98,84
0	28	76	80	150	68	121	29	24	134	54	109	12	138	0				13	190	76,50
0	26	149	130	55	25	139	39	67	23	56	4	110	0					12	191	108,63
0	89	147	6	94	112	0												5	99	29,40
0	116	77	3	79	129	78	34	135	35	136	65	71	66	20	70	101	0	16	200	122,37
0	88	148	62	107	11	64	49	143	36	47	19	123	7	0				13	196	120,33
0	146	52	106	82	48	124	46	45	125	8	114	83	18	0				13	189	89,95
									k=12									n=150	2235	1049,51

Problema número 5

rutas																	ciudades visitadas	$Q = 200$ capacidad	$L = 999999,00$ tiempo
0	147	6	183	94	95	97	87	137	58	152	0						10	193	46,63
0	153	106	194	7	82	18	166	89	112	156	0						10	158	56,99
0	64	49	143	36	47	168	48	124	46	174	8	114	0				12	199	130,80
0	61	16	86	140	38	14	192	119	44	141	191	91	0				12	191	103,37
0	96	104	99	5	84	173	113	17	45	125	199	83	60	118	0		14	197	82,39
0	132	69	101	162	31	190	127	167	27	0							9	152	42,88
0	146	88	148	62	159	11	175	107	19	123	182	52	0				12	194	77,00
0	171	133	22	41	145	15	43	142	42	172	144	57	178	115	2	0	15	198	101,05
0	180	198	110	197	56	186	23	75	74	72	73	21	40	0			13	193	79,60
0	28	138	154	12	177	109	195	26	105	53	0						10	181	45,30
0	13	117	151	92	98	37	100	193	85	93	59	0					11	190	56,16
0	111	50	102	157	185	79	129	3	158	77	196	76	0				12	199	57,06
0	161	71	66	65	136	35	135	164	34	78	169	0					11	191	125,69
0	130	165	55	25	170	67	39	187	139	155	4	0					11	181	94,81
0	184	116	68	150	80	121	29	24	163	134	54	179	149	0			13	193	77,36
0	33	81	120	9	103	188	128	20	51	122	1	176	0				12	190	90,95
0	10	189	108	90	126	63	181	32	131	160	30	70	0				12	186	92,32
									k=17								n=199	3186	1360,35

Problema número 6

rutas												ciudades visitadas	$Q = 160$ capacidad	$L = 200,00$ tiempo
0	32	11	16	29	21	50	34	30	9	38	0	10	141	195,33
0	12	37	44	15	45	33	39	10	49	5	0	10	155	199,12
0	14	25	13	41	40	19	42	17	0			8	131	189,94
0	27	48	8	26	7	43	24	23	6	0		9	133	190,64
0	2	20	35	36	3	28	31	22	1	0		9	137	198,08
0	46	47	4	18	0							4	80	82,33
k=6												n=50	777	1055,43

Problema número 7

rutas										ciudades visitadas	$Q = 140$ capacidad	$L = 160,00$ tiempo
0	4	45	29	5	37	36	47	74	0	8	140	157,79
0	12	9	25	55	50	32	17	0		7	136	158,40
0	48	69	71	60	70	20	0			6	77	159,13
0	40	72	39	31	10	58	26	0		7	140	155,93
0	3	44	18	24	49	16	51	0		7	104	156,10
0	73	42	64	22	62	2	68	0		7	111	159,95
0	33	1	43	41	56	23	63	0		7	121	154,46
0	46	8	19	59	14	35	7	0		7	138	151,36
0	53	11	66	65	38	0				5	129	127,16
0	6	28	61	21	30	75	0			6	133	135,45
0	27	15	57	13	54	52	34	67	0	8	135	157,54
k=11										n=75	1364	1673,25

Problema número 8

rutas													ciudades visitadas	$Q = 200$ capacidad	$L = 230,00$ tiempo
0	52	7	19	11	64	49	36	47	48	82	18	0	11	178	227,55
0	50	33	81	9	35	71	65	66	20	51	1	0	11	163	227,93
0	27	69	70	30	32	90	63	10	62	88	31	0	11	155	200,12
0	12	80	68	24	29	34	78	79	3	77	76	28	0	12	169	210,26
0	54	55	25	39	67	23	56	4	26	0			9	153	197,08
0	95	97	92	37	98	100	91	85	93	59	94	0	11	188	168,46
0	58	2	57	41	22	75	74	72	73	21	40	53	0	12	149	194,83
0	6	96	99	5	84	17	45	46	8	83	60	89	0	12	113	212,04
0	61	16	86	38	44	14	43	15	42	87	13	0	11	190	228,60
k=9													n=100	1458	1866,87

Problema número 9

rutas														ciudades visitadas	$Q = 200$ capacidad	$L = 200,00$ tiempo
0	18	114	8	46	36	47	124	48	82	106	0			10	152	193,76
0	107	11	64	49	143	19	123	7	52	0				9	128	199,95
0	104	99	5	84	17	45	125	83	60	118	147	89	0	12	192	195,70
0	146	127	88	148	62	126	63	90	70	101	69	27	0	12	119	195,92
0	113	86	140	38	14	119	44	141	16	0				9	174	194,07
0	6	96	59	93	85	61	91	100	98	37	92	95	0	12	176	183,76
0	13	87	144	57	15	43	142	42	97	117	94	112	0	12	176	199,20
0	53	40	21	73	74	133	22	41	145	115	2	137	58	0	13	151	197,04
0	28	116	68	80	150	54	130	55	25	149	26	105	0	12	184	199,27
0	110	4	139	39	67	23	56	75	72	0				9	172	186,11
0	76	77	3	129	78	34	35	135	120	9	0			10	132	191,20
0	138	12	109	134	24	29	121	79	81	33	102	0		11	181	192,13
0	132	1	122	30	128	131	32	108	10	31	0			10	165	179,08
0	111	50	51	103	71	136	65	66	20	0				9	133	199,64
k=14														n=150	2235	2706,83

Problema número 10

rutas														ciudades visitadas	$Q = 200$ capacidad	$L = 200,00$ tiempo
0	89	166	114	8	174	46	45	125	199	83	60	118	0	12	196	199,58
0	64	49	143	36	47	168	124	0						7	105	182,61
0	27	132	176	1	185	79	129	3	158	77	196	76	28	0	13	194	192,64
0	146	52	153	106	194	7	182	148	88	31	190	127	69	0	13	197	191,81
0	165	55	25	170	67	39	187	139	155	4	0				10	162	194,62
0	6	99	104	59	93	85	100	37	98	151	92	97	117	0	13	182	187,94
0	137	2	178	115	145	41	22	133	74	171	152	58	0		12	191	186,69
0	180	198	110	197	56	23	186	75	72	73	21	40	0		12	185	199,88
0	159	62	11	175	107	19	123	48	82	18	0				10	190	182,49
0	154	138	12	80	150	177	109	195	149	26	105	53	0		12	199	170,32
0	111	184	116	68	121	29	24	163	134	54	130	179	0		12	197	198,22
0	160	131	32	181	63	126	90	108	10	189	0				10	160	191,62
0	156	0													1	19	14,47
0	13	87	144	57	15	43	142	42	172	95	94	112	0		12	193	199,63
0	183	96	61	16	86	113	17	173	84	5	147	0			11	194	192,68
0	20	188	103	161	71	65	66	128	0						8	160	191,38
0	169	78	34	164	135	35	136	9	120	0					9	138	191,63
0	193	91	191	141	44	140	38	14	119	192	0				10	142	191,63
0	167	162	101	70	30	122	51	81	33	157	102	50	0		12	182	193,27
k=19															n=199	3186	3453,11

Problema número 11

rutas															ciudades visitadas	$Q = 200$ capacidad	$L = 999999,00$ tiempo
0	52	54	57	59	65	61	62	64	66	63	60	56	58	55			
	53	0													15	192	213,63
0	21	20	23	26	28	32	35	29	36	34	31	30	33	27			
	24	22	25	19	16	17	0								20	184	207,51
0	8	12	13	14	15	11	10	9	7	6	5	4	3	1			
	2	0													15	195	134,96
0	120	105	106	107	104	100	116	98	110	115	109	108	118	18			
	114	90	91	89	86	111	88	0							21	197	86,28
0	103	73	76	68	77	79	80	78	72	75	74	71	70	69			
	67	0													15	200	144,41
0	37	38	39	42	41	44	46	47	49	50	51	48	45	43			
	40	0													15	191	199,62
0	87	95	102	101	99	97	94	96	93	92	85	112	84	113			
	83	117	81	82	0										18	196	66,67
0	119	0													1	20	14,14
k=8															n=120	1375	1067,22

Problema número 12															ciudades	$Q = 200$	$L = 999999,00$
rutas															visitadas	capacidad	tiempo
0	98	96	95	94	92	93	97	100	99	0					9	190	95,94
0	5	3	7	8	11	9	6	4	2	1	75	0			11	170	56,17
0	67	65	63	74	62	66	0								6	150	43,59
0	43	42	41	40	44	46	45	48	51	50	52	49	47	0	13	160	64,81
0	57	59	60	58	56	53	54	55	0						8	200	101,88
0	13	17	18	19	15	16	14	12	10	0					9	200	96,04
0	34	36	39	38	37	35	31	33	32	0					9	200	97,23
0	21	22	23	26	28	30	29	27	25	24	20	0			11	170	50,80
0	69	68	64	61	72	80	79	77	73	70	71	76	78	81	0	14	200	137,02
0	90	87	86	83	82	84	85	88	89	91	0					10	170	76,07
k=10																n=100	1810	819,56

Problema número 13															ciudades	$Q = 200$	$L = 720,00$
rutas															visitadas	capacidad	tiempo
0	102	101	99	100	116	98	110	115	97	94	93	96	95	0	13	132	702,67
0	92	89	91	90	114	118	18	84	112	85	86	111	88	0	13	128	692,61
0	73	71	74	72	75	78	80	79	77	76	68	0			11	141	686,49
0	57	62	64	66	63	60	56	58	55	53	0				10	114	705,09
0	42	48	45	43	40	59	65	61	54	52	0				10	147	719,34
0	39	38	47	51	50	49	46	44	41	37	0				10	122	689,69
0	16	19	22	24	25	23	26	21	20	17	0				10	123	673,33
0	28	32	35	31	27	30	33	34	36	29	0				10	61	696,34
0	87	109	108	6	5	1	2	83	113	117	81	82	0		12	145	716,44
0	7	9	10	4	3	11	15	14	13	12	8	0			11	144	672,77
0	105	106	107	104	103	67	70	69	120	119	0				10	118	618,44
k=11															n=120	1375	7573,21

Problema número 14												ciudades	$Q = 200$	$L = 1040,00$
rutas												visitadas	capacidad	tiempo
0	67	65	63	62	74	72	61	64	68	66	0	10	190	958,14
0	75	96	95	97	100	99	2	1	3	5	0	10	180	990,29
0	98	94	93	92	85	84	82	83	86	87	0	10	200	1014,88
0	57	55	54	53	56	58	60	59	0			8	200	821,88
0	29	34	37	38	39	36	30	28	26	23	0	10	160	991,84
0	80	79	77	73	70	71	76	78	81	0		9	150	937,30
0	69	41	40	44	45	48	51	50	0			8	80	783,29
0	13	17	18	19	15	16	14	12	10	0		9	200	906,04
0	7	8	11	9	6	4	91	88	89	90	0	10	150	982,01
0	32	33	35	31	52	49	47	46	42	43	0	10	190	990,01
0	20	24	27	25	22	21	0					6	110	575,03
k=11												n=100	1810	9950,71

A continuación, comparamos el resultado de la función objetivo, la suma del tiempo de todas las rutas, entre los resultados que mencionan en Gendreau *et al.* (1994) y los obtenidos por la implementación desarrollada en esta memoria. Cabe mencionar que en todas las tablas que aquí presentamos hemos incluido el tiempo de servicio δ de los nodos, mientras que en Gendreau *et al.* (1994) hay que sumarlos aparte. En los problemas 1, 6 y 12 se obtiene el mismo resultado mientras que en el resto la diferencia relativa es menor que el 4 %.

Problema	1	2	3	4	5	6	7	8	9	10	11
Gendreau *et al.* (1994)	524,61	835,32	826,14	1031,07	1311,35	1055,43	1659,68	1865,94	2662,89	3394,75	1042,11
Implementación desarrollada	524,61	843,68	832,57	1049,51	1360,35	1055,43	1673,25	1866,87	2706,83	3453,11	1067,22
Diferencia relativa	0 %	1 %	0.78 %	1.79 %	3.74 %	0 %	0.82 %	0.05 %	1.65 %	1.72 %	2.41 %

Problema	12	13	14
Gendreau *et al.* (1994)	819,56	7545,93	9866,37
Implementación desarrollada	819,56	7573,21	9950,71
Diferencia relativa	0 %	0.36 %	0.85 %

Finalmente mostramos la comparación, para cada uno de los problemas, entre el tiempo de cálculo indicado en Gendreau *et al.* (1994) y el de nuestra implementación. Debe tenerse en cuenta que el tiempo utilizado depende esencialmente de la potencia y velocidad de los procesadores de los ordenadores empleados. Todos los tiempos se dan en segundos redondeados a dos decimales.

Problema	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Gendreau *et al.* (1994)	360	3228	1104	3528	54	810	3276	1536	4260	5988	1332	960	3552	3942
Implementación desarrollada	9,90	30,85	65,91	243,57	246,78	12,44	30,61	36,84	101,11	195,70	79,76	32,09	47,99	36,06

Bibliografía

D. L. Applegate, R. E. Bixby, V. Chvatal y W. J. Cook. *The traveling salesman problem: a computational study*. Princeton University Press, 2011.

M. L. Balinski y R. E. Quandt. On an integer program for a delivery problem. *Operations Research*, 12 (2):300–304, 1964.

N. Christofides, A. Mingozzi, P. Toth y C. Sandi. *Combinatorial optimization*. John Wiley, Chichester, 1979.

G. B. Dantzig y J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.

M. Gendreau. *An introduction to tabu search*. Springer, 2003.

M. Gendreau, A. Hertz y G. Laporte. New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40(6):1086–1094, 1992.

M. Gendreau, A. Hertz y G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276–1290, 1994.

F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1): 156–166, 1977.

F. Glover. Tabu search-part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

F. Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

F. Glover y M. Laguna. *Tabu Search*. Springer, 2013.

B. L. Golden, T. L. Magnanti y H. Q. Nguyen. Implementing vehicle routing algorithms. *Networks*, 7 (2):113–148, 1977.

M. Laguna. A guide to implementing tabu search. *Investigación Operativa*, 4(1):5–25, 1994.

G. Laporte. Fifty years of vehicle routing. *Transportation Science*, 43(4):408–416, 2009.

G. Laporte, Y. Nobert y M. Desrochers. Optimal routing under capacity and distance restrictions. *Operations Research*, 33(5):1050–1073, 1985.

G. Laporte, H. Mercure y Y. Nobert. An exact algorithm for the asymmetrical capacitated vehicle routing problem. *Networks*, 16(1):33–46, 1986.

D. Pecin, A. Pessoa, M. Poggi y E. Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. En Jon Lee y Jens Vygen, eds., *Integer Programming and Combinatorial Optimization*, páginas 393–403. Springer, 2014.

C. R. Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., 1993.

P. Toth y D. Vigo. *The vehicle routing problem*. Society for Industrial and Applied Mathematics, 2001.

P. Toth y D. Vigo. *Vehicle Routing: Problems, Methods, and Applications - Segunda edición*. Siam, 2014.

Siglas