FACULTADE DE MATEMÁTICAS

Trabajo Fin de Grado

Metaheurísticas del TSP
Un recorrido didáctico y computacional

Elena Fernández del Sel

2024-2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Trabajo Fin de Grado

Metaheurísticas del TSP
Un recorrido didáctico y computacional

Elena Fernández del Sel

Febrero, 2025

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Trabajo propuesto

Área de Coñecemento: Estadística e Investigación Operativa
Título: Metaheurísticas del TSP: un recorrido didáctico y computacional
Breve descrición do contido
El Problema del Viajante de Comercio (TSP, por sus siglas en inglés) ha sido un tema profundamente estudiado en el campo de la optimización combinatoria durante décadas. El objetivo de este TFG es ofrecer un recorrido amplio y accesible sobre las metaheurísticas más relevantes aplicadas al TSP. Durante el trabajo se analizarán tanto los fundamentos teóricos de estas técnicas como sus aplicaciones prácticas en la resolución eficiente del TSP, comparando el rendimiento de las distintas metaheurísticas y presentando conclusiones sobre sus ventajas y limitaciones.

Índice

Resumen ix

Introducción xi

1. Preliminares 1

1.1. Problema del viajante de comercio 1

1.1.1. Un poco de historia 1

1.1.2. Descripción 2

1.1.3. Tipos de TSP 2

1.1.4. Algunos algoritmos 3

1.2. Nociones sobre teoría de grafos 4

1.3. NP-completitud 8

1.3.1. Algoritmos de aproximación 15

2. Metaheurísticas 17

2.1. Búsqueda tabú 17

2.1.1. Descripción 17

2.1.2. Programación en R 21

2.1.3. Ejemplo ilustrativo 23

2.2. Templado simulado 25

2.2.1. Descripción 25

2.2.2. Programación en R 31
2.2.3. Ejemplo ilustrativo 32
2.3. Algoritmos genéticos 34
2.3.1. Descripción 34
2.3.2. Programación en R 40
2.3.3. Ejemplo ilustrativo 43
2.4. Optimización de la colonia de hormigas 46
2.4.1. Descripción 46
2.4.2. Programación en R 48
2.4.3. Ejemplo ilustrativo 50

3. Estudio computacional **53**
3.1. Formato de las pruebas 53
3.2. Resultados de las pruebas 55
3.2.1. Análisis de la complejidad algorítmica 55
3.2.2. Desviación respecto al coste óptimo 57

4. Conclusiones **59**

I. Código de R completo **61**
I.1. core.R 61
I.2. busquedaTabu.R 62
I.3. templadoSimulado.R 65
I.4. algoritmoGenetico.R 68
I.5. aco.R 71

II. Tablas **75**
II.1. Resultados búsqueda tabú 75

II.2. Resultados templado simulado 75

II.3. Resultados algoritmo genético 78

II.4. Resultados optimización de la colonia de hormigas 80

III. Exploración visual de parámetros en el problema del viajante 83

Resumen

Durante la historia de la computación, los problemas de rutas han suscitado un gran interés debido a sus múltiples aplicaciones en diferentes campos, como son la planificación y la logística. Este trabajo se centra en el problema del viajante de comercio o TSP. En concreto, en las técnicas para resolverlo de forma aproximada en un tiempo polinómico, las metaheurísticas. El objetivo principal de este estudio es proporcionar una guía para comprender cuatro de las más importantes, tanto en el ámbito teórico como en el computacional. Para ello, se realizó una revisión bibliográfica, encontrando información relevante de ellas y sintetizándola. Estas son: la búsqueda tabú, el templado simulado, el algoritmo genético y la optimización de la colonia de hormigas. Para la parte computacional, se realizaron implementaciones en R de todas las metaheurísticas y se evaluaron con distintas instancias de la librería TSPLIB. Como resultado, se obtuvo que no hay una metaheurística mejor que el resto en todos los aspectos. La búsqueda tabú y la optimización de la colonia de hormigas obtienen resultados muy prometedores en términos de distancia al coste óptimo; sin embargo, son temporalmente más costosas que las otras dos. El templado simulado obtiene unos resultados algo peores que los anteriores, pero de forma muy rápida. Por último, el algoritmo genético obtiene muy malos resultados en un tiempo, relativamente, aceptable. En conclusión, este trabajo sirve como guía a las personas que quieran comprender estos conceptos.

Abstract

During the history of computing, routing problems have attracted great interest due to their multiple applications in different fields, such as planning and logistics. This study focuses on the traveling salesman problem or TSP. Specifically, on the techniques to solve it in an approximate way in polynomial time, the metaheuristics. The main objective of this study is to provide a guide to understand four of the most important ones, both theoretically and computationally. For this

purpose, a literature review was performed, finding relevant information of them and synthesizing it. They are: tabu search, simulated annealing, genetic algorithm and ant colony optimization. For the computational part, R implementations of all metaheuristics were performed and evaluated with different instances of the TSPLIB library. As a result, it was obtained that there is no metaheuristic better than the rest in all aspects. Tabu search and ant colony optimization obtain very promising results in terms of distance to optimal cost, however, they are temporarily more expensive than the other two. Simulated annealing obtains somewhat worse results than the previous ones, but in a very fast way. Finally, the genetic algorithm obtains very bad results in a relatively acceptable time. In conclusion, this work serves as a guide to people who want to understand these concepts.

Introducción

El presente trabajo de fin de grado trata de proporcionar una guía para comprender las metaheurísticas más importantes que se aplican al TSP. Este problema trata de minimizar el coste de recorrer distintos nodos de un grafo, cada uno una única vez, y regresar al nodo inicial. El estudio del TSP, así como su solución puede ser abordado desde la matemática discreta junto con la investigación operativa, de forma que un grafo y un modelo matemático pueden ofrecer una representación completa del mismo. Este problema tiene múltiples aplicaciones en términos de planificación y logística, lo que le proporciona una gran importancia. En Applegate (2006), se detallan algunas aplicaciones. Una de ellas, es la necesidad de encontrar la forma óptima en la que un telescopio debe realizar la observación de elementos lejanos como planetas, galaxias y estrellas. En este problema los "nodos" serían los elementos que observar y el coste sería el tiempo que tardaría el telescopio en posicionarse para observarlo.

El TSP o problema del viajante es un problema muy sencillo de entender, pero computacionalmente intratable. A pesar de ello, existen distintos métodos específicos que son capaces de dar una solución del problema en un tiempo razonable para instancias pequeñas o grandes. Un ejemplo de estos métodos es el método de ramificación y acotación. Por su intratabilidad, a lo largo de la historia de la computación se han aplicado distintas metaheurísticas para obtener la solución óptima. El término metaheurística se refiere, según Gendreau y Potvin (2010), a cualquier procedimiento que emplee estrategias para superar la optimalidad local, especialmente aquellos que usan una o más estructuras de entornos para definir movimientos admisibles para pasar de una solución a otra, para construír o para destruir soluciones.

En este trabajo de fin de grado se analizarán algunas de las distintas metaheurísticas que existen para la resolución del problema del viajante de comercio. Asimismo, se realizará un estudio computacional con ellas para instancias medianas y grandes, con el objetivo de encontrar la metaheurística que mejor se comporta.

El objetivo principal del trabajo es obtener una guía didáctica para entender todas las metaheurísticas analizadas y aplicarlas a casos prácticos para mostrar su funcionamiento.

La estructura de trabajo se inicia con la sección de preliminares, donde se presentarán concep-

tos introductorios importantes para la buena comprensión del análisis posterior. A continuación, se describirán las metaheurísticas seleccionadas. Estas son, búsqueda tabú, templado simulado, algoritmos genéticos y la optimización de la colonia de hormigas. Para cada metaheurística se proporcionará una descripción detallada, con su programación en R. Además, se incluirá un ejemplo ilustrativo. En una última sección, se realizará un estudio de eficiencia para instancias medianas y grandes para obtener la que mejor funcione. Por último, en los anexos se incluye el código desarrollado, los resultados de los experimentos. Asimismo, se presenta una interfaz gráfica para visualizar los resultados obtenidos al aplicar el código desarrollado.

Capítulo 1

Preliminares

1.1. Problema del viajante de comercio

1.1.1. Un poco de historia

Históricamente, el problema del viajante de comercio ha suscitado un gran interés. En la literatura, el nombre se vio por primera vez en un informe de Julia Robinson publicado en 1949, llamado "On the Hamiltonian game (a travelling salesman problem)". Sin embargo, no introducía el término, sino que simplemente lo usaba. Según Applegate (2006), el origen del término es todavía un misterio, aunque la hipótesis más ampliamente aceptada dice que se introdujo sobre los años 30 en Princeton y a partir de ese momento los matemáticos empezaron a usarlo y a interesarse cada vez más por él.

El término TSP ya se usaba ampliamente hacia mediados de los 50. Sin embargo, la idea de este ya se usaba desde antes. Este hecho se puede comprobar en la existencia de una gran cantidad de libros, historias o canciones que hablan sobre él. Uno de los primeros ejemplos de *tour* en la literatura matemática se puede ver un artículo de Euler de 1757 en el que se hablaba de una solución al problema del caballo. Este problema consiste en encontrar el recorrido en el cual un caballo puede recorrer cada casilla del tablero de ajedrez únicamente una vez, volviendo a la casilla inicial. Euler proponía un *tour* para solucionar dicho problema.

Como se puede leer en Applegate (2006), a lo largo de los años se han propuesto varios algoritmos para resolver el TSP, por ejemplo, el algoritmo Held-Karp propuesto en 1962, que tiene una complejidad $O(n^2 \cdot 2^n)$. Hasta la actualidad la investigación de nuevos algoritmos, así como, la mejora en la capacidad de computación ha permitido la resolución de instancias cada vez más grandes del TSP. En 1990, Reinelt, introduce TSPLIB que contiene una gran cantidad de instancias del TSP para pruebas. Un hito importante fue la introducción del software Concorde

que permitió resolver instancias más grandes. Actualmente, a partir de metaheurísticas es posible resolver instancias enormes con un margen de error pequeño en un tiempo razonable.

1.1.2. Descripción

Como se indica en Reina et al. (2020), el problema del agente viajero, problema del viajero o *Traveling Salesman Problem* (TSP) consiste en minimizar la distancia de un recorrido, en el cual se deben visitar n ciudades, cada cual, una única vez, con la salvedad, de que la ciudad de inicio y la de destino deben ser la misma. Una forma de representar este problema es la que se indica en el Modelo (1.1)

$$
\begin{aligned}
\text{mín} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \\
\text{sujeto a:} \quad & \sum_{j=1}^{n} x_{ij} = 1, \quad i \in \{1, 2, \ldots, n\} \\
& \sum_{i=1}^{n} x_{ij} = 1, \quad j \in \{1, 2, \ldots, n\} \\
& \sum_{(i,j)\in M(X)} x_{ij} \leq |X| - 1, X \subsetneq \{1, ..., n\}
\end{aligned}
\tag{1.1}
$$

siendo n el número de ciudades, d_{ij} la distancia entre la ciudad i y la ciudad j, y x_{ij} una variable binaria que indica si el viajero se desplazará de la ciudad i a la j. La última condición representa que no haya subciclos, siendo $M(X) = \{(i, j) \in N \times N, i \neq j, i, j \in X\}$, donde N representa el conjunto de ciudades.

1.1.3. Tipos de TSP

Según se indica en Korte y Vygen (2002), existen dos tipos de TSP, antes de introducirlos necesitamos presentar la siguiente definición.

Definición 1.1. Un *circuito hamiltoniano* en un grafo G es un circuito que contiene a todos los vértices de G. Un grafo que contiene a un circuito hamiltoniano se llama *grafo hamiltoniano.*

En base a ello, los dos tipos de TSP son:

- **TSP métrico**: Dado un grafo completo, $G = (N, M)$, con función de coste $c : M \to \mathbb{R}^+$ tal que $c((i, j)) + c((j, k)) \geq c((i, k))$, para todo, $i, j, k \in N$, el objetivo es encontrar un circuito hamiltoniano en G de coste mínimo.

- **TSP euclidiano**: Dado un conjunto finito $N \subseteq \mathbb{R}^2$, $|N| \geq 3$, el objetivo es encontrar un circuito hamiltoniano en el grafo completo generado a partir de N, $G = (N, M)$, tal que la longitud $\sum_{(i,j)\in M} ||i - j||^2$ sea mínima.

1.1.4. Algunos algoritmos

En esta sección, analizaremos algunos algoritmos que se usan para encontrar la solución al problema del TSP.

Ramificación y acotación

Según Korte y Vygen (2002), el algoritmo de ramificación y acotación es un algoritmo que permite revisar todas las soluciones al problema sin tener que hacerlo solución a solución. Es el mejor método existente para encontrar la solución óptima del TSP.

El método se divide en dos fases. En primer lugar, la **ramificación** que consiste en dividir la región factible del problema original en conjuntos más pequeños. De esta forma, se construye un árbol mediante todas las soluciones parciales que surgen de la ramificación. En segundo lugar, la **acotación** que consiste en a partir de cada solución parcial encontrar un límite inferior en el coste de cualquier solución parcial que se está tratando en ese momento y una cota superior a partir de la mejor solución factible obtenida.

Algoritmo de Christofides

Como se puede leer en Korte y Vygen (2002), el algoritmo de Christofides presentado en 1976, es uno de los mejores algoritmos de aproximación conocidos para el TSP métrico. Este algoritmo tiene complejidad $O(n^3)$.

Definición 1.2. Un *árbol de cobertura* es un subgrafo que contiene todos los nodos del grafo y, además, es un árbol. Se dice que es mínimo cuando su coste es menor que el de cualquier otro árbol de cobertura.

El algoritmo de Christofides consiste en crear, inicialmente, un árbol de cobertura mínimo, al que llamaremos, T, a partir del grafo G que representa el problema. Llamaremos W al conjunto de vértices de grado impar en T.

Definición 1.3. El *grado del vértice* es el número de aristas conectadas a él.

A continuación, se obtiene el emparejamiento, E; de coste mínimo de los vértices de W (que son siempre una cantidad par).

Definición 1.4. Un *grafo euleriano* es aquel que contiene un circuito euleriano. Un *circuito euleriano* en un grafo G es un circuito cerrado que contiene cada arista.

Se considera el grafo formado por el conjunto de nodos original y los arcos de T y E, a partir del cual se puede obtener un circuito euleriano. Finalmente, basta considerar el circuito hamiltoniano embebido.

1.2. Nociones sobre teoría de grafos

Los casos que trata el TSP se pueden representar también mediante grafos. Por tanto, veremos los conceptos formales de la teoría de grafos con ejemplos aplicables a nuestro análisis. Se pueden consultar y ampliar los conceptos sobre grafos comentados en Ahuja et al. (1993).

Definición 1.5. Un *grafo* G es un par (N, M) consistente en un conjunto N de elementos llamados *nodos o vértices* y un conjunto M cuyos elementos representan *arcos o aristas*.

Según cómo sean los elementos de M podemos distinguir entre dos tipos de grafos:

- **Grafo dirigido u orientado:** Aquel en el que $M \subseteq N \times N$, es decir, los arcos son pares ordenados; el arco (i, j) empieza en el nodo i y termina en el nodo j. Se puede ver un ejemplo en la Figura 1.1.

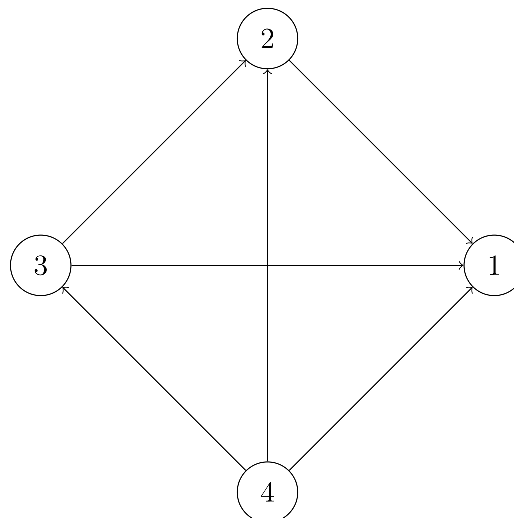


Figura 1.1: Grafo dirigido.

- **Grafo no dirigido:** Aquel que está compuesto por subconjuntos de N de dos elementos. En este caso, como (i, j) y (j, i) son el mismo conjunto, representarán el mismo arco. Se

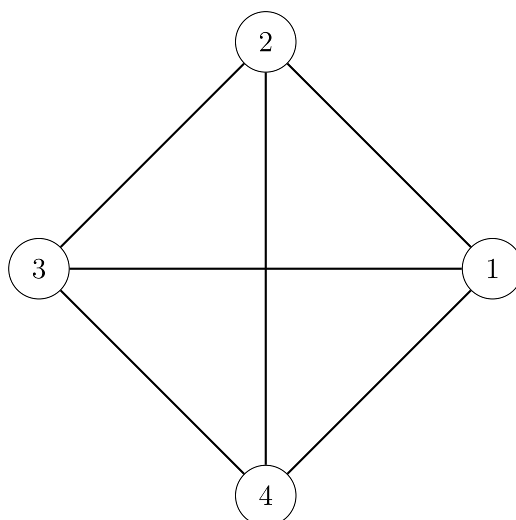


Figura 1.2: Grafo no dirigido.

puede ver un ejemplo en la Figura 1.2.

Usaremos n y m para referirnos al número total de nodos y aristas respectivamente.

Definición 1.6. Un *grafo completo* es aquel en el que todas las aristas posibles están presentes. En el caso de un grafo dirigido el máximo número de aristas es $n \cdot (n-1)$ (no permitimos lazos, por eso no es n^2) y, en el caso de un grafo no dirigido, tendremos la mitad, $n(n-1)/2$. La Figura 1.2 representa un grafo completo.

Definición 1.7. Un *subgrafo* de un grafo G es un grafo $G' = (N', M')$ que tiene todos sus vértices y aristas en G, es decir, $N' \subseteq N$ y $M' \subseteq M$. Un *subgrafo de expansión* de G es un subgrafo $G' = (N', M')$ tal que $N' = N$. Es decir, todos los nodos del grafo están en el subgrafo.

Si la arista (i, j) está presente en un grafo G, decimos que los nodos i y j son *adyacentes*. También, que son incidentes con la arista (i, j) y que la arista (i, j) es *incidente* con los nodos i y j. A partir de esto, podemos redefinir el concepto de grado como sigue.

Definición 1.8. El *grado* de un nodo cualquiera de un grafo G es el número de aristas incidentes con él.

Definición 1.9. Sea G un grafo no dirigido y sea $(a_1, a_2, \ldots, a_r)$ una secuencia de aristas distintas en G. Si existen vértices $(v_0, v_1, \ldots, v_r)$ tales que, para $l \in \{1, 2, \ldots, r\}, a_l = (v_{l-1}, v_l)$, decimos que la secuencia es una *cadena*. Para referirnos a una cadena usaremos indistintamente la secuencia de aristas o la secuencia de nodos que la forma. Una *cadena cerrada* es aquella en la que $v_0 = v_r$.

Definición 1.10. Un *camino* es una cadena en la que todos los vértices son distintos.

Definición 1.11. Un *circuito o ciclo* es una cadena cerrada en la que no hay más nodos coincidentes que el primero y el último.

Definición 1.12. Un grafo se dice que es *conexo* si para cada par de vértices existe una cadena no dirigida que los une. Se dice que es *fuertemente conexo* si para cada par de vértices existe una cadena dirigida que los une.

Definición 1.13. Un grafo se dice que es un *árbol* si es conexo y no contiene ciclos (no dirigidos).

Proposición 1.14. *Dado un grafo no dirigido $G = (N, M)$, las siguientes propiedades son equivalentes:*

1. *G es un árbol.*
2. *G es conexo y tiene $n - 1$ aristas.*
3. *G no tiene ciclos y tiene $n - 1$ aristas.*
4. *G no tiene ciclos y, si añadimos una arista cualquiera, se formará un ciclo (y sólo uno).*
5. *G es conexo y, si eliminamos una arista cualquiera, deja de ser conexo.*
6. *Cada par de nodos de G están unidos por un único camino.*

En los grafos presentes en el problema del viajante, las aristas tienen asociados valores que representan los costes asociados a ese recorrido concreto. Normalmente, estos costes representarán distancias, pero también podrían representar otros factores dependiendo de lo que se quiera minimizar. El TSP consiste en encontrar el circuito hamiltoniano que minimice los costes. En la Figura 1.3 podemos ver un ejemplo del TSP donde los nodos representan las 7 ciudades gallegas.

Sea $G = (N, M)$ un grafo dirigido con nodos dados por $N = \{1, 2, \dots, n\}$ y arcos por $M = \{a_1, a_2, \dots, a_m\}$. El grafo G puede ser representado por la denominada *matriz de adyacencia* $\bar{A}_{n\times n}$ definida por:

$$\bar{a}_{ij} = \begin{cases} 1 & \text{si } (i, j) \text{ es un arco de } G \\ 0 & \text{en otro caso.} \end{cases}$$

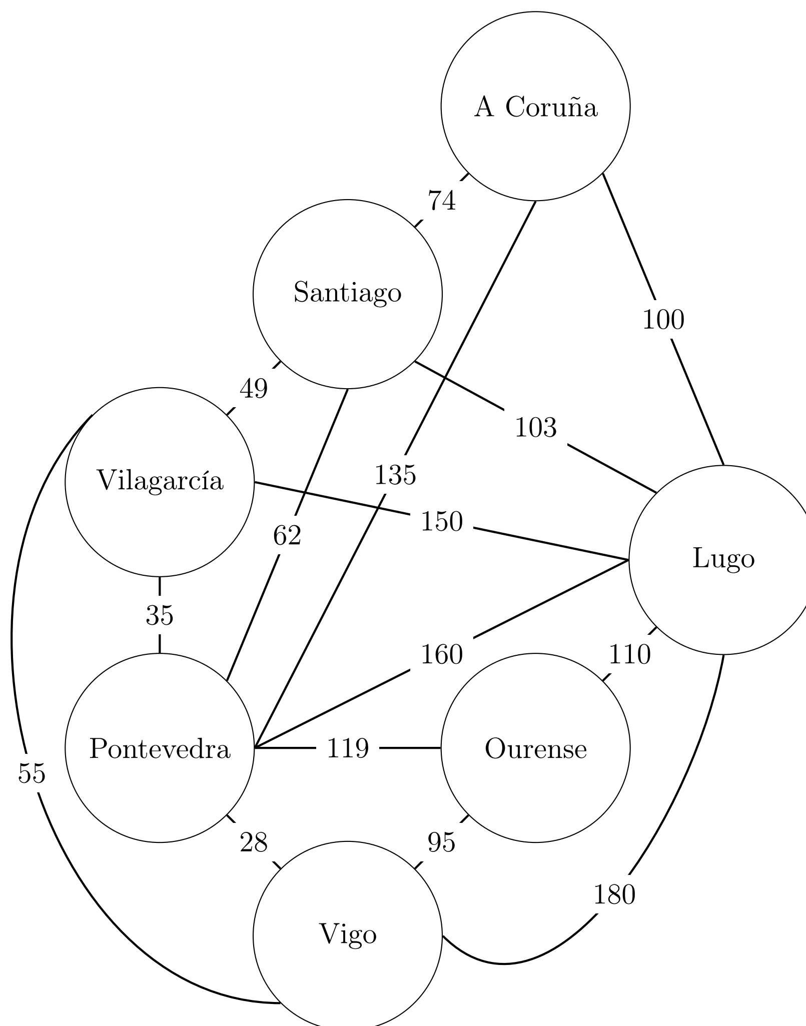


Figura 1.3: Ejemplo de un problema para resolver con el TSP.

Para el grafo de ejemplo de la Figura 1.3 la matriz de adyacencia sería la de la matriz (1.2).

$$A = \begin{pmatrix} & \textbf{LC} & \textbf{L} & \textbf{O} & \textbf{Vig} & \textbf{P} & \textbf{Vil} & \textbf{S} \\ \textbf{LC} & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \textbf{L} & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \textbf{O} & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \textbf{Vig} & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \textbf{P} & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ \textbf{Vil} & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ \textbf{S} & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}. \tag{1.2}$$

Otra posible representación matricial del grafo G es a través de la *matriz de incidencia* $\overline{\boldsymbol{B}}_{n\times m}$ definida como:

$$\bar{b}_{ik} = \begin{cases} 1 & \text{si } i \text{ es el nodo inicial de } a_k \\ -1 & \text{si } i \text{ es el nodo terminal de } a_k \\ 0 & \text{en otro caso.} \end{cases}$$

1.3. NP-completitud

Para fundamentar la complejidad computacional del TSP uno de los conceptos fundamentales es la NP-completitud. Todos los conceptos usados en esta sección se pueden ampliar en Korte y Vygen (2002).

Conceptos básicos sobre las máquinas de Turing

Definición 1.15. Un *alfabeto* A es un conjunto finito con al menos dos elementos. Este no contiene el carácter especial $\sqcup$ que representa un espacio en blanco.

Definición 1.16. Dado un alfabeto, A, denotamos por $A^* = \bigcup_{n \in \mathbb{Z}^+} A^n$, al conjunto de todas las cadenas finitas cuyos símbolos son elementos de A. Se suele tomar $A = \{0, 1\}$ y el conjunto de las cadenas binarias, cuyos componentes llamamos bits.

Definición 1.17. Un *lenguaje*, S, de A es un subconjunto de A^*. Los elementos del lenguaje se llaman *palabras*. Dada $x \in A^n$, llamamos tamaño$(x) := n$ a la longitud de la cadena.

Dado un alfabeto A, una máquina de Turing toma como entrada una palabra perteneciente a A^*. La entrada se completa con símbolos en blanco, $\sqcup$, para extenderla a una cadena infinita bidireccional $s \in (A \cup \{\sqcup\})^{\mathbb{Z}}$. Esta se escribirá sobre una cinta que funcionará como un dispositivo de almacenamiento. Como, por ejemplo, la que se muestra en la Figura 1.4.

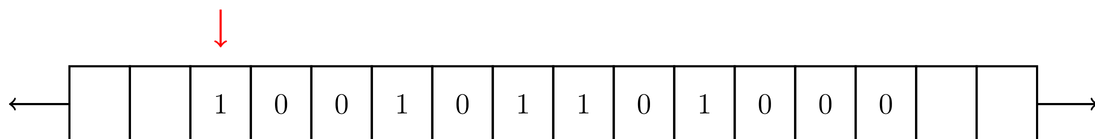


Figura 1.4: Cinta infinita bidireccional partiendo de 100101101000.

Además, de esta cinta, una máquina de Turing tiene un cabezal de lectura-escritura, que se representa en rojo en la Figura 1.4. Este cabezal se coloca en la posición que se está leyendo.

En la máquina de Turing se ejecutan un conjunto de sentencias. La sentencia inicial es la 0 y la posición inicial es la 1. Cada sentencia lee el bit en la posición actual, y dependiendo del valor y estado en el que se encuentre, realiza, simultáneamente, una de las siguientes operaciones:

- Sobrescribe el bit actual por algún elemento de $A \cup \{\sqcup\}$.
- Mueve la posición actual uno a la derecha o a la izquierda o se queda donde está.

Si lee -1, finaliza el cómputo. La salida es la cadena que queda escrita en la cinta, en las posiciones que van desde la 1 hasta el primer carácter en blanco. Con un sistema tan sencillo se puede computar cualquier cosa.

Formalmente, la idea anterior se puede definir de la siguiente manera:

Definición 1.18. Sea A un alfabeto y $\bar{A} := A \cup \{\sqcup\}$. Una *máquina de Turing* (con alfabeto A) está definida por una función

$$\Phi : \{0, \ldots, N\} \times \bar{A} \to \{-1, \ldots, N\} \times \bar{A} \times \{-1, 0, 1\}$$

para algún $N \in \mathbb{Z}_+$.

La computación de Φ con entrada x, donde $x \in A^*$, es la secuencia finita o infinita de triples $(n^{(i)}, s^{(i)}, \pi^{(i)})$ con $n^{(i)} \in \{-1, \ldots, N\}, s^{(i)} \in \bar{A}^{\mathbb{Z}}$ y $\pi^{(i)} \in \mathbb{Z}$ (para $i = 0, 1, 2, \ldots$) definida recursivamente como sigue (donde $n^{(i)}$ denota el estado actual, $s^{(i)}$ representa la cadena, y $\pi^{(i)}$ es la posición actual):

$$\begin{aligned} n^{(0)} &:= 0, \\ s_j^{(0)} &:= x_j \text{ para } 1 \leq j \leq \text{tamaño}(x), \\ s_j^{(0)} &:= \sqcup \text{ para todo } j \leq 0 \text{ y } j > \text{tamaño}(x), \\ \pi^{(0)} &:= 1. \end{aligned}$$

Si $(n^{(i)}, s^{(i)}, \pi^{(i)})$ ya está definido, distinguimos dos casos.

- Si $n^{(i)} \neq -1$, entonces definimos $(m, \sigma, \delta) := \Phi(n^{(i)}, s_{\pi^{(i)}}^{(i)})$ y definimos:

$$\begin{aligned} n^{(i+1)} &:= m, \\ s_{\pi^{(i)}}^{(i+1)} &:= \sigma, \\ s_j^{(i+1)} &:= s_j^{(i)} \text{ para } j \in \mathbb{Z} \setminus \{\pi^{(i)}\}, \\ \pi^{(i+1)} &:= \pi^{(i)} + \delta. \end{aligned}$$

- Si $n^{(i)} = -1$, entonces esto es el final de la secuencia. Definimos $\mathit{tiempo}(\Phi, x) := i$ y $\mathit{salida}(\Phi, x) \in A^k$, donde $k := \min\{j \in \mathbb{N} : s_j^{(i)} = \sqcup\} - 1$, por $\mathit{salida}(\Phi, x)_j := s_j^{(i)}$ para $j = 1, \ldots, k$.

Si la secuencia es infinita (es decir, $n^{(i)} \neq -1$ para todo i), entonces definimos $\textit{tiempo}(\Phi, x) := \infty$. En este caso, $\textit{salida}(\Phi, x)$ no está definida.

Para entender mejor este concepto, veamos un ejemplo concreto,

Ejemplo 1.19. Dado el alfabeto $A = \{0, 1\}$, este ejemplo mostrará el funcionamiento de una máquina de Turing que suma dos números en formato unario que están separados por un 0 y da como salida la suma seguida de un 0. La función:

$$\Phi : \{0, 1, 2\} \times \bar{A} \to \{-1, 0, 1, 2\} \times \bar{A} \times \{-1, 0, 1\}.$$

Tendrá el funcionamiento que se observa en la Figura 1.5. Los nodos representan los diferentes estados. Las aristas representan el estado de destino y el de origen. El triple de cada arista representa el valor que se lee, el valor que se escribe y el valor del movimiento, respectivamente.

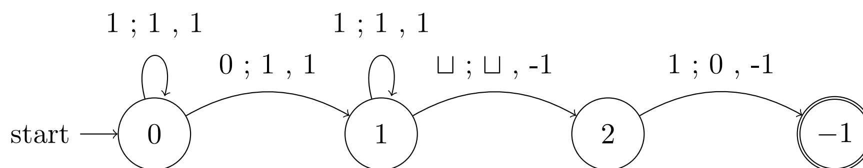


Figura 1.5: Autómata de máquina de Turing.

Sea $x = 101$, la cinta al inicio estaría como se muestra en la Figura 1.6.

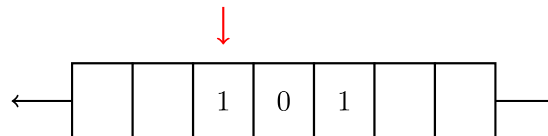


Figura 1.6: Cinta al inicio.

El triple de la definición al inicio sería, $(n^{(0)}, s^{(0)}, \pi^{(0)}) = (0, \sqcup\sqcup 101 \sqcup\sqcup, 1)$.

A partir de la función Φ surge la siguiente secuencia de acciones.

1. **Primer paso.** La máquina está en el estado $n(0) = 0$ y la cabeza de lecto-escritura está en la posición $\pi(0) = 1$. En la posición 1 la cabeza lee un 1. Por tanto, como $n^{(0)} \neq -1$, entonces, $\Phi(0, 1) = (0, 1, 1)$. Con lo que, $n^{(1)} = 0$, $\pi^{(1)} = 2$ y $s^{(1)}$ se mantiene como se muestra en la Figura 1.7.

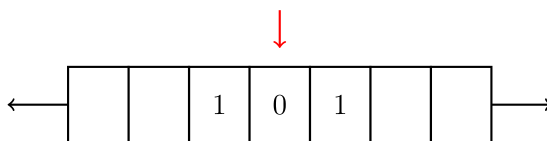


Figura 1.7: Cinta tras el primer paso.

2. **Segundo paso.** La máquina está en el estado 0. La cabeza de lecto-escritura está en la posición 2, donde lee un 0, es decir, ha terminado de leer el primer sumando. Como $n^{(1)} \neq -1$, entonces, $\Phi(0,0) = (1,1,1)$. Con lo que, $n^{(2)} = 1$, $\pi^{(2)} = 3$ y $s^{(2)}$ se mantiene como como se muestra en la Figura 1.8.

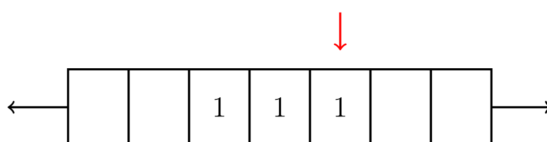


Figura 1.8: Cinta tras el segundo paso.

3. **Tercer paso**. La máquina está en el estado 1. La cabeza de lecto-escritura está en la posición 3, donde lee un 1. Como $n^{(2)} \neq -1$, entonces, $\Phi(1,1) = (1,1,1)$. Con lo que, $n^{(3)} = 1$, $\pi^{(3)} = 4$ y $s^{(3)}$ se mantiene como se muestra en la Figura 1.9.

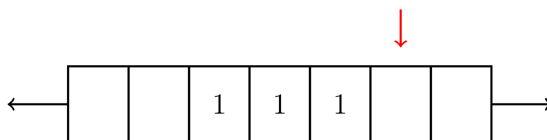


Figura 1.9: Cinta tras el tercer paso.

4. **Cuarto paso**. La máquina está en el estado 1. La cabeza de lecto-escritura está en la posición 4, donde lee un $\sqcup$. Como $n^{(3)} \neq -1$, entonces, $\Phi(1,\sqcup) = (2,\sqcup,-1)$. Con lo que, $n^{(4)} = 2$, $\pi^{(4)} = 3$ y $s^{(4)}$ se mantiene como se muestra en la Figura 1.10.

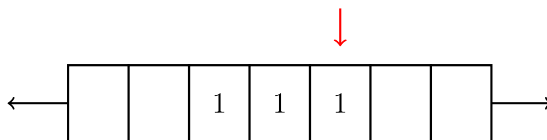


Figura 1.10: Cinta tras el cuarto paso.

5. **Quinto paso**. La máquina está en el estado 2. La cabeza de lecto-escritura está en la posición 3, donde lee un 1. Como $n^{(4)} \neq -1$, entonces, $\Phi(2,1) = (-1,0,-1)$. Con lo que, $n^{(5)} = -1$, $\pi^{(5)} = 2$ y $s^{(5)}$ se mantiene como como se muestra en la Figura 1.11.

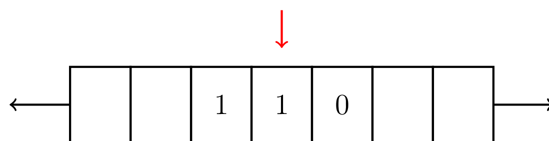


Figura 1.11: Cinta tras el quinto paso.

6. **Último paso**. La máquina está en el estado -1. Como $n^{(5)} = -1$, estamos ante el final de la secuencia. Con $tiempo(\Phi, x) = 5$ y $salida(\Phi, x) = 110$.

Definición 1.20. Sea A un alfabeto, $S, T \subseteq A^*$ dos lenguajes, y $f : S \to T$ una función. Dada la máquina de Turing con alfabeto A tal que $tiempo(\Phi, x) < \infty$ y $salida(\Phi, x) = f(x)$, para cada, $x \in S$. Entonces diremos que Φ *computa* f.

Definición 1.21. Si existe un polinomio p, tal que, para todo $x \in S$ tenemos que $tiempo(\Phi, x) \leq p(\text{tamaño}(x))$, entonces, estaremos ante una *máquina de Turing de tiempo polinómico.*

Definición 1.22. En el caso en el que $S = A^*$ y $T = \{0, 1\}$, diremos que Φ *decide* el lenguaje $L := \{x \in S : f(x) = 1\}$.

Definición 1.23. Si existe alguna máquina de Turing de tiempo polinómico computando la función f (o decidiendo un lenguaje L), entonces diremos que f es *computable en tiempo polinómico* (o que *L es decidible en tiempo polinómico*).

Definición 1.24. Un *problema de decisión* es un par $\mathcal{P} = (X, Y)$, donde X es un lenguaje decidible en tiempo polinómico y $Y \subseteq X$. Los elementos de X se llamarán instancias de $\mathcal{P}$; los elementos de Y son instancias de tipo sí y los elementos pertenecientes a $X \backslash Y$ son instancias tipo no. Un algoritmo para el problema de decisión es aquél que computa la función (1.3) definida como sigue:

$$\begin{aligned} f : X \to &\{0, 1\} \\ f(x) := &\begin{cases} 1, & x \in Y \\ 0, & x \in X \backslash Y. \end{cases} \end{aligned} \tag{1.3}$$

Definición 1.25. Un *problema de optimización* es un par (Ω, f), donde Ω es el espacio de soluciones y $f : \Omega \to \mathbb{R}$ es la función de coste. El objetivo del problema es encontrar una solución $\omega^* \in \Omega$ tal que, para todo $\omega \in \Omega$, $f(\omega^*) \leq f(\omega)$. Es decir, $\omega^* = \arg\min_{\omega \in \Omega} f(\omega)$. Además, denotaremos por f^* al valor de la función de coste de ω^*.

Definición 1.26. Un *algoritmo de optimización* para un problema de decisión $\mathcal{P}$ es un algoritmo $\mathcal{A}$ que computa para cada entrada $x \in X$ una solución factible, cuando existe dicha solución para la entrada x. Si la solución es la óptima para cualquier entrada, entonces, $\mathcal{A}$ es un algoritmo exacto.

Clases de complejidad

Presentemos ahora, el concepto de clases de complejidad.

Definición 1.27. La *clase* **P** es aquella que engloba a todos los problemas de decisión para los cuales hay un algoritmo que los compute en tiempo polinómico.

Definición 1.28. Un problema de decisión $P = (X, Y)$ pertenece a la *clase NP* si existe un polinomio p y un problema de decisión $P' = (X', Y')$ tal que

$$X' := \{x\#c : x \in X, c \in \{0,1\}^{\lfloor p(\text{tamaño}(x))\rfloor}\},$$

de modo que

$$Y = \{y \in X : \text{Existe una cadena } c \in \{0,1\}^{\lfloor p(\text{tamaño}(y))\rfloor} \text{ tal que } y\#c \in Y'\}.$$

Aquí, $x\#c$ denota la concatenación de la cadena x, el símbolo $\#$ y la cadena c. Una cadena c con $y\#c \in Y'$ se denomina *certificado* para y (ya que c prueba que $y \in Y$). Un algoritmo para P' se llama *algoritmo de verificación de certificados.*

Dado un problema, decimos que pertenece a la clase de complejidad NP, si existe un algoritmo que pueda verificar cualquier solución a un ejemplo en dicha clase en tiempo polinomial. Para entender esto, nos podemos fijar en un sudoku. Resolver un sudoku desde cero es costoso, sin embargo, una vez que ya tenemos la solución comprobarla es muy sencillo.

Definición 1.29. En el estudio de las distintas clases de complejidad resulta de especial utilidad transformar unos problemas en otros. Supongamos que un problema $\mathcal{P}_1$ es tal que cualquier ejemplo en dicha clase se puede trasformar en tiempo polinomial en un ejemplo de la clase $\mathcal{P}_2$. En este caso se dice que $\mathcal{P}_1$ *se reduce a* $\mathcal{P}_2$ *en tiempo polinomial.* Si la clase $\mathcal{P}_2$ pertenece a P, entonces también la clase $\mathcal{P}_1$ pertenece a P. Dado un elemento de $\mathcal{P}_1$ lo podremos resolver en tiempo polinomial sin más que transformarlo previamente (en tiempo polinomial) en un elemento de $\mathcal{P}_2$.

Definición 1.30. Sean $\mathcal{P}_1 = (X_1, Y_1)$ y $\mathcal{P}_2 = (X_2, Y_2)$ dos problemas de decisión. Se dice que $\mathcal{P}_1$ *se transforma de forma polinómica* en $\mathcal{P}_2$ si existe una función $f : X_1 \to X_2$ computable en tiempo polinómico tal que $f(x_1) \in Y_2$ para todo $x_1 \in Y_1$ y $f(x_1) \in X_2 \backslash Y_2$ para todo $x_1 \in X_1 \backslash Y_1$.

Definición 1.31. Un problema $\mathcal{P}_1 \in$ **NP** es *NP-completo* si cualquier problema en NP se puede reducir a $\mathcal{P}_1$ en tiempo polinomial. Por tanto, dado un algoritmo para el problema $\mathcal{P}_1$, puede ser adaptado para resolver en un tiempo extra polinomial cualquier otro problema de la clase NP. Entonces, si apareciese algún algoritmo polinomial para resolver un problema NP-completo, tendríamos que cualquier problema en NP se puede resolver también en tiempo polinomial, lo que implicaría que P = NP.

Definición 1.32. Un problema es NP-duro si cualquier problema en la clase NP se puede reducir a él en tiempo polinomial.

La diferencia entre las clases NP-completo y NP-duro es que un problema puede ser NP-duro sin pertenecer a la clase NP. En la Figura 1.12 se puede ver la relación entre las clases P, NP, NP-duro y NP-completo.

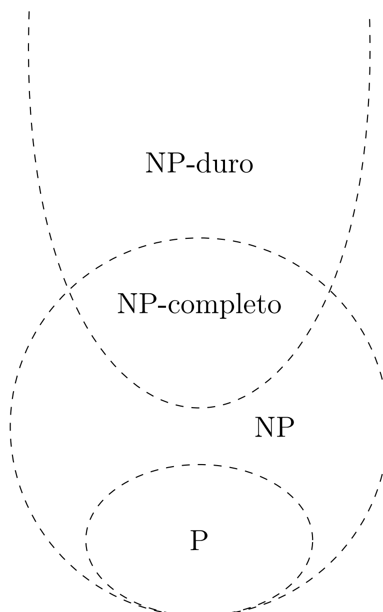


Figura 1.12: Relaciones entre las clases de complejidad.

Complejidad del TSP

Definición 1.33. El *problema del circuito hamiltoniano* es un problema que tiene como objetivo determinar si un grafo G contiene un circuito hamiltoniano.

El problema anterior es NP-duro.

Teorema 1.34. *El TSP es NP-duro.*

Demostración. La prueba se basa en describir una transformación polinómica desde el problema del circuito hamiltoniano.

Para ello tenemos un grafo, $G = (N, M)$, de n vértices. Con él, construiremos una instancia del TSP con las siguientes características.

- Cada vértice de G es una ciudad en el TSP.

- Las distancias serán 1 cuando las aristas estén en M y 2 en otro caso.

Por tanto, es trivial ver que G es hamiltoniano, si y solo si, el recorrido óptimo del TSP tiene longitud n. □

1.3.1. Algoritmos de aproximación

Los algoritmos de optimización se pueden clasificar en dos grandes grupos:

- Algoritmos de optimización exactos: garantizan la obtención de la solución óptima para todas las entradas en las que exista solución factible.
- Algoritmos de optimización metaheurísticos o aproximados: No garantizan la obtención de la solución óptima para todas las entradas en las que exista solución factible.

En Korte y Vygen (2002), encontramos el siguiente concepto:

Definición 1.35. Sea $\mathcal{P}$ un problema de optimización, un *algoritmo de aproximación de factor* k para $\mathcal{P}$ es un algoritmo de tiempo polinómico A para $\mathcal{P}$ tal que:

$$\frac{1}{k}\text{OPT}(I) \leq A(I) \leq k\text{OPT}(I)$$

para todas las instancias I de $\mathcal{P}$, siendo $\text{OPT}(I)$, el valor objetivo de la solución óptima. $A(I)$ representa el valor objetivo para la solución que da A.

Teorema 1.36. *A no ser que P=NP, no hay ningún algoritmo de aproximación de factor* $k \geq 1$ *para el TSP.*

Demostración. Supongamos que existe un algoritmo de aproximación de factor k, A, para el TSP. Entonces, tendría que existir un algoritmo de tiempo polinómico para el problema del circuito hamiltoniano, lo cual, ya que este problema es NP-completo querría decir que P = NP.

Dado un grafo $G = (L, T)$, construiremos una instancia del TSP, a la que llamaremos K_n que será un grafo completo (N, M), siendo $N = n$ el número de ciudades. Las distancias están definidas como $c : M \to \mathbb{Z}_+$,

$$c((i,j)) := \begin{cases} 1 & \text{si } (i,j) \in T \\ 2 + (k-1)n & \text{si } (i,j) \notin T. \end{cases}$$

Aplicamos el algoritmo A a esta instancia. Y tenemos dos opciones:

- Si el recorrido obtenido tiene longitud n, entonces, este camino es un circuito hamiltoniano en G.
- Si el camino obtenido tiene longitud mayor, tendrá longitud de al menos $n+1+(k-1)n = kn+1$.

Llamamos $OPT(K_n, c)$ a la longitud del camino óptimo, entonces,

$$\frac{kn+1}{OPT(K_n, c)} \leq k,$$

por ser A un algoritmo de aproximación de factor k. Entonces,

$$OPT(K_n, c) \geq \frac{kn+1}{k}.$$

Por tanto, $OPT(K_n, c) > n$, concluyendo que G no tiene circuitos hamiltonianos. Esto es una contradicción con el hecho de que existe un algoritmo de tiempo polinómico para el problema del circuito hamiltoniano. Dicha contradicción viene de suponer que existe un algoritmo de aproximación de factor k para el TSP. □

Capítulo 2

Metaheurísticas

El término metaheurística fue acuñado en Glover (1986). En Gendreau y Potvin (2010) se presenta un profundo análisis de todas las metaheurísticas que se tratarán durante este trabajo.

2.1. Búsqueda tabú

2.1.1. Descripción

Motivación

La búsqueda tabú es una metaheurística muy popular que se propuso originalmente en Glover (1986). Esta vino motivada por el desarrollo de la teoría de la complejidad a principios de la década de 1970, donde se reconoce la existencia de muchos problemas NP-duros. A partir de aquí, se empiezan a buscar maneras de encontrar soluciones a estos problemas. Para ello, el enfoque que más destacó inicialmente fue la búsqueda local. Consistía en que partiendo de una solución inicial factible se le aplicaba una mejora iterativamente hasta llegar a un óptimo local. Esta tenía un problema, la solución que se obtenía solía ser mediocre. En 1983 aparece el templado simulado. Este suceso da lugar a una época en la que se tienden a crear algoritmos que se basan en imitar fenómenos naturales.

Funcionamiento

La idea principal de la búsqueda tabú es mejorar la búsqueda local, explorando el entorno de la solución actual estratégicamente y permitiendo visitar nodos que no mejoren la solución. Esto suscita un problema, se puede volver a soluciones ya visitadas. Para evitarlo, se propuso

utilizar una pequeña memoria que almacenase los movimientos tabús, es decir, aquellos que no se pueden hacer para no volver a una solución ya visitada.

Una vez definido el problema con sus variables, restricciones y función de coste, el espacio de búsqueda considerado se conforma por el espacio de todas las posibles soluciones factibles. Si se aplicase una relajación de las restricciones, también, se añadirían algunas soluciones infactibles.

Definición 2.1. Un entorno de una solución ω, $N(\omega)$, es el subconjunto del espacio de búsqueda, Ω, que se obtiene al realizar una única transformación a la solución ω.

Tras seleccionar una estructura de entornos adecuada, hace falta escoger el mejor de los elementos pertenecientes a la misma. Para ello, entran en juego diferentes conceptos. El primero, es la principal diferencia entre la búsqueda local y la búsqueda tabú, la existencia de **tabús**. Como ya se describió anteriormente, estos se usan para prevenir ciclos, así como para realizar una búsqueda más extensa alejándose de zonas del espacio de búsqueda ya exploradas. Su funcionamiento se basa en una o varias listas tabús, T, estas son, listas de los movimientos que invertirían los efectos de los movimientos recientes. Para almacenar dichas listas se recurre a una memoria a corto plazo con una capacidad limitada. En un escenario ideal, se podrían guardar las soluciones completas, sin embargo, esto haría muy costosas las búsquedas. Por esta razón, comúnmente se opta por prohibir únicamente las inversas de las últimas transformaciones para la solución actual, ω. Además, lo más habitual, es que estas listas se actualicen a medida que se avanza en la ejecución del algoritmo.

Los tabús, aunque pieza central del algoritmo, albergan problemas. Algunos son, la posibilidad de prohibir movimientos que no tendrían peligro de crear ciclos o la opción de paralizar el algoritmo. Para solucionarlos, se introduce el concepto de *aspiration criteria* o criterios de aspiración. Estos tienen la habilidad de rescindir tabús. Por ejemplo, un criterio de aspiración, usado con asiduidad, es aquel que admite un movimiento, a condición de que proporcione como resultado una solución con una función objetivo que supere la óptima obtenida hasta el momento.

Otro aspecto crucial en este algoritmo es la existencia de un criterio de parada. Si no se dispone del mismo, el algoritmo podrá computarse infinitamente. Los criterios de parada más comunes son:

- Limitar el número de iteraciones o el tiempo de CPU.
- Limitar el número de iteraciones en el que no mejora el valor de la función objetivo.
- Establecer un valor prefijado para que el algoritmo pare cuando la función objetivo lo alcanza.

Técnicas avanzadas

Para cada problema hay que adaptar el algoritmo. Para ello, se puede recurrir a técnicas avanzadas como son la intensificación, la diversificación o la lista de candidatos.

En primer lugar, tener que comparar uno a uno, todos los valores de soluciones presentes en el entorno de la solución, es computacionalmente muy costoso. Por ello, se propone considerar solo una lista de candidatos $N'(\omega)$ que es un subconjunto de $N(\omega)$. Esta puede ser obtenida de forma aleatoria o con estrategias más complejas. Una adecuada lista de candidatos puede mejorar considerablemente la efectividad de la búsqueda.

En segundo lugar, el concepto de intensificación consiste en ver que componentes de la solución estuvieron en ella durante más tiempo y aprovecharlos para encontrar las zonas del espacio de búsqueda más prometedoras.

En tercer lugar, tenemos el concepto de diversificación. La diversificación es una técnica que sirve para evitar que la búsqueda sea demasiado local, mejorando así la calidad de las soluciones. Esta técnica consiste en posicionar la búsqueda en zonas inexploradas del espacio de búsqueda. Hay tres formas de realizarlas.

- Diversificación de reinicio:

  Definición 2.2. Denominamos *diversificación de reinicio* a la técnica que consiste en reiniciar la búsqueda a partir de un componente de la solución raramente usado.

  Por ejemplo, tomemos una instancia del TSP con 5 ciudades, $[A, B, C, D, E]$. Aplicamos el algoritmo de la búsqueda tabú y observamos que las soluciones que se van obteniendo no varían mucho de la solución inicial. En ese momento, se aplica diversificación de reinicio y se empieza la búsqueda desde una solución muy distinta a las que se han estado tratando, digamos $[A, E, C, D, B]$. De esta manera, sacaremos la búsqueda de la zona del espacio en la que estaba y exploraremos soluciones completamente inexploradas.

- Diversificación continua:

  Definición 2.3. Denominamos *diversificación continua* a la técnica que consiste en añadir, a la evaluación de los posibles movimientos, un pequeño termino relacionado con la frecuencia de los componentes.

  Tomemos el ejemplo anterior. Después de un cierto número de iteraciones el algoritmo registra que la conexión $A \to B$ se registra muchas veces en las soluciones tratadas. Al detectar esto el algoritmo penaliza a esta conexión, por ello será más difícil que se seleccione. El proceso derivará en que se priorice la exploración de conexiones poco usadas.

En cuarto lugar, cabe destacar el concepto de relajación de las restricciones. Este concepto se basa en que restringir la búsqueda únicamente a soluciones factibles puede acotar demasiado el espacio de búsqueda. Su principal ventaja radica en que, para algunos problemas, un espacio de búsqueda más completo facilita la búsqueda. Para aplicar estas relajaciones eliminan algunas restricciones y se añade a la función objetivo ciertas penalizaciones para las restricciones que se incumplen. Surge la pregunta de cómo se debe penalizar el incumplimiento de una restricción. Una manera muy recomendada es la aplicación de las penalizaciones autoajustadas, esto es, ajustar las penalizaciones de la función objetivo dinámicamente. Si entre las soluciones recientes hay muchas soluciones infactibles se penalizará más las soluciones infactibles, es decir, su coste será mayor. En cambio, si las soluciones recientes eran en su mayoría factibles, la función de coste penalizará poco a las soluciones infactibles. Este sistema se puede usar también para aplicar la diversificación, en cuyo caso, se llama oscilación estratégica.

En último lugar, se presentan los conceptos de función objetivo sustituta y función objetivo auxiliar. La función objetivo sustituta es una función relacionada con la función objetivo y con un coste computacional menor. Esta permite conseguir una lista de soluciones prometedora. Posteriormente, esta lista se evalúa con la función objetivo original. Su objetivo es restringir el uso de funciones computacionalmente muy exigentes. Las funciones objetivo auxiliares se utilizan para conseguir información adicional a la que nos proporciona la función objetivo. Esta función se encarga de evaluar los atributos deseables de la solución.

Algoritmo 1 Algoritmo de Búsqueda Tabú. (Gendreau & Potvin, 2010)

```
1: Inicialización:
2: Elegir (construir) una solución inicial ω0
3: Establecer ω ← ω0, f* ← f(ω0), ω* ← ω0, T ← ∅
4: Búsqueda:
5: while criterio de terminación no satisfecho do
6:     ωaux ← null
7:     for ω' ∈ N(ω) do
8:         if (ω' no está en T) or (ω' está en T and f(ω') < f*) then
9:             if f(ω') < f(ωaux) then
10:                ωaux ← ω'
11:            end if
12:        end if
13:    end for
14:    ω ← ωaux
15:    if f(ω) < f* then
16:        f* ← f(ω)
17:        ω* ← ω
18:    end if
19:    Registrar el movimiento actual en la lista tabú T
20:    Eliminar la entrada más antigua de T si es necesario
21: end while
```

2.1.2. Programación en R

En esta parte, se detallará cómo implementar la búsqueda tabú, previamente descrita, con el lenguaje de programación R.

La programación continúa con el esquema descrito en el Algoritmo 1. Se comienza con la inicialización de la solución. Inicialmente, se consideró una solución aleatoria, pero con el fin de orientar mejor la búsqueda desde el inicio, se optó por emplear una técnica voraz. Así se procede.

```r
solucion <- c(1)
ciudadActual <- 1
visitadas <- logical(numCiudades)
visitadas[ciudadActual] <- TRUE
```

```
for(i in 2:numCiudades){
  distancias <- matrizDistancias[ciudadActual, ]
  distancias[visitadas] <- Inf
  ciudadActual <- which.min(distancias)

  solucion[i] <- ciudadActual
  visitadas[ciudadActual] <- TRUE
}
```

Un aspecto crucial en la implementación del algoritmo es cómo se define la función de coste de la solución y la representación de las soluciones. En la formulación del problema del TSP que se muestra en la ecuación (1.1), se emplea una matriz. En esta matriz, las aristas utilizadas en el grafo se marcan con unos y todos los demás elementos son ceros. Para hacer el código más sencillo, se ha optado por representar la solución mediante un vector donde cada componente representa una ciudad. Así, para calcular el valor correspondiente a una arista en la solución, basta con revisar el vector en la posición $(i, i+1)$. Finalmente, evaluar la solución implica simplemente sumar todas estas distancias, recordando incluir el regreso a la ciudad inicial.

```
indices <- cbind(solucion, c(solucion[-1], solucion[1]))

return(sum(matrizDistancias[indices]))
```

Asimismo, es importante subrayar el método empleado para la creación de vecindarios. Aunque hay muchas técnicas disponibles, se optó por utilizar la técnica de intercambio de ciudades. Esta técnica consiste en seleccionar aleatoriamente dos ciudades y permutarlas, lo que produce una solución vecina.
Para la representación de la lista tabú se ha empleado una lista. En ella, cada movimiento se denota como i_j, donde i y j indican las ciudades intercambiadas para crear el vecino. Además, cada elemento tiene un valor asociado que indica el tiempo que permanecerá en la lista tabú.

```
listaTabu[[mejorMovimiento]] <- permanenciaListaTabu
```

Además, se emplean criterios de aspiración, que verifican si el coste del mejor vecino, aunque esté en la lista tabú, supera al mejor coste alcanzado hasta ahora.
El algoritmo cuenta con dos condiciones de finalización. La primera detiene la ejecución después de un número determinado de iteraciones. La segunda previene iteraciones redundantes al detener la ejecución tras un número especificado de iteraciones sin mejora en el mejor coste. El código completo se encuentra disponible en el Anexo I.

2.1.3. Ejemplo ilustrativo

En esta sección, demostraremos cómo aplicar la búsqueda tabú a un ejemplo del problema del viajante (TSP) con pocas ciudades. Para ello, utilizaremos el grafo 1.3 y lo convertiremos en una matriz de distancias completa, que se presentará en la ecuación siguiente.

$$A = \begin{pmatrix} & \textbf{LC} & \textbf{L} & \textbf{O} & \textbf{Vig} & \textbf{P} & \textbf{Vil} & \textbf{S} \\ \textbf{LC} & 0 & 100 & 174 & 158 & 135 & 125 & 74 \\ \textbf{L} & 100 & 0 & 110 & 180 & 160 & 150 & 108 \\ \textbf{O} & 174 & 110 & 0 & 95 & 119 & 151 & 105 \\ \textbf{Vig} & 158 & 180 & 95 & 0 & 28 & 55 & 88 \\ \textbf{P} & 135 & 160 & 119 & 28 & 0 & 35 & 62 \\ \textbf{Vil} & 125 & 150 & 151 & 55 & 35 & 0 & 49 \\ \textbf{S} & 74 & 108 & 105 & 88 & 62 & 49 & 0 \end{pmatrix} \tag{2.1}$$

Es trivial ver que el coste óptimo es 491. Además, la solución óptima está representada en rojo en el grafo de la Figura 2.1. En el grafo, los nodos describen las siete ciudades gallegas, conectados mediante aristas que indican las distancias aproximadas en kilómetros entre ciudades. Se han excluido algunas aristas para simplificar la representación gráfica del grafo. No obstante, la matriz de distancias se mantiene completa para el problema. Este problema se ha escogido debido a su fácil visualización y comprensión.
A continuación, exploraremos cómo se aplica el algoritmo de búsqueda tabú en este contexto. Los parámetros considerados son la matriz de distancias (2.1), un máximo de 100 iteraciones, hasta 10 iteraciones sin mejora en el mejor coste, y 5 iteraciones durante las cuales un movimiento permanece en la lista tabú.
Para empezar, se creará una solución inicial utilizando un algoritmo voraz, proporcionando el recorrido $[1, 7, 6, 5, 4, 3, 2]$. En este caso, el algoritmo voraz ya ha llegado a la solución óptima, lo cual no siempre sucede, por eso se continúa iterando. Por ejemplo, al aplicar la matriz de distancias (2.2), el coste de la solución inicial es 28, mientras que el óptimo es 26.

$$\text{Matriz de distancias} = \begin{bmatrix} 0 & 2 & 9 & 10 & 7 \\ 2 & 0 & 6 & 4 & 3 \\ 9 & 6 & 0 & 8 & 5 \\ 10 & 4 & 8 & 0 & 6 \\ 7 & 3 & 5 & 6 & 0 \end{bmatrix} \tag{2.2}$$

Una vez obtenida la solución inicial, se selecciona el entorno con el menor coste. Esta elección se convierte en la nueva solución y el movimiento que la originó se añade a la lista tabú. La Figura 2.2 ilustra cómo la solución y la lista tabú evolucionan a través de las iteraciones.

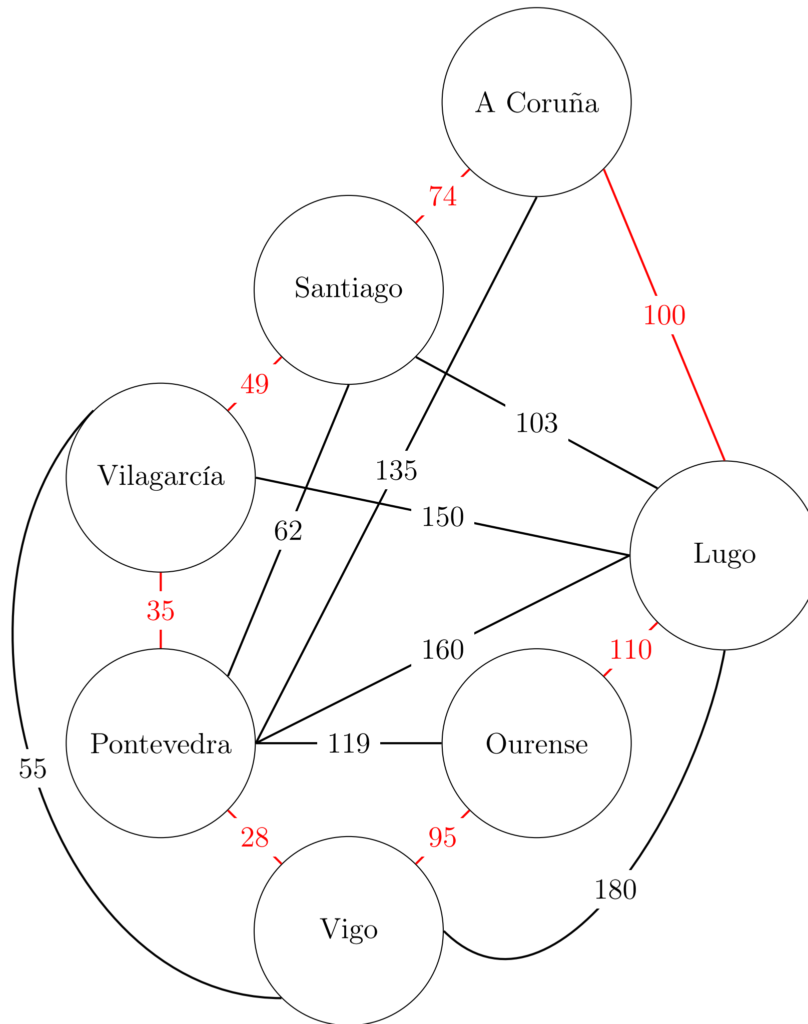


Figura 2.1: Grafo del ejemplo con la solución óptima marcada en rojo.

```
Iteración  1  Coste  491  Mejor coste  491
Lista tabú:
numeric(0)
Iteración  2  Coste  531  Mejor coste  491
Lista tabú:
6_5
  4
Iteración  3  Coste  491  Mejor coste  491
Lista tabú:
6_5 5_6
  3   4
Iteración  4  Coste  535  Mejor coste  491
Lista tabú:
6_5 5_6 5_4
  2   3   4
Iteración  5  Coste  491  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5
  1   2   3   4
Iteración  6  Coste  569  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5 7_6
  0   1   2   3   4
Iteración  7  Coste  491  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5 7_6 6_7
  0   0   1   2   3   4
Iteración  8  Coste  531  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5 7_6 6_7
  4   0   0   1   2   3
Iteración  9  Coste  491  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5 7_6 6_7
  3   4   0   0   1   2
Iteración  10  Coste  535  Mejor coste  491
Lista tabú:
6_5 5_6 5_4 4_5 7_6 6_7
  2   3   4   0   0   1

El coste de la búsqueda tabú es  491  y la solución es  1 7 6 5 4 3 2
```

Figura 2.2: Coste y lista tabú en las distintas iteraciones del algoritmo.

2.2. Templado simulado

2.2.1. Descripción

El templado simulado es una metaheurística de búsqueda local, presentada en Kirkpatrick et al. (1983). Esta proporciona un mecanismo para evitar óptimos locales. Como se señala en Anily y Federgruen (1987), los algoritmos deterministas de búsqueda presentan dos problemas. En primer lugar, la solución es fuertemente dependiente del punto de partida. El segundo es que tienden a quedarse atrapados en los óptimos locales. Estos problemas son los que el templado simulado trata de evitar, permitiendo, a veces, la posibilidad de escoger soluciones peores que las ya obtenidas.

El nombre de esta metaheurística viene del proceso en el que se basa, es decir, el proceso de *physical annealing* con sólidos. Consiste en calentar un sólido y después permitir que se enfríe lentamente. Con esto se consigue un entramado regular y sin defectos.

A grandes rasgos, el templado simulado consiste en, iterativamente, comparar la solución actual con la candidata. Para aceptar una u otra se usa una probabilidad dependiente de la distancia entre las soluciones y de un parámetro de temperatura, que disminuirá progresivamente con el tiempo.

Funcionamiento

El objetivo del templado simulado es encontrar la solución óptima dado un problema de optimización. En una gran cantidad de problemas habrá más de un óptimo. Para tratar de evitar óptimos locales la metaheurística acepta soluciones subóptimas.

El proceso se inicia seleccionando una solución inicial $\omega_0 \in \Omega$.

Un parámetro clave del proceso es la temperatura t_k, la cual se actualiza en cada iteración k. Inicialmente la temperatura es alta, lo que ayuda al algoritmo a salir de óptimos locales pudiendo seleccionar, con mayor probabilidad, soluciones que empeoren la solución actual, ω. Conforme avanza el proceso, la temperatura va disminuyendo y con ella la probabilidad de aceptar soluciones peores que la actual. El parámetro temperatura debe cumplir,

$$t_k > 0 \quad \forall k \in \mathbb{N} \quad y \quad \lim_{k \to \infty} t_k = 0.$$

Esto permite que al final del proceso el algoritmo se centre en mejorar la solución.

Fijada la temperatura, para cada iteración del bucle k, se generarán un número fijo de so-

luciones $\omega' \in N(\omega)$, ya sea de forma aleatoria o aplicando algún tipo de regla. Para cada nueva solución ω' se realiza una comparación con ω utilizando la **probabilidad de aceptación**.

$$P(\text{Aceptar } \omega' \text{ como la siguiente solución}) = \begin{cases} \exp\left(-\frac{f(\omega')-f(\omega)}{t_k}\right), & \text{si } f(\omega') > f(\omega) \\ 1, & \text{si } f(\omega') \leq f(\omega). \end{cases}$$

La solución actual se sustituye por la alternativa si esta es mejor, es decir, su función de coste es menor. En otro caso, se sustituye con una probabilidad que es más pequeña cuanto más baja es la temperatura o mejor es la solución actual con respecto de la alternativa.

La Figura 2.3 muestra una representación gráfica del proceso. La primera gráfica representa la función de coste, esta tiene múltiples mínimos. En la segunda gráfica se representa la evolución del parámetro de temperatura. El punto rojo y el verde representan dos soluciones en distintos momentos del proceso. En el caso del punto rojo estamos al principio del algoritmo. La solución está muy cerca de un mínimo local, pero como la temperatura en ese punto es todavía alta, aún puede escapar del mismo. En el punto verde ya nos encontramos en el final del proceso. La temperatura ya es baja, así que hay menos probabilidad de que se acepten soluciones peores. En ese momento, el algoritmo solo se centrará en mejorar la solución.

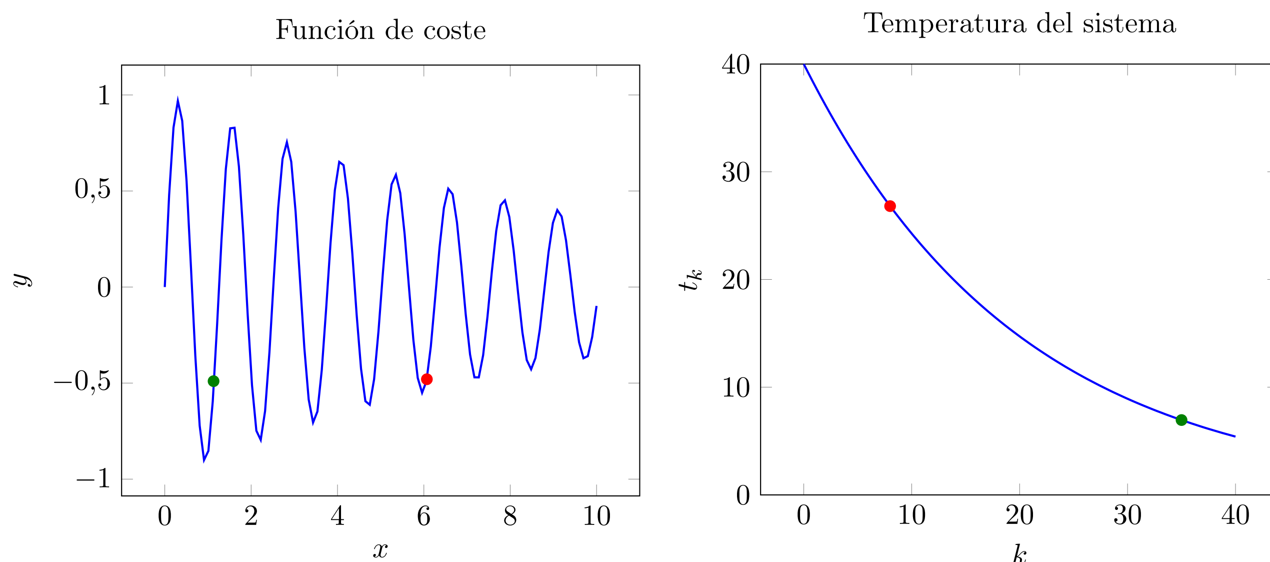


Figura 2.3: Ejemplo de templado simulado.

Sea $g_k(\omega, \omega')$ la probabilidad de generar una solución candidata ω' partiendo de la solución actual ω en cada iteración k, que cumple,

$$\sum_{\omega' \in N(\omega)} g_k(\omega, \omega') = 1, \quad \forall \omega \in \Omega, k = 1, 2, ..., m.$$

Entonces podemos definir una matriz estocástica no negativa $\mathbf{P}_k$ que representa la probabilidad de transición de ω a ω', siendo,

$$\mathbf{P}_k(\omega,\omega') = \begin{cases} g_k(\omega,\omega')\exp(\frac{-\Delta_{\omega,\omega'}}{t_k}), & \text{si } \omega' \in N(\omega), \omega' \neq \omega \\ 0, & \text{si } \omega' \notin N(\omega) \\ 1 - \sum_{\omega'' \in N(\omega), \omega'' \neq \omega} \mathbf{P}_k(\omega,\omega''), & \text{si } \omega' = \omega \end{cases}$$

para todo $\omega \in \Omega$ y $k = 1, 2, \ldots$ y siendo $\Delta_{\omega,\omega'} = f(\omega') - f(\omega)$. Esta transición de probabilidades define una secuencia de soluciones generada desde una cadena de Markov no homogénea.

Convergencia asintótica

Podemos modelar el algoritmo utilizando la teoría de la cadena de Markov.

Definición 2.4. (Çinlar, 1975) Dado un espacio de probabilidad (Ω, S, P) y un espacio numerable de estados E, se denomina *cadena de Markov*, al proceso estocástico $X = \{X_n : \Omega \to E; n \in \mathbb{N}\}$ que cumple

$$P\{X_{n+1} = j | X_0, \ldots, X_n\} = P\{X_{n+1} = j | X_n\}$$

para todo $j \in E$ y $n \in \mathbb{N}$.

Es decir, una cadena de Markov es una secuencia de variables aleatorias en la que cada eslabón de la cadena solo es condicionalmente dependiente del eslabón inmediatamente anterior.

En el caso del templado simulado, cada estado de la cadena de Markov representaría una posible solución del problema. Las transiciones entre estados vienen dadas por la probabilidad de pasar de una solución ω a otra ω', $\mathbf{P}_k(\omega,\omega')$, que como ya vimos antes depende de la temperatura y de la variación entre el coste de la soluciones.

En la literatura, la convergencia asintótica del algoritmo adquiere dos posibles enfoques, uno basado en la teoría de la cadena de Markov homogénea y otro basado en la teoría de la cadena de Markov no homogénea.

Definición 2.5. (Çinlar, 1975) Dado un espacio de probabilidad (Ω, S, P) y un espacio de estados E, llamamos cadena de Markov homogénea a aquella en la que su probabilidad condicional cumple,

$$P\{X_{n+1} = j | X_n = i\} = P(i,j), \quad i, j \in E,$$

es decir, es independiente de n, siendo $P(i,j)$ la probabilidad de ir del estado i al j.

Esto es, la probabilidad de ir de un estado a otro no cambia con el tiempo.

La visión de la cadena de Markov homogénea, concluye que cada temperatura t_k será constante por un número de iteraciones suficiente para que la matriz $\mathbf{P}_k$ pueda alcanzar la distribución

estacionaria π_k. Esto viene dado por el teorema 2.12. Pasaremos primero a discutir ciertos resultados previos que son clave para comprender el teorema.

Definición 2.6. (Çinlar, 1975) Para cualquier estado i en una cadena de Markov, si la probabilidad de que empezando en ese estado i, el proceso vuelva a pasar alguna vez por dicho estado, es uno, entonces, el estado i se dice *recurrente*. Se dice *nulo* si el tiempo esperado de retorno es infinito.

Definición 2.7. (Çinlar, 1975) Un estado j, es *alcanzable* desde, i, si existe un entero $n \geq 0$, tal que, $\mathbf{P}^n(i,j) > 0$. Es decir, que empezando desde el estado i, es posible (con probabilidad positiva) entrar en el estado j en un número finito de transiciones.

Definición 2.8. (Çinlar, 1975) Una cadena de Markov es *irreducible*, si y sólo si, todos los estados pueden alcanzarse entre sí.

Definición 2.9. (Çinlar, 1975) Una cadena de Markov es *aperiódica* si todos sus estados son aperiódicos. Un estado j se dice aperiódico si el máximo común divisor del conjunto de los $n \geq 1$ para los cuales $P^n(j,j) > 0$ es 1.

Lema 2.10. *(Çinlar, 1975) Dados dos estados k y j, con j recurrente, donde k es alcanzable desde j, entonces, j es alcanzable desde k y $F(k,j) = 1$.*

Lema 2.11. *(Çinlar, 1975) Sea j un estado recurrente no nulo aperiódico, entonces,*

$$\pi(j) = \lim_{n\to\infty} P^n(j,j) > 0,$$

y para cualquier otro estado i del conjunto de estados,

$$\lim_{n\to\infty} P^n(i,j) = F(i,j) \cdot \pi(j)$$

Teorema 2.12. *Sea $\boldsymbol{P}_k(\omega,\omega')$ la probabilidad de pasar de la solución ω a la solución ω' en una iteración interna del bucle k, y $\boldsymbol{P}_k^{(m)}(\omega,\omega')$ la probabilidad de ir desde la solución ω a la solución ω' en m bucles internos. Si la cadena de Markov asociada con $\boldsymbol{P}_k^{(m)}(\omega,\omega')$ es irreducible, aperiódica con un número finito de soluciones y se supone que todos los estados son recurrentes no nulos, entonces*

$$\lim_{m\to\infty} \boldsymbol{P}_k^{(m)}(\omega,\omega') = \pi_k(\omega') \tag{2.3}$$

existe para todo $\omega,\omega' \in \Omega$ e iteraciones k. Además, $\pi_k(\omega')$ es la única solución estrictamente positiva de

$$\pi_k(\omega') = \sum_{\omega\in\Omega} \pi_k(\omega)\boldsymbol{P}_k(\omega,\omega'), \quad \forall\omega' \in \Omega, \tag{2.4}$$

y

$$\sum_{\omega\in\Omega} \pi_k(\omega) = 1. \tag{2.5}$$

Demostración. (Çinlar, 1975). Todas las soluciones representan estados recurrentes no nulos. Por tanto, por el Lema 2.10, la probabilidad de que empezando en ω, el algoritmo llegue a visitar, ω' es 1, es decir, lo denotamos $F(\omega, \omega') = 1$ para todo $\omega, \omega' \in \Omega$, y entonces, por el Lema 2.11,

$$\lim_{m\to\infty} \mathbf{P}_k^{(m)}(\omega, \omega') = \gamma_k(\omega') \quad \omega, \omega' \in \Omega, \tag{2.6}$$

existe. Además,

$$\gamma_k(\omega') > 0, \quad \sum_{\omega'\in\Omega} \gamma_k(\omega') = 1. \tag{2.7}$$

Para cualquier subconjunto finito A en Ω,

$$\mathbf{P}_k^{(n+1)}(\omega, \omega') = \sum_{\widetilde{\omega}\in\Omega} \mathbf{P}_k^{(n)}(\omega, \widetilde{\omega})\mathbf{P}_k(\widetilde{\omega}, \omega') \geq \sum_{\widetilde{\omega}\in A} \mathbf{P}_k^{(n)}(\omega, \widetilde{\omega})P(\widetilde{\omega}, \omega'),$$

y en el límite, como $n \to \infty$, esto implica por (2.6) que

$$\gamma_k(\omega') \geq \sum_{\widetilde{\omega}\in A} \gamma_k(\widetilde{\omega})\mathbf{P}_k(\widetilde{\omega}, \omega'), \quad \omega' \in \Omega.$$

Como esto se cumple para cualquier subconjunto finito A, entonces, tomando una sucesión de esos subconjuntos A creciente hasta Ω, tenemos que

$$\gamma_k(\omega') \geq \sum_{\widetilde{\omega}\in\Omega} \gamma_k(\widetilde{\omega})\mathbf{P}_k(\widetilde{\omega}, \omega'), \quad \omega' \in \Omega. \tag{2.8}$$

Si la desigualdad de (2.8) fuera estricta para algún ω', entonces, sumando ambos lados sobre ω' y sabiendo que $\sum_{\omega'} \mathbf{P}_k(\widetilde{\omega}, \omega') = 1$, tendríamos que $\sum_{\omega'} \gamma_k(\omega') > \sum_{\widetilde{\omega}} \gamma_k(\widetilde{\omega})$. Siendo esto absurdo, tenemos

$$\gamma_k(\omega') = \sum_{\widetilde{\omega}\in\Omega} \gamma_k(\widetilde{\omega})\mathbf{P}_k(\widetilde{\omega}, \omega'), \quad \omega' \in \Omega. \tag{2.9}$$

Ahora (2.9) y (2.7) nos muestran que γ_k es la solución de (2.4) y de (2.5). Si π fuese otra solución, entonces, en notación matricial, $\pi_k = \pi_k\mathbf{P}_k$, el cual por iteración nos da $\pi_k = \pi_k\mathbf{P}_k = \pi_k\mathbf{P}_k^{(2)} = \cdots = \pi_k\mathbf{P}_k^{(m)}$; es decir,

$$\pi_k(\omega') = \sum_{\widetilde{\omega}\in\Omega} \pi_k(\widetilde{\omega})P_k^{(m)}(\widetilde{\omega}, \omega'). \tag{2.10}$$

Ahora tomado límites en (2.10), por el teorema de la convergencia dominada, obtenemos,

$$\pi_k(\omega') = \sum_{\widetilde{\omega}\in\Omega} \pi_k(\widetilde{\omega}) \lim_{n\to\infty} \mathbf{P}_k^{(n)}(\widetilde{\omega}, \omega') = \sum_{\widetilde{\omega}} \pi_k(\widetilde{\omega})\gamma_k(\omega') = \gamma_k(\omega');$$

es decir, solo hay una solución π_k y (2.3) se cumple.

□

Este enfoque conduce a que el algoritmo convergerá a la solución óptima en ciertas condiciones si la temperatura se reduce a una velocidad lo suficientemente lenta y el sistema se equilibra en cada iteración, k.

Definición 2.13. (Çinlar, 1975) Dado un espacio de probabilidad (Ω, S, P) y un espacio de estados E, llamamos cadena de Markov no homogénea a aquella que cumple,

$$P_n\{X_{n+1} = j | X_n = i\} = P_n(i, j), \quad i, j \in E.$$

La probabilidad de ir de i a j puede depender del tiempo.

El enfoque de la cadena de Markov no homogénea en el templado simulado varía del anterior en que no se necesita alcanzar una distribución estacionaria en cada bloque externo k. A medida que va cambiando la temperatura va cambiando la matriz de transición. Esto es más consistente con la descripción del algoritmo que se proporcionó al inicio de la sección. Este enfoque puede parecer más complejo, pero asegura que de igual manera se llegará a una solución óptima si la temperatura t_k desciende lo suficientemente rápido.

El resultado y su demostración se puede ver en detalle en Mitra et al. (1986). Este enfoque es mucho más complejo que el enfoque homogéneo, no obstante, es más robusto pues da condiciones explícitas para encontrar la tasa de convergencia correcta.

Algoritmo 2 Templado simulado. (Gendreau & Potvin, 2010, capítulo 1)

1: Seleccionar una solución inicial $\omega \in \Omega$
2: Establecer la temperatura inicial $T = T_0$
3: Seleccionar un programa de enfriamiento de temperatura t_k
4: Seleccionar el contador de cambio de temperatura inicial $k = 0$
5: Seleccionar un programa de iteración M_k que define el número de iteraciones ejecutadas a cada temperatura t_k
6: **repeat**
7:     Establecer el contador de repetición $m = 0$
8:     **repeat**
9:         Generar una solución $\omega' \in N(\omega)$
10:         Calcular $\Delta_{\omega,\omega'} = f(\omega') - f(\omega)$
11:         **if** $\Delta_{\omega,\omega'} < 0$ **then**
12:             $\omega \leftarrow \omega'$
13:         **else**
14:             $\omega \leftarrow \omega'$ con probabilidad $\exp\left(-\frac{\Delta_{\omega,\omega'}}{t_k}\right)$
15:         **end if**
16:         $m \leftarrow m + 1$
17:     **until** $m = M_k$
18:     $k \leftarrow k + 1$
19: **until** se cumple el criterio de parada

2.2.2. Programación en R

Siguiendo el Algoritmo 2, se ha implementado el algoritmo de templado simulado. Para ello se ha usado la misma implementación de la función objetivo, la estructura de entornos y la generación de la solución inicial que en la búsqueda tabú.

Los parámetros del algoritmo como son la temperatura inicial, la temperatura final, el número de iteraciones internas y la tasa de enfriamiento los selecciona el usuario. El ciclo principal del algoritmo termina cuando la temperatura llegó a la temperatura de parada o llevan ciertos cambios de temperatura sin variación del coste. Hay que tener especial cuidado a la hora de elegir los parámetros pues una convergencia demasiado rápida puede llevar a unas soluciones deficientes.
Dentro del ciclo principal, hay un ciclo interno. En él para cada temperatura se van seleccionando entornos aleatorios. En estos entornos se acepta la solución si es mejor que la anterior o si está dentro de la probabilidad de aceptación. Esta probabilidad de aceptación se ha implementado escogiendo un número aleatorio con el que comparamos la probabilidad de aceptación.

```
for (iter_interior in seq_len(maxIteraciones)) {
  indices <- sample(seq_along(solucion), 2)
  i <- indices[1]
  j <- indices[2]

  solucionTmp <- solucion
  solucionTmp[c(i, j)] <- solucionTmp[c(j, i)]

  costeTmp <- core$funcionObjetivo(solucionTmp)
  delta <- costeTmp - coste
  probAceptacion <- exp(-delta / temperatura)

  if (delta < 0 || runif(1) < probAceptacion) {
    solucion <- solucionTmp
    coste <- costeTmp
  }
}
```

2.2.3. Ejemplo ilustrativo

En esta sección usaremos el ejemplo de la sección de búsqueda tabú para ilustrar el algoritmo templado simulado. Recordamos que el objetivo es encontrar la ruta más corta para visitar las siete ciudades gallegas y regresar a la ciudad de origen, visitando cada ciudad una única vez. La matriz de distancias que representa el problema se puede ver en la matriz (2.1). Además, el recorrido óptimo se marca en rojo en el grafo 2.1.

Inicialmente, el algoritmo genera una solución inicial con una estrategia voraz, en este caso es la óptima con coste 491. Tras generar la solución inicial y fijar los parámetros iniciales, estos son,

- Temperatura inicial: 40.
- Tasa de enfriamiento: 0,1.
- Temperatura de parada: 0.
- Número de cambios de temperatura (iteraciones del bucle externo) sin cambios en el coste para parar: 10.
- Número de iteraciones del bucle interno: 100.

A continuación, el algoritmo empieza el bucle externo. Este bucle irá disminuyendo la temperatura en función a la tasa de enfriamiento hasta que se cumpla alguno de los criterios de parada. La representación del enfriamiento, para el ejemplo, se puede ver en la Figura 2.4.
Dentro del bucle externo, hay un bucle interno que durante 100 iteraciones genera entornos aleatorios y va comparando el coste de ese entorno aleatorio con el coste actual. En la Figura 2.5 podemos ver la evolución de los costes aceptados.
Los entornos generados tienen, o bien, un coste mejor, o bien, peor, si la probabilidad de aceptación supera un número aleatorio. En la Figura 2.6 podemos ver cómo evoluciona la probabilidad de aceptación a lo largo de la ejecución. Un punto interesante es como a medida que la temperatura disminuye la probabilidad de aceptación también lo hace. De esta manera, será más probable que el algoritmo converja.

En este ejemplo, vemos que entra en juego el criterio de parada de iteraciones sin cambios. Si solo usásemos el otro criterio de parada, es decir, alcanzar la temperatura mínima, entonces, como el coste se habría estabilizado, las iteraciones restantes solo añadirían complejidad al programa.

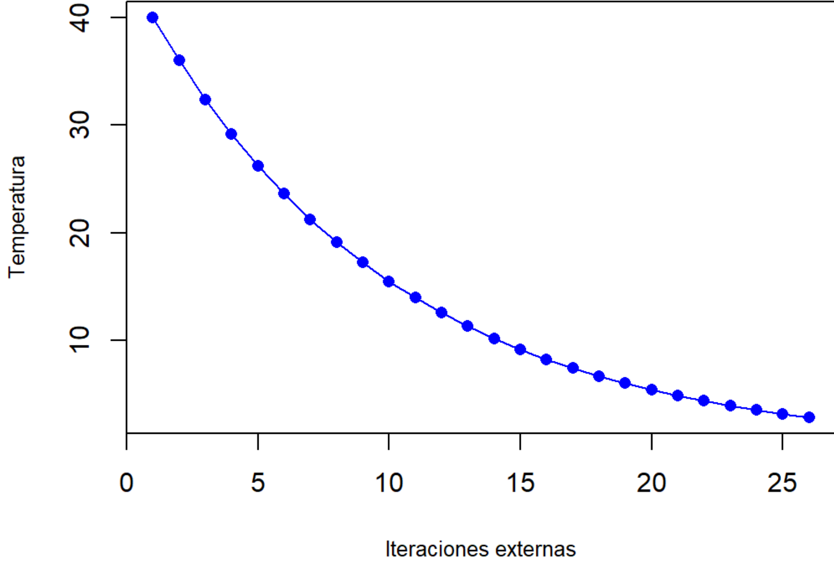


Figura 2.4: Enfriamiento.

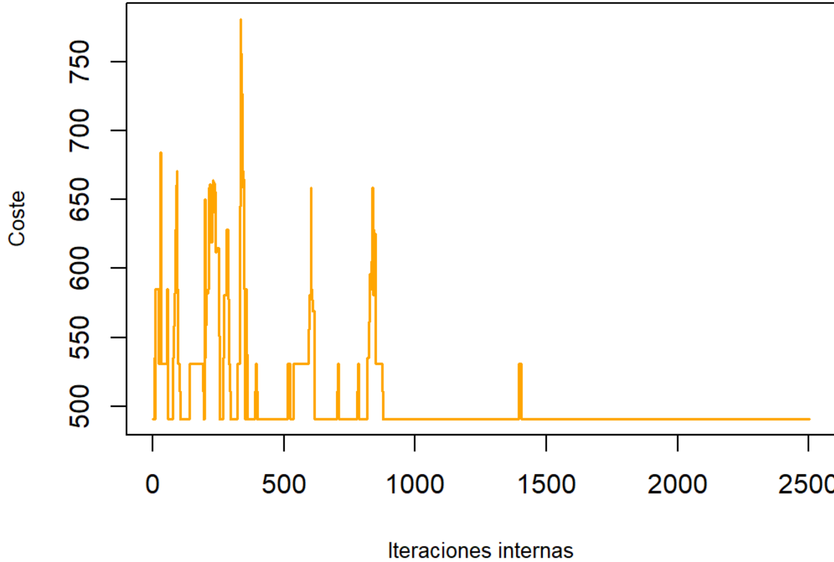


Figura 2.5: Variación del coste.

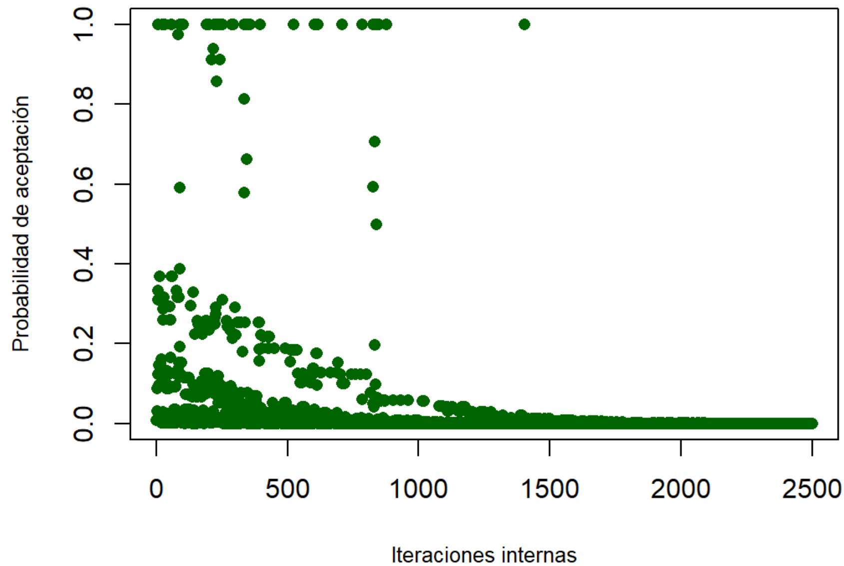


Figura 2.6: Variación de la probabilidad de aceptación.

2.3. Algoritmos genéticos

2.3.1. Descripción

Motivación

La primera vez que aparece el término algoritmo genético, o GA por sus siglas en inglés, es en Holland (1975). Este libro, junto con Goldberg (1989), fueron fundamentales.

Los algoritmos genéticos imitan el proceso de selección natural. De forma general, la descendencia tendrá características determinadas por la combinación de las características de sus padres.

Conceptos básicos

En los algoritmos genéticos, se puede separar la representación del problema, *genotipo*, de las variables con las que está formulado, *fenotipo*. Tenemos que el fenotipo es el vector de variables de decisión, ω, y el genotipo es la representación de ω mediante una cadena de caracteres, s, de longitud l, formada por símbolos del alfabeto, $\mathcal{A}$.

Existen muchas representaciones posibles. Sin embargo, muchos autores optan por la representación binaria ($\mathcal{A} = \{0, 1\}$), ya que la consideran óptima en un contexto matemático.

En general, llamaremos *cromosomas* a las cadenas de caracteres, *gen* a cada uno de los caracteres de un cromosoma y *alelo* a cada posible valor que puede tomar un gen.

Llamaremos $\mathcal{S} \subseteq \mathcal{A}^l$ al espacio de búsqueda definido como el conjunto de todos los cromosomas del alfabeto $\mathcal{A}$ de longitud l que son válidos.

Puesto que el genotipo es solo una representación de la solución, para obtener el fenotipo usamos el mapeo genotipo-fenotipo.

Definición 2.14. Una función de mapeo $c : \mathcal{S} \subseteq \mathcal{A}^l \rightarrow \Omega$, a la que denominaremos *mapeo genotipo-fenotipo*, asigna a cada genotipo, $s \in \mathcal{S}$, un fenotipo $\omega = c(s)$.

Es preferible que esta función sea biyectiva, para que, a cada cadena, s, le corresponda un solo vector, ω, y a cada vector, ω, le corresponda solo una cadena de caracteres, s.

Habiendo introducido este concepto, podemos reformular el problema de optimización inicial. Para ello, introduciremos la función (2.11).

$$\begin{aligned} g : \mathcal{S} \subseteq \mathcal{A}^l &\rightarrow \Omega \rightarrow \mathbb{R} \\ s &\longrightarrow c(s) \longrightarrow g(s) = f(c(s)). \end{aligned} \tag{2.11}$$

Con ella el nuevo problema de optimización consistirá en encontrar (2.12)

$$\arg \min_{s \in S} g. \tag{2.12}$$

Para evaluar cada solución introducimos una nueva función monótona, h, a la que denominaremos *aptitud*, que se usa para eliminar la existencia de valores negativos:

$$\begin{aligned} h : \ \mathbb{R} &\rightarrow \mathbb{R}^+ \\ g(s) &\rightarrow h(g(s)). \end{aligned} \tag{2.13}$$

Población inicial

Otro ingrediente de los algoritmos genéticos es la población inicial. Escoger la adecuada es clave para el correcto funcionamiento del proceso. Uno de los puntos más importantes a analizar es el tamaño de esta. Esto se debe a que una población demasiado pequeña puede hacer que el espacio de búsqueda no se recorra adecuadamente y una demasiado grande puede afectar

enormemente a la eficiencia. En todo caso, en Reeves (1993), se propone que todo punto del espacio de búsqueda debe poder alcanzarse desde la población inicial únicamente por cruce.

Otro punto importante es la forma en la que se eligen los individuos de la población inicial. En este ámbito, es mucho más efectivo que no se haga de forma aleatoria.

Criterio de parada

Un algoritmo genético puede ejecutarse infinitamente si carece de criterio de parada. Algunos ejemplos de criterios de parada podrían ser limitar el tiempo de ejecución o parar cuando la población no sea lo suficientemente diversa.

Selección

El término selección se refiere a la forma en la que se seleccionan los padres para generar la descendencia. Esta selección se basa en la aptitud (2.13).

El método de selección por ruleta, *Roulette-Wheel Selection* o RWS es un método de selección donde la probabilidad de selección de un cromosoma determinado es proporcional a su aptitud. Este funciona como una ruleta de la fortuna, donde los sectores son los individuos de la población con un tamaño proporcional a la probabilidad de selección. Este método permite escoger con más probabilidad a los sectores más grandes, es decir, los mejores progenitores. Además, da la posibilidad a otros individuos de generar descendencia permitiendo así diversidad en la población. Por ejemplo, dado cinco cromosomas con aptitudes $\{32, 9, 17, 17, 25\}$, podríamos representarlo como se ve en la Figura 2.7. En esta representación cada sector representa la probabilidad de que se escoja cada individuo. En total la probabilidad del círculo es uno. En el borde de cada sector se representa la probabilidad acumulada hasta ese momento. La flecha representa una ejecución del método donde se ha escogido el individuo número 1.

El método SUS (*stochastic universal selection*), propuesto en Baker (1987), funciona igual que el RWS con la única excepción de que en vez de que en cada "tirada" se escoja un individuo, se seleccionan varios a la vez. Para el ejemplo anterior, el resultado podría representarse como en la Figura 2.8. La figura es la misma con un único cambio, la existencia de cinco flechas igualmente espaciadas para la selección simultánea de cinco individuos.

El método de selección por *ranking* consiste en ordenar los individuos de la población por su aptitud. Una vez se ordenan la aptitud pasa a un segundo plano y la selección se realiza de acuerdo con el *ranking*. Este método no está directamente ligado a la aptitud, esto permite evitar

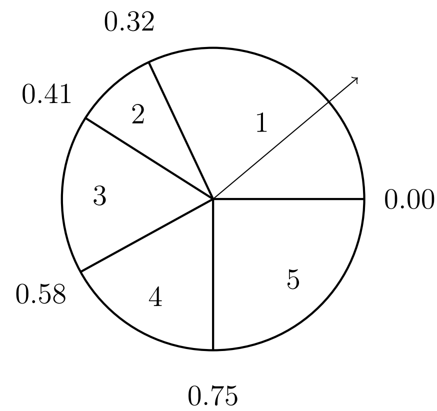


Figura 2.7: Ejemplo RWS.

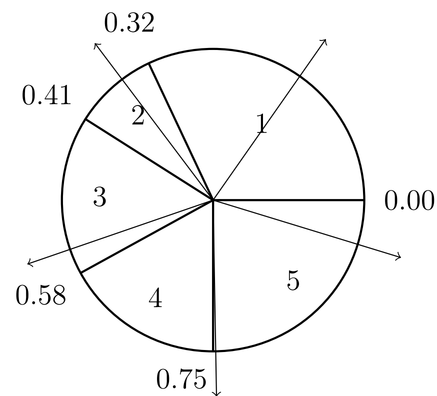


Figura 2.8: Ejemplo SUS.

que unos valores con aptitud muy alta estropeen la diversidad de la población.

El método de selección por torneo consiste en seleccionar un número de cromosomas al azar y escoger el mejor según su aptitud. La ventaja de este método es que solo hace falta ordenar un pequeño grupo de individuos y no toda la población, esto se traduce en una ganancia en tiempo de computación.

Cruce

Definición 2.15. Denominamos *cruce* a la forma en la que se combinan los genes de las soluciones padres para generar nuevas soluciones descendientes.

Existen distintos tipos de cruce.

Definición 2.16. Un *cruce de m puntos* es un cruce en el que se seleccionan m puntos de cruce, y se intercambian las partes de los cromosomas entre esos puntos.

Padre 1	1	0	1	0	0	1	0	→	**Hijo 1**	1	0	1	1	0	0	1
Padre 2	0	1	1	1	0	0	1		**Hijo 2**	0	1	1	0	0	1	0

Cuadro 2.1: Cruce de un punto.

Padre 1	1	0	1	0	0	1	0	→	**Hijo 1**	1	0	1	1	0	1	0
Padre 2	0	1	1	1	0	0	1		**Hijo 2**	0	1	1	0	0	0	1

Cuadro 2.2: Cruce de dos puntos.

Definición 2.17. Un *cruce uniforme* es aquel que, en vez de usar puntos de cruce fijos, decide que genes cruzar a partir de una máscara binaria (si es 0 el gen se coge del primer padre, si es 1 del segundo). La máscara se conforma de modo aleatorio siguiendo una distribución de Bernoulli.

Padre 1	1	0	1	0	0	1	→	**Hijo 1**	0	0	1	0	0	0
Padre 2	0	1	1	1	0	0								

Cuadro 2.3: Cruce uniforme dada la máscara: 1 0 1 0 0 1.

Hay en algunas situaciones en el que los cruces definidos con anterioridad pueden llevar a problemas, como puede ser el caso del TSP. Para solucionarlos se han introducido nuevos enfoques.

Definición 2.18. El *cruce parcialmente mapeado (PMX)* es un cruce en que inicialmente se eligen dos puntos de cruce de forma aleatoria. En el segmento comprendido entre los dos puntos de cruce se aplica el mapeo de intercambio y de forma que todos los cambios que surgiesen en esa zona se aplicarían a toda la cadena.

Padre 1	1	6	3	4	5	2	→	$3 \leftrightarrow 1$	→	**Hijo 1**	3	5	1	2	6	4
Padre 2	4	3	1	2	6	5		$4 \leftrightarrow 2$		**Hijo 2**	2	1	3	4	5	6
								$5 \leftrightarrow 6$								

Cuadro 2.4: Ejemplo cruce parcialmente mapeado con puntos de cruce $X = 2$ e $Y = 5$.

Definición 2.19. El *cruce no lineal con máscara* es el cruce que utiliza una máscara para realizarse. Los unos en la máscara indican que los números en esas posiciones se mantienen de un padre. En las posiciones con ceros se colocarán los alelos faltantes en el orden que aparezcan en el otro padre.

Padre 1	1	6	3	4	5	2
Padre 2	4	3	1	2	6	5

→

Hijo 1	1	_	3	_	_	2
Hijo 2	4	_	1	_	_	5

→

Hijo 1	1	4	3	6	5	2
Hijo 2	4	6	1	3	2	5

Cuadro 2.5: Cruce no lineal con máscara: 1 0 1 0 0 1.

Mutación

Definición 2.20. La *mutación* es un concepto que designa a la variación en un gen o genes elegidos de manera aleatoria. Para representar la mutación se usa una máscara, donde los unos representan los genes mutados y los ceros los que no.

Antes de la mutación	1	0	1	1	0	0	1
Después de la mutación	1	0	0	1	1	0	1

Cuadro 2.6: Mutación con máscara: 0 0 1 0 1 0 0.

El objetivo principal de la mutación es conseguir variabilidad en la población para evitar así, una convergencia prematura y recorrer más en profundidad el espacio de búsqueda.

Para generar la máscara, una opción es usar una distribución de Poisson con parámetro λ. El parámetro representa el número promedio de mutaciones por cromosoma.

Para escoger el alelo en el que se mutará el gen puede hacerse de forma aleatoria, mediante una distribución de probabilidad o usando otras técnicas como la de escoger valores cercanos.

Estrategias

Los anteriores conceptos se usan partiendo de ciertas estrategias. Estas son las siguientes:

Definición 2.21. Denominamos cruce-Y-mutación a la estrategia en la que primero se realiza el cruce y después se realiza la mutación.

Definición 2.22. Denominamos cruce-O-mutación a la estrategia en la que dependiendo del momento del proceso se aplica un cruce o una mutación.

Nueva población

El planteamiento inicial de Holland consistía en aplicar todo el proceso a una población inicial hasta que esta hubiese generado suficientes cromosomas para crear la nueva generación.

Una forma más natural que conlleva mejores resultados es el uso de *elitismo* y *solapamientos poblacionales*.

El elitismo consiste en permitir únicamente la supervivencia de los mejores individuos.

El solapamiento poblacional consiste en sustituir únicamente una parte de la población, g. La nueva generación estará formada por $m - g$ individuos de la anterior generación y g nuevos individuos, siendo m el tamaño de la población. Para seleccionar los g individuos que se eliminan se suele escoger los peores individuos, aunque se pueden aplicar otras estrategias.

Conservación de la diversidad

Esta es una pieza clave para el óptimo funcionamiento de los algoritmos genéticos. Debido a que uno de los efectos de la selección es evitar la diversidad, debemos evitar que esta se realice demasiado rápido, para lo cual usamos diversos métodos. Una técnica popular es el uso de una política de no duplicados, es decir, la descendencia tendrá la restricción de que no podrán ser clones de los individuos existentes. En la práctica, esta política es muy costosa. Esto se debe a la necesidad de comparar cada individuo con el nuevo. Para solucionar este problema hay técnicas que permiten reducir la posibilidad de crear clones fijándose en los padres.

2.3.2. Programación en R

En esta sección, analizaremos la implementación de un algoritmo genético aplicado al TSP, siguiendo la estructura explicada en el Algoritmo 3.
Inicialmente, se escoge una población inicial. Para ello, se han generado tantas listas aleatorias de soluciones como tamaño tiene la población.

```
poblacion <- list()
for (i in seq_len(tamPoblacion)) {
    solucion <- sample(seq_len(core$numCiudades))
    poblacion[[i]] <- solucion
}
```

En segundo lugar, se ha implementado la selección de los padres en la población. Para ello, se

Algoritmo 3 Algoritmo Genético. (Gendreau & Potvin, 2010)[capítulo 5]

1: Escoger una población inicial de cromosomas;
2: **while** la condición de terminación no esté satisfecha **do**
3:     **repeat**
4:         **if** la condición de cruce está satisfecha **then**
5:             seleccionar cromosomas padres;
6:             elegir parámetros de cruce;
7:             realizar cruce;
8:         **end if**
9:         **if** la condición de mutación está satisfecha **then**
10:             elegir puntos de mutación;
11:             realizar mutación;
12:         **end if**
13:         evaluar la aptitud de la descendencia;
14:     **until** se hayan creado suficientes descendientes;
15:     seleccionar nueva población;
16: **end while**

ha usado el método de selección por torneo con 3 individuos y donde se escogen los 2 que tengan mejor valor de la función de aptitud.

```
tamTorneo <- 3
torneo <- sample(poblacion, tamTorneo, replace = FALSE)
indices <- order(sapply(torneo, function(sol) calcularAptitud(sol, core)),
↪  decreasing = TRUE)[1:2]
padres <- torneo[indices]
```

Para el cálculo de la función de aptitud, se ha escogido un método sencillo de trasformación lineal.

```
distanciaTotal <- core$funcionObjetivo(solucion)
return(1 / distanciaTotal)
```

Posteriormente, se implementaron el cruce y la mutación. En esta implementación, ambos se aplican siempre. Para el cruce se ha usado un *Partially Mapped Crossover*. El TSP requería que se usase un cruce no uniforme, pues, en otro caso, se podrían generar soluciones no válidas como, por ejemplo, la existencia de una misma ciudad presente dos veces en la misma solución. Para implementar este cruce se escogen dos puntos aleatorios de la solución. Después, se define la

función de mapeo mediante un diccionario. Por último, se hacen los intercambios comprobando por orden los elementos del mapeo para evitar problemas.

```
puntos <- sort(sample(seq_len(core$numCiudades), 2))
punto1 <- puntos[1]
punto2 <- puntos[2]

hijo1 <- padre1
hijo2 <- padre2

mapeo <- setNames(padre2[punto1:punto2], padre1[punto1:punto2])

for (i in names(mapeo)) {
    i <- as.numeric(i)

    if (i %in% hijo1) {
        indice1 <- which(hijo1 == i)
        valor_map <- mapeo[[as.character(i)]]

        if (valor_map %in% hijo1) {
            indice2 <- which(hijo1 == valor_map)
            hijo1[indice2] <- i
        }
        hijo1[indice1] <- valor_map
    }

    if (i %in% hijo2) {
        indice1 <- which(hijo2 == i)
        valor_map <- mapeo[[as.character(i)]]

        if (valor_map %in% hijo2) {
            indice2 <- which(hijo2 == valor_map)
            hijo2[indice2] <- i
        }
        hijo2[indice1] <- valor_map
    }
}
```

La mutación en esta implementación se realiza intercambiando dos ciudades aleatorias. Esto ocurre cuando un número aleatorio es menor que 0.5.

```
if (runif(1) < 0.5) {
    indices <- sample(seq_along(gen), 2)
    tmp <- gen[indices[1]]
    gen[indices[1]] <- gen[indices[2]]
    gen[indices[2]] <- tmp
}
```

Por último, solamente con esta implementación no se obtenían muy buenos resultados. Por lo que decidió implementarse elitismo. Para ello en cada iteración se seleccionan los m mejores individuos de la población y se copian directamente a la nueva población. Posteriormente, se general los $n - m$ individuos restantes, obteniendo así la nueva población.

2.3.3. Ejemplo ilustrativo

En esta sección, vamos a ilustrar el algoritmo genético con el mismo ejemplo usado en las secciones anteriores.
En primer lugar, generamos una población inicial que se ha decidido tenga 30 individuos, es decir, soluciones. Su generación se hará de forma aleatoria sin importar que haya individuos iguales. A continuación, tomamos nuestra población inicial y la ordenamos en orden de aptitud descendiente. En la Tabla 2.7 se muestra la población inicial generada en una ejecución del algoritmo.

Seleccionamos el 5 % de los mejores elementos de la población y los copiamos directamente a la nueva población. Para rellenar el resto de la nueva población realizamos una selección por torneo que nos dará dos padres pertenecientes a la población inicial. Con estos dos padres mediante un proceso de cruce PMX obtendremos dos descendientes que podrán sufrir alguna mutación para introducir diversidad en la población. A continuación, tenemos el primer cruce y mutación que se realizan.

- **Padre 1:** [5, 4, 6, 1, 7, 3, 2]. **Coste:** 657.
- **Padre 2:** [7, 6, 5, 3, 4, 1, 2]. **Coste:** 664.

Hijos generados:

- **Hijo 1 antes de la mutación:** [7, 5, 4, 1, 6, 3, 2]. **Coste:** 742.

Solución	Coste	Solución	Coste
[7, 4, 3, 6, 2, 5, 1]	853	[1, 2, 5, 7, 6, 4, 3]	695
[5, 6, 4, 1, 3, 7, 2]	795	[5, 4, 6, 1, 7, 3, 2]	657
[5, 7, 4, 3, 2, 6, 1]	765	[7, 1, 5, 6, 3, 4, 2]	778
[2, 5, 4, 3, 7, 1, 6]	737	[3, 6, 2, 5, 1, 4, 7]	947
[2, 5, 1, 3, 6, 7, 4]	937	[7, 4, 1, 3, 6, 5, 2]	874
[1, 2, 5, 3, 7, 6, 4]	746	[2, 3, 5, 6, 7, 1, 4]	725
[7, 6, 4, 1, 5, 3, 2]	734	[3, 1, 6, 7, 5, 2, 4]	845
[7, 4, 1, 6, 2, 3, 5]	812	[4, 6, 3, 1, 2, 7, 5]	678
[4, 2, 5, 1, 6, 7, 3]	849	[7, 6, 2, 4, 1, 5, 3]	896
[2, 6, 3, 4, 1, 5, 7]	859	[1, 4, 7, 5, 2, 3, 6]	854
[7, 6, 5, 3, 4, 1, 2]	664	[5, 2, 4, 1, 7, 6, 3]	891
[1, 6, 4, 2, 3, 5, 7]	725	[1, 3, 5, 6, 2, 7, 4]	832
[2, 1, 3, 6, 7, 5, 4]	744	[2, 7, 5, 1, 3, 4, 6]	779
[1, 5, 7, 2, 4, 6, 3]	865	[5, 6, 4, 7, 2, 1, 3]	679
[6, 1, 7, 3, 5, 2, 4]	818	[2, 6, 7, 3, 1, 5, 4]	821

Cuadro 2.7: Población inicial.

- **Hijo 2 antes de la mutación:** [6, 4, 7, 3, 5, 1, 2]. **Coste:** 752.

Resultados después de la mutación:

- **Hijo 1 después de la mutación:** [7, 5, 4, 1, 6, 3, 2]. **Coste:** 742.
- **Hijo 2 después de la mutación:** [6, 4, 7, 1, 5, 3, 2]. **Coste:** 731.

En la Figura 2.9, podemos ver que durante toda la ejecución la población es lo suficientemente diversa.

Al ser un ejemplo pequeño podemos ver en la Figura 2.10 que rápidamente llegaremos al coste óptimo y al aplicar elitismo, este se mantendrá en la población hasta el final.

Aquí entra en juego el criterio de parada de detener la ejecución tras varias generaciones sin cambios en el coste óptimo, en este caso, 10 generaciones. Si la ejecución no terminase el proceso seguiría hasta crear 100 generaciones. Al terminar, de cualquiera de las dos maneras, se tomará como óptimo el individuo de la última población con mayor aptitud, en este caso, es óptima pues su coste es 491.

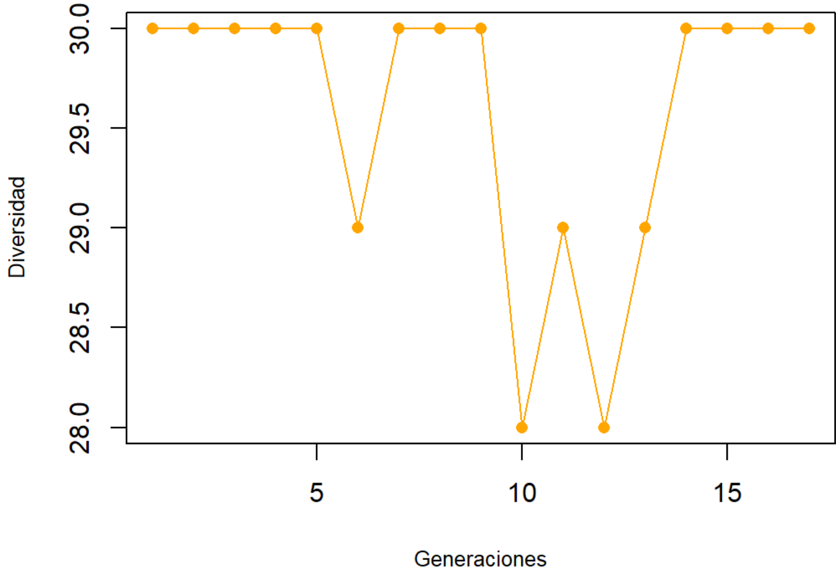


Figura 2.9: Diversidad de las generaciones.

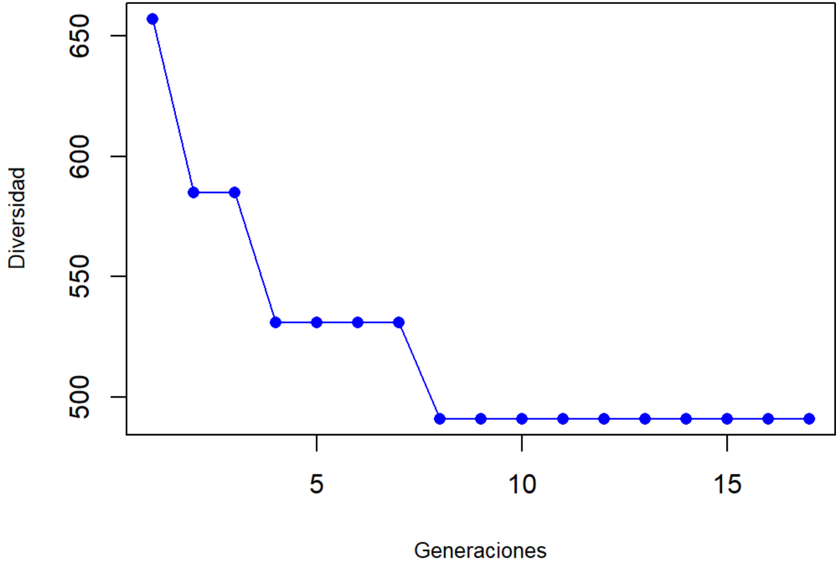


Figura 2.10: Mejor coste de cada generación.

2.4. Optimización de la colonia de hormigas

2.4.1. Descripción

Historia y motivación

Esta metaheurística se basa en los comportamientos que tienen las hormigas durante su búsqueda de comida. En esta, las hormigas depositan feromonas que guían la búsqueda del resto de la colonia. Se ha observado que muchas especies de hormigas utilizan este sistema para encontrar el camino más corto hacia su comida. Por ejemplo, bajo la existencia de dos caminos, uno corto y uno largo, las hormigas elegirán, inicialmente, de forma aleatoria. A medida que pase el tiempo, las hormigas que escogieron el camino corto tardarán menos en volver. Esto hará que el camino corto se llene antes de feromonas llevando a más hormigas a escogerlo.

El primer método propuesto, el cual se basa directamente en el sistema anterior, fue el sistema de hormigas o AS. Este enfoque fue presentado en Dorigo (1992), aplicándolo al TSP.

Conceptos básicos

La optimización de la colonia de hormigas es una metaheurística basada en el comportamiento de las hormigas. En ella, se usan hormigas artificiales cuyo objetivo es encontrar la solución al problema, en este caso, la ruta más corta. Además, estas hormigas artificiales se transmiten información entre ellas mediante rastros de feromonas artificiales. Esta información les facilitará la búsqueda.

Formalmente, las hormigas artificiales son procedimientos estocásticos de construcción de soluciones. Por ser un procedimiento estocástico cada hormiga podrá crear variedad de soluciones, lo que permitirá al algoritmo explorar un gran número de estas. La construcción de estas soluciones se basará en dos aspectos. El primero, será la información heurística de la que se disponga, es decir, que opciones serían mejores dependiendo de las características del problema. El segundo, serán los rastros de feromonas artificiales que se van actualizando durante todo el proceso y guiarán la búsqueda.

Para construir las soluciones, cada hormiga parte de una solución parcial vacía. En cada iteración, las hormigas extienden su solución parcial actual con un componente de solución factible. Este componente se elige de forma probabilística entre los disponibles. Si no se puede encontrar tal componente, esta solución, o bien se desecha, o bien se completa su construcción, dando lugar a una solución infactible.

Para adaptar el problema de optimización a esta metaheurística necesitaremos asociar a cada

componente, ω_i^j, de cada solución, ω_i, una variable. A esta variable se le denomina variable de feromonas ya que representará el rastro de feromonas y se denotará por τ_{ij}. El valor de esta variable indica la capacidad de cada componente de ser añadido a la solución parcial actual de cada hormiga. Inicialmente, todos los valores de las variables de feromonas se fijarán en un valor inicial.

El valor de las feromonas depende del tiempo. Por tanto, debe existir un mecanismo para ir actualizándolos a lo largo de todo el proceso. Para esto se usan estos dos mecanismos:

- Depósito de feromonas. Dado un conjunto de soluciones de buena calidad, en este mecanismo se depositan más feromonas en la variable de feromonas que corresponda a los componentes de las soluciones que forman parte de dicho conjunto.
- Evaporación del rastro de feromonas: Como pasa en las situaciones reales, en esta estrategia las feromonas se van evaporando con el tiempo. De esta manera, se permite explorar nuevas áreas del espacio de búsqueda.

Generalmente, la actualización de los valores de las variables de feromonas a los que llamaremos τ_{ij} se puede ver en (2.14). En esta ecuación, S_{upd} representa las soluciones que se van a usar para depositar feromonas. En el mecanismo de depósito de feromonas, sería el conjunto de soluciones de buena calidad. Además, $\rho \in [0, 1]$ se llama tasa de evaporación. Por último, $g(s)$ es denominada función de evaluación y cumple que dadas dos soluciones ω y ω', tales que, $f(\omega) < f(\omega')$, entonces, $g(\omega) \geq g(\omega')$.

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{s \in S_{\text{upd}} : \omega_i^j \in s} g(s). \tag{2.14}$$

Extensiones

Durante los últimos años, se han propuesto numerosas mejoras para el sistema de hormigas con el objetivo de que compitiese con las metaheurísticas más importantes.

Entre las mejoras que se propusieron, está el ampliar la optimización mediante la búsqueda local. Esta optimización parece proporcionar muy buenos resultados. Consiste en aplicar una búsqueda local una vez que las hormigas seleccionaron las soluciones candidatas para obtener la solución final.

Otra mejora introducida es el uso de una estrategia elitista. Esta estrategia consiste en añadir a las soluciones que se usan para depositar feromonas la mejor solución para cada iteración.

Ampliando la estrategia anterior se creó el sistema de hormigas basado en ranking. En este sistema, las hormigas se ordenan según la longitud de la solución que generaron en esa iteración.

Para depositar feromonas se escogen las mejores y la mejor solución global. Cada solución deposita feromonas de forma proporcional a su posición en el ranking, siendo la mejor solución la que deposita un mayor número de ellas.

El sistema de hormigas máximo-mínimo o MMAS es una modificación del sistema de hormigas. La principal diferencia es la existencia de un mínimo y de un máximo para las variables de feromonas. Además, las variables de feromonas se inicializan con el valor máximo. Por último, la actualización de feromonas solo la puede realizar la mejor solución, o bien la generada, o bien la global.

La mejora del sistema de la colonia de hormigas se diferencia del sistema de hormigas en tres sentidos como se presenta en Dorigo y Gambardella (1997): el uso de una fuerte estrategia elitista; la actualización continua de feromonas, es decir, cuando se añade cada componente, no solo cuando se crea una solución; y, por último, el uso de una regla pseudoaleatoria proporcional. Esta regla la usan las hormigas para elegir el próximo componente de su solución. Siendo q_0 un parámetro entre 0 y 1 y q un número aleatorio entre 0 y 1, entonces, la regla dice que, con una probabilidad q menor que q_0 se elegirá el componente donde el producto entre el valor de la variable de feromonas y la información heurística sea máximo. En otro caso, se sigue el mecanismo del sistema de hormigas.

Algoritmo 4 Optimización de la colonia de hormigas. (Gendreau & Potvin, 2010)[capítulo 8]

```
1: Inicialización
2: while se cumple el criterio de parada do
3:     while para cada hormiga do
4:         Construir las soluciones de las hormigas
5:         Opcionalmente, aplicar búsqueda local
6:     end while
7:     Actualizar las feromonas
8: end while
```

2.4.2. Programación en R

Como ya hemos estudiado, el algoritmo de optimización de la colonia de hormigas consigue encontrar una buena aproximación del camino óptimo basándose en el comportamiento que tienen las hormigas en su búsqueda de comida. En esta sección, trataremos de explicar su implementación en R para su aplicación al TSP. Cabe recordar, que la estructura de las soluciones y el cálculo de la función objetivo se mantiene igual que en las metaheurísticas anteriores.
La implementación sigue el Algoritmo 4. En primer lugar, en cada iteración, se construyen las soluciones parciales de cada hormiga. La construcción de dichas soluciones se hace de forma pro-

babilística, empezando en una ciudad aleatoria y seleccionando las siguientes ciudades en función de la distancia a la ciudad actual d_{ij}, las variables de feromonas τ_{ij} y unos parámetros alfa α y beta β predefinidos, que representan la importancia de la matriz de feromonas y la importancia de la heurística, respectivamente. Se sigue la Fórmula (2.15) para fijar la probabilidad de las ciudades no visitadas.

$$p(j) = \frac{\tau_{ij}^{\alpha} * d_{ij}^{-\beta}}{\sum \tau_{ij}^{\alpha} * d_{ij}^{-\beta}}, \tag{2.15}$$

siendo i la ciudad actual y siendo j cualquier ciudad no visitada. A continuación, podemos ver cómo se implementaría.

```
probabilidades <- ifelse(
    visitadas,
    0,
    (matrizFeromonas[ciudadActual, ]^alfa) * ((1 /
    ↪  matrizDistancias[ciudadActual, ])^beta)
)

probabilidades <- probabilidades / sum(probabilidades)

return(sample(seq_len(core$numCiudades), 1, prob = probabilidades))
```

Posteriormente, para cada hormiga se escoge el mejor coste a modo de búsqueda local. Esto mejorará la búsqueda y acelerará la convergencia.

```
if (coste < mejorCoste) {
    mejorCoste <- coste
    mejorSolucion <- solucion
}
```

Por último, al final de cada iteración se actualiza la matriz de feromonas siguiendo la fórmula (2.14) de la siguiente manera.

```
matrizFeromonas <- matrizFeromonas * (1 - tasaEvaporacion)

for (i in seq_along(soluciones)) {
    solucion <- soluciones[[i]]
    coste <- costes[i]
    for (j in seq_len(length(solucion) - 1)) {
```

```
            ciudad1 <- solucion[j]
            ciudad2 <- solucion[j + 1]
            matrizFeromonas[ciudad1, ciudad2] <- matrizFeromonas[ciudad1, ciudad2]
            ↪  + (1 / coste)
        }
    }
```

El algoritmo terminará cuando se hayan completado las iteraciones dadas por el usuario.

2.4.3. Ejemplo ilustrativo

En esta sección, ilustraremos la implementación anteriormente presentada. Para ello, utilizaremos el mismo ejemplo que se ha estado usando con el resto de metaheurísticas, representado en la Figura 2.1 y cuya matriz de distancias es (2.1). En primer lugar, fijamos los parámetros iniciales como:

- Alfa: 1.
- Beta: 2.
- Tasa de evaporación: 0,5.
- Número de hormigas: 10.
- Número de iteraciones: 100.

A continuación, empezamos a generar las soluciones de las hormigas. Podemos ver cómo evoluciona el coste medio de las soluciones parciales de las hormigas en la Figura 2.11. Podemos ver que este va descendiendo, pues va convergiendo debido al depósito de feromonas.

Por último, vemos que la matriz de feromonas se va actualizando. En la Figura 2.12 podemos ver la matriz de feromonas en la última iteración representada como un mapa de calor, es decir, cuanto más alto el valor de la matriz de feromonas en un punto ij más rojo se verá el cuadrado correspondiente. En ella, se observa que las posiciones con un valor mayor, las representadas en rojo, se corresponden con la solución. Los representados en azul tiene valores muy bajos, como se puede ver en la diagonal que el valor es cero.

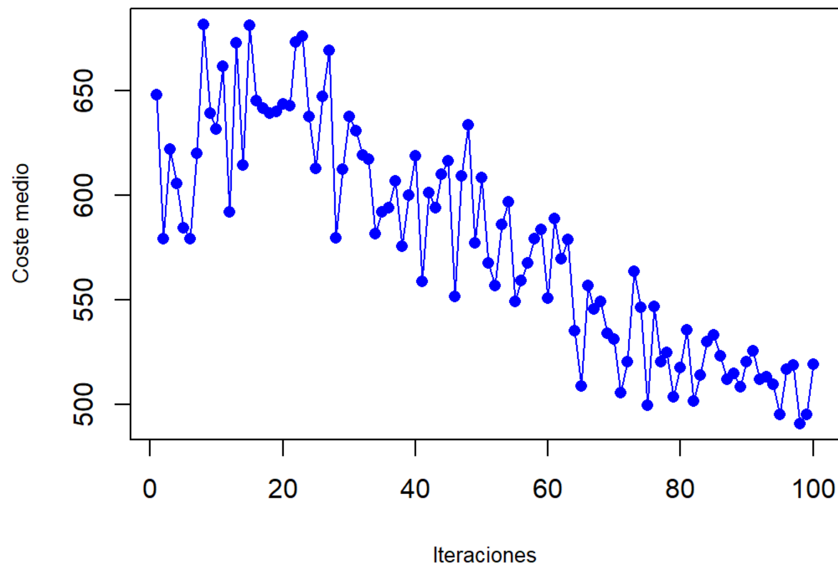


Figura 2.11: Coste medio de las soluciones de las hormigas.

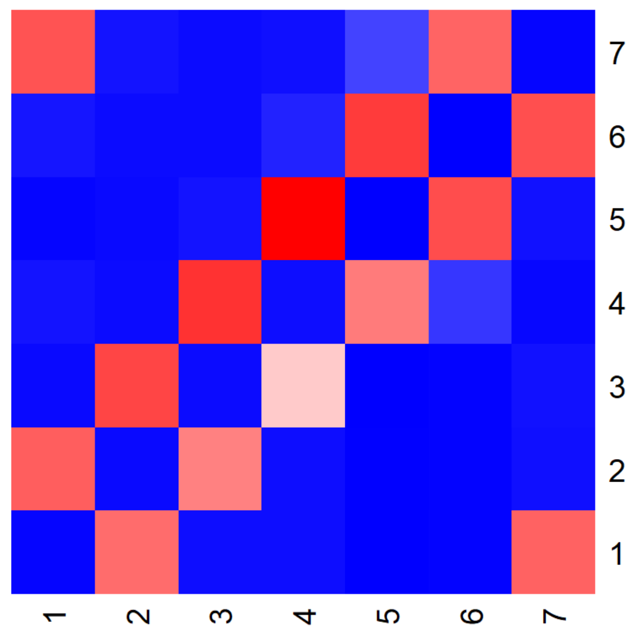


Figura 2.12: Mapa de calor de la matriz de feromonas en la última iteración.

Capítulo 3

Estudio computacional

Este capítulo tiene por objetivo llevar a cabo un análisis computacional detallado de las metaheurísticas implementadas en secciones previas.

3.1. Formato de las pruebas

Para realizarlo se utilizarán algunas instancias seleccionadas de la librería TSPLIB. Esta librería contiene variedad de instancias del TSP tanto simétricas como asimétricas y de diferentes niveles de complejidad. Los detalles de esta librería se pueden encontrar en Reinelt (s.f.).

Para el estudio, se escogerán instancias de diversos tamaños. Consideraremos entonces tres grupos:

- Pequeñas: Menos de 50 ciudades.
- Medianas: Hasta 300 ciudades.
- Grandes: Más de 300 ciudades.

Asimismo, se seleccionarán instancias del TSP solamente simétricas porque son para las que se hizo la implementación. Además, de distintos tipos de distancias. La Tabla 3.1 muestra las instancias seleccionadas clasificadas por tamaño, simetría y tipo de distancia utilizada.

Tamaño	**Tipo**	**Instancia**	**Coste óptimo**
Pequeño, 21	Simétrico	*gr21*	2707
Pequeño, 42	Simétrico	*swiss42*	1273
Mediano, 58	Simétrico	*brazil58*	25395
Mediano, 101	Simétrico, distancia euclídea en $\mathbb{R}^2$	*eil101*	629
Mediano, 493	Simétrico, distancia euclídea en $\mathbb{R}^2$	*d493*	35002
Grande, 783	Simétrico, distancia euclídea en $\mathbb{R}^2$	*rat783*	8806
Grande, 1748	Simétrico, distancia euclídea en $\mathbb{R}^2$	*vm1748*	336556
Grande, 1889	Simétrico, distancia euclídea en $\mathbb{R}^2$	*rl1889*	316536

Cuadro 3.1: Selección de instancias del TSP.

Las pruebas se llevaron a cabo utilizando el supercomputador Finisterrae III, proporcionado por el CESGA (Centro de Supercomputación de Galicia). Para cada metaheurística se han realizado 10 ejecuciones, intentando mantener la regularidad en los parámetros para que las comparaciones fuesen lo más justas posibles. Estos parámetros son:

- **Búsqueda tabú:**
  - maxIteraciones: 1000.
  - permanenciaListaTabu: $\sqrt{n}$.
  - maxIterSinMejora: 100.
- **Templado simulado:**
  - temperaturaInicial: 80.
  - tasaEnfriamiento: $1 - \frac{1}{n}$.
  - temperaturaParada: 1.
  - maxIteracionesSinCambios: 100.
  - maxIteraciones: 1000.
- **Algoritmo genético:**
  - tamPoblacion: 50.
  - numGeneraciones: 1000.
  - porcentajeElitismo: 0,05.
  - maxIterSinCambios: 100.
- **Optimización de la colonia de hormigas:**

- numHormigas: $\sqrt{n}$.
- maxIteraciones: 1000.
- tasaEvaporacion: 0,1.
- alfa: 1.
- beta: 2.

3.2. Resultados de las pruebas

Las pruebas se han realizado siguiendo las directrices presentes en la sección anterior. Los resultados obtenidos se presentan en las tablas incluidas en el Anexo II. Estos resultados se evaluaron en función de dos métricas clave: el tiempo de ejecución y la desviación respecto al coste óptimo.

3.2.1. Análisis de la complejidad algorítmica

La complejidad algorítmica de las metaheurísticas programadas se puede expresar en función del número de ciudades (n) y los parámetros de la esta. En este análisis, se evalúa si los tiempos de ejecución medidos en las pruebas experimentales son consistentes con las complejidades teóricas.

Búsqueda tabú

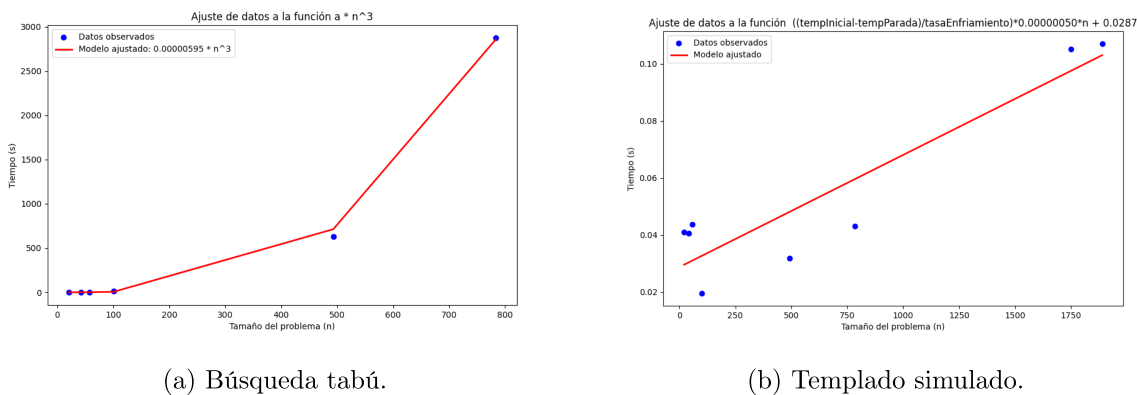
Para el algoritmo de búsqueda tabú desarrollado, la complejidad teórica en el peor de los casos es $\mathcal{O}(maxIteraciones \times n^2)$. Este resultado se obtiene a través de un análisis de bucles. El bucle principal se repite, en el peor caso, *maxIteraciones* veces. Dos elementos dentro de este bucle contribuyen a la complejidad. El primero es la función de generación de vecindarios, que produce todas las combinaciones posibles de pares (i, j) con $i < j$. Esta operación tiene una complejidad de $\mathcal{O}(n^2)$, donde n representa el número de ciudades. Para cada combinación, se genera una nueva solución intercambiando i y j, también con una complejidad de $\mathcal{O}(n^2)$. Además, las operaciones de cálculo de costes para todas las soluciones generadas y la verificación y filtrado de movimientos tabú implican una complejidad de $\mathcal{O}(n^2)$. Finalmente, reducir el tiempo en la lista tabú tiene una complejidad de $\mathcal{O}(t)$, donde t es el tamaño de dicha lista. De ahí que la complejidad sea $\mathcal{O}(\text{maxIteraciones} \times (n^2 + n^2 + n^2 + t))$, lo cual simplifica al resultado dado.

Con los parámetros indicados previamente, logramos en la práctica obtener el ajuste presentado en la Figura 3.1a. Este ajuste demuestra un rendimiento significativamente superior al análisis en el peor de los casos, posiblemente porque la mayoría de las instancias concluye mucho antes de alcanzar el número máximo de iteraciones.

Templado simulado

Para analizar la complejidad en el peor de los casos del algoritmo de templado simulado, es crucial evaluar sus bucles. En el peor escenario, el bucle principal puede continuar hasta que la temperatura llegue a *temperaturaParada*. A su vez, el bucle interno se ejecutará *maxIteraciones* veces en el peor de los casos. Dentro de dicho bucle, se lleva a cabo el cálculo de la función objetivo, cuya complejidad es $\mathcal{O}(n)$. Además, se realizan otras operaciones de manera secuencial que no superan esta complejidad. Por consiguiente, la complejidad en el peor de los casos será de $\mathcal{O}(\frac{tempInicial - tempParada}{tasaEnfriamiento} \times maxIteraciones \times n)$.

Usando los parámetros especificados, obtenemos en la práctica el ajuste mostrado en la Figura 3.1b. Este ajuste no es tan evidente como los demás. Esto se debe a que, por la naturaleza de las metaheurísticas, su comportamiento no siempre es consistente para cada instancia. En algunos casos, el algoritmo puede converger más rápidamente, mientras que en otros no.



(a) Búsqueda tabú. (b) Templado simulado.

Figura 3.1: Complejidad temporal.

Algoritmo genético

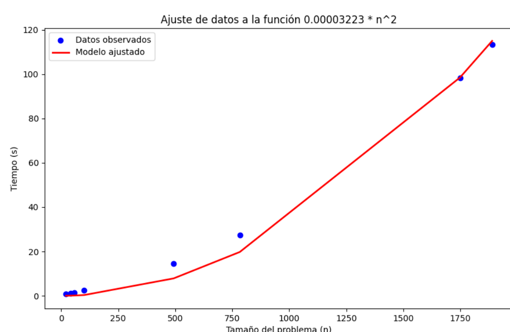
La complejidad en el peor caso del algoritmo genético es $\mathcal{O}(tamPoblacion * numGeneraciones * n)$. Esto viene de que el bucle exterior hace como máximo *numGeneraciones* iteraciones. Dentro del mismo, hay otro bucle que se hace como máximo *tamPoblacion* veces. La función con complejidad más alta es $\mathcal{O}(n)$.

Analizando los resultados de las pruebas obtenemos el ajuste de la Figura 3.2a.

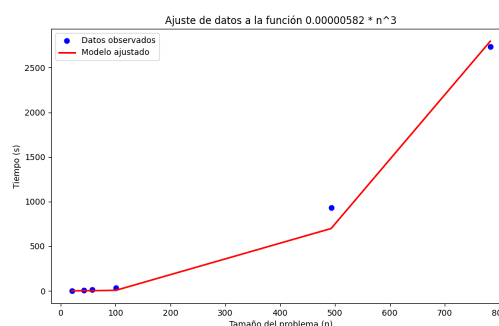
Optimización de la colonia de hormigas

Para analizar la complejidad en el peor caso de la optimización de la colonia de hormigas, nos fijaremos como en anteriores aproximaciones en sus bucles. El bucle principal itera como máximo *maxIteraciones* veces. Asimismo, dentro de cada iteración, las operaciones principales son la generación de soluciones de las hormigas y la actualización de la matriz de feromonas. La primera, tiene una complejidad $\mathcal{O}(n^2)$ y se tiene que hacer tantas veces como el número de hormigas que haya. La segunda, tiene una complejidad $\mathcal{O}(n^2)$. Por tanto, el algoritmo tiene una complejidad $\mathcal{O}(maxIteraciones \times numHormigas \times n^2)$.

Analizando los resultados de las pruebas obtenemos el ajuste de la Figura 3.2b.



(a) Algoritmo genético.



(b) Optimización de la colonia de hormigas.

Figura 3.2: Complejidad temporal.

Comparativa

Tras analizar la complejidad temporal de cada metaheurística, se ha determinado que depende en gran medida de los parámetros seleccionados. Por eso es importante encontrar un equilibrio entre tiempo y distancia. La metaheurística con menor complejidad es el templado simulado, seguida por el algoritmo genético. Sin embargo, las más temporalmente complejas son la búsqueda tabú y la optimización de la colonia de hormigas.

3.2.2. Desviación respecto al coste óptimo

El objetivo de esta sección es comprobar qué metaheurística dista menos del coste óptimo. Por ello, tomando las medias de los costes de las ejecuciones para cada instancia, veremos cual es

el error relativo con respecto a los costes óptimos de la Tabla 3.1, utilizando la siguiente fórmula:

$$\frac{|\text{coste óptimo} - \text{coste medio}|}{\text{coste medio}},$$

donde "coste óptimo" representa el coste óptimo para una instancia y "coste medio" representa el coste medio obtenido de realizar sucesivas ejecuciones con una misma metaheurística usando dicha instancia.

Estos valores se pueden consultar en la Tabla 3.2, donde se puede observar que los algoritmos que mejor se aproximan al óptimo son la búsqueda tabú y la optimización de la colonia de hormigas. Siendo el primero, el que se aproxima de forma más constante. Sin embargo, en complejidad temporal estos son los peores. Otro punto que destacar es el pésimo desempeño del algoritmo genético.

Instancia	**Búsqueda tabú**	**Templado simulado**	**Algoritmo genético**	**ACO**
gr21	19.50 %	14.66 %	2274.17 %	0.85 %
swiss42	11.39 %	62.66 %	64.14 %	2.47 %
brazil58	14.90 %	16.70 %	85.11 %	2.64 %
eil101	13.04 %	394.66 %	22772.03 %	10.27 %
d493	15.06 %	21.14 %	887.81 %	19.01 %
rat783	22.25 %	56.44 %	148590.90 %	27.01 %
vm1748	NaN	21.27 %	99.27 %	NaN
rl1889	NaN	22.98 %	3992.41 %	NaN

Cuadro 3.2: Errores relativos.

Capítulo 4

Conclusiones

Este trabajo de fin de grado tuvo como objetivo principal ofrecer una guía didáctica de cuatro de las metaheurísticas aplicables al TSP, estas son, búsqueda tabú, templado simulado, algoritmos genéticos y optimización de la colonia de hormigas. Por ello, durante él se han abordado tanto su explicación teórica como su implementación práctica, con el fin de proporcionar una herramienta que facilite la enseñanza de estas técnicas.

Durante el estudio, se observó que no existe una metaheurística que sea mejor que las demás. Por ejemplo, aunque la búsqueda tabú sea la que más se acerca al óptimo de forma constante, esta no es la más eficiente en términos temporales. Por otra parte, aunque la optimización de la colonia de hormigas tenga muy buenos resultados en términos de desviación del óptimo es muy costosa y aumenta con el tamaño del problema. Por ello, para decidir que metaheurística utilizar deberíamos analizar el problema concreto. Además, las metaheurísticas se comportarán de una manera distinta dependiendo de los parámetros con los que se trabaje. Por otra parte, se vio que las metaheurísticas son útiles en los casos en los que se necesite una solución aproximada en un tiempo relativamente reducido, pero no cuando se necesite una solución exacta.

Este trabajo aporta un análisis comparativo de las metaheurísticas para el TSP, proporcionando una visión de sus características. Asimismo, se desarrolló un paquete de R que implementa los algoritmos para poder reproducir los experimentos y generar ampliaciones en futuros trabajos. Este paquete será público para poder completarlo con nuevos algoritmos o funcionalidades. También, se ofrece una descripción clara y sencilla de las distintas metaheurísticas que pueden ser usadas como complemento en contextos educativos para facilitar la comprensión de estos algoritmos.

Entre las limitaciones del trabajo se encontraron que las implementaciones no tuvieron en cuenta todos los aspectos del TSP, como aquellos con distancias variables.

Para futuros estudios, se recomienda incorporar un análisis comparativo entre los mejores parámetros para cada metaheurística. Además, se podría investigar su aplicabilidad en problemas del TSP especialmente complejos, así como para instancias asimétricas. También, se sugiere realizar más pruebas para tener una visión más clara de la capacidad de estas. En el ámbito de la programación, se podrían hacer implementaciones de otras metaheurísticas y paralelizar las ya existentes para mejorar su eficiencia. Asimismo, se podría modificar la implementación de la búsqueda tabú para que no haga todos los cambios, si no que solo calcule la diferencia de coste que supondría y escoja la mejor. Por último, se podrían comparar los resultados obtenidos con *solvers* del estado del arte como es Gurobi.

En conclusión, este trabajo de fin de grado ha aportado una valiosa guía didáctica sobre las metaheurísticas aplicadas al TSP, con una implementación práctica de interés para cualquier aficionado a los problemas de optimización complejos.

Anexo I

Código de R completo

En este anexo, se presenta el código de R completo usado para las pruebas, el cual se puede usar accediendo a este enlace.

I.1. core.R

```
crear_core_tsp <- function(matrizDistancias) {
  # Validar la matriz
  numCiudades <- nrow(matrizDistancias)
  if (!all(diag(matrizDistancias) == 0)) {
    stop("La diagonal de la matriz de distancias debe contener ceros.")
  }
  if (any(matrizDistancias[upper.tri(matrizDistancias)] <= 0)) {
    stop("La matriz no puede tener ceros (salvo en la diagonal) o valores
    ↪  negativos.")
  }

  # Función para calcular el coste de una solución
  funcionObjetivo <- function(solucion) {
    # Crear un vector de índices que representen los pares consecutivos de
    ↪  ciudades
    indices <- cbind(solucion, c(solucion[-1], solucion[1]))

    # Sumar las distancias
    return(sum(matrizDistancias[indices]))
```

```
  }

  # Generación de una solución inicial, usando un algoritmo voraz
  generarSolucionInicial <- function() {
    solucion <- c(1)
    ciudadActual <- 1
    visitadas <- logical(numCiudades)
    visitadas[ciudadActual] <- TRUE

    for( i in 2:numCiudades ){
      distancias <- matrizDistancias[ciudadActual, ]
      distancias[visitadas] <- Inf
      ciudadActual <- which.min(distancias)

      solucion[i] <- ciudadActual
      visitadas[ciudadActual] <- TRUE
    }

    return(solucion)
  }

  # Devolver el objeto core con las funciones
  list(
    numCiudades = numCiudades,
    funcionObjetivo = funcionObjetivo,
    generarSolucionInicial = generarSolucionInicial
  )
}
```

I.2. busquedaTabu.R

```
busqueda_tabu <- function(matrizDistancias, maxIteraciones,
↪  permanenciaListaTabu, maxIterSinMejora) {

  core <- crear_core_tsp(matrizDistancias = matrizDistancias)

  # Tiempo de inicio
```

```
inicio <- Sys.time()

# Inicialización
listaTabu <- list()
solucion <- core$generarSolucionInicial()
coste <- core$funcionObjetivo(solucion)
mejorSolucion <- solucion
mejorCoste <- coste
iteracionesSinMejora <- 0

# Iteraciones de búsqueda tabú
for (iter in seq_len(maxIteraciones)) {
  # Generar vecindario
  vecindario <- generar_mejor_vecindario(solucion, mejorCoste,
  ↪  permanenciaListaTabu, core, listaTabu)
  coste <- vecindario$mejorCosteLocal
  solucion <- vecindario$mejorSolucionLocal
  listaTabu <- vecindario$listaTabu

  # Actualizar mejor solución global
  if (coste < mejorCoste) {
    iteracionesSinMejora <- 0
    mejorCoste <- coste
    mejorSolucion <- solucion
  } else {
    iteracionesSinMejora <- iteracionesSinMejora + 1
    if (iteracionesSinMejora == maxIterSinMejora) break
  }

  # Reducir tiempo de permanencia en lista tabú
  if (length(listaTabu) > 0) {
    listaTabu <- lapply(listaTabu, function(x) max(x - 1, 0))
  }
}

# Tiempo final
fin <- Sys.time()
```

```
  # Devolver resultados
  list(
    mejorSolucion = mejorSolucion,
    mejorCoste = mejorCoste,
    tiempoCPU = difftime(fin, inicio, units = "secs"),
    listaTabu = listaTabu
  )
}

# Función interna para generar el mejor vecindario
generar_mejor_vecindario <- function(solucion, mejorCosteGlobal,
↪  permanenciaListaTabu, core, listaTabu) {
    # Generar todas las combinaciones posibles de pares (i, j) donde i < j
    indices <- combn(seq_along(solucion), 2)
    i <- indices[1, ]
    j <- indices[2, ]

    # Crear todas las soluciones posibles intercambiando i y j
    solucionesAux <- t(apply(indices, 2, function(idx) {
      solucionAux <- solucion
      solucionAux[idx] <- solucionAux[rev(idx)]
      solucionAux
    }))

    # Calcular costes de todas las soluciones
    costos <- apply(solucionesAux, 1, core$funcionObjetivo)

    # Comprobar si el movimiento es tabú
    movimientos <- paste(solucion[i], solucion[j], sep = "_")
    movimientosEnTabu <- movimientos %in% names(listaTabu)
    movimientosTabuActivos <- sapply(movimientos, function(mov) {
      if (mov %in% names(listaTabu) && is.numeric(listaTabu[[mov]])) {
        listaTabu[[mov]] > 0
      } else {
        FALSE
      }
    })
```

```
    tabu <- movimientosEnTabu & movimientosTabuActivos

    # Filtrar movimientos no tabú o aquellos que mejoran el coste global
    validos <- !tabu | (tabu & costos < mejorCosteGlobal)

    # Seleccionar el mejor movimiento
    if (any(validos)) {
      mejorIdx <- which.min(costos[validos])
      mejorCosteLocal <- costos[validos][mejorIdx]
      mejorSolucionLocal <- solucionesAux[validos, , drop = FALSE][mejorIdx, ]
      mejorMovimiento <- movimientos[validos][mejorIdx]

      # Actualizar lista tabú
      listaTabu[[mejorMovimiento]] <- permanenciaListaTabu
    } else {
      mejorCosteLocal <- Inf
      mejorSolucionLocal <- solucion
    }

    list(
      mejorCosteLocal = mejorCosteLocal,
      mejorSolucionLocal = mejorSolucionLocal,
      listaTabu = listaTabu
    )
}
```

I.3. templadoSimulado.R

```
templado_simulado <- function(matrizDistancias, temperaturaInicial,
↪   tasaEnfriamiento, temperaturaParada, maxIteracionesSinCambios,
↪   maxIteraciones) {

  core <- crear_core_tsp(matrizDistancias)

  # Inicialización
```

```
inicio <- Sys.time()
solucion <- core$generarSolucionInicial()
mejorSolucion <- solucion
mejorCoste <- core$funcionObjetivo(solucion)

iteracionesSinCambios <- 0
temperatura <- temperaturaInicial

# Prealocar vectores
listaTemperaturas <- c(temperaturaInicial)
listaCostes <- c(mejorCoste)
listaProbAceptacion <- c()

coste <- mejorCoste
costeAnterior <- mejorCoste
iter <- 1

# Algoritmo de templado simulado
while (temperatura > temperaturaParada) {
  for (iter_interior in seq_len(maxIteraciones)) {
    # Elegir dos ciudades aleatoriamente
    indices <- sample(seq_along(solucion), 2)
    i <- indices[1]
    j <- indices[2]

    # Crear una solución vecina intercambiando las ciudades i y j
    solucionTmp <- solucion
    solucionTmp[c(i, j)] <- solucionTmp[c(j, i)]

    # Calcular el coste de la nueva solución
    costeTmp <- core$funcionObjetivo(solucionTmp)
    delta <- costeTmp - coste
    probAceptacion <- min(1,exp(-delta / temperatura))

    # Decidir si se acepta la nueva solución
    if (delta < 0 || runif(1) < probAceptacion) {
      solucion <- solucionTmp
      coste <- costeTmp
```

```
      }

      # Guardar métricas
      listaProbAceptacion <- c(listaProbAceptacion, probAceptacion)
      listaCostes <- c(listaCostes, coste)
      iter <- iter + 1
    }

    # Enfriar la temperatura
    temperatura <- temperatura * (1 - tasaEnfriamiento)
    listaTemperaturas <- c(listaTemperaturas, temperatura)

    # Control de iteraciones sin cambios
    if (coste == costeAnterior) {
      iteracionesSinCambios <- iteracionesSinCambios + 1
      if (iteracionesSinCambios >= maxIteracionesSinCambios) break
    } else {
      iteracionesSinCambios <- 0
      costeAnterior <- coste
    }
  }

  # Fin del tiempo de ejecución
  fin <- Sys.time()

  # Devolver resultados
  list(
    mejorSolucion = solucion,
    mejorCoste = coste,
    tiempoCPU = difftime(fin, inicio, units = "secs"),
    listaTemperaturas = listaTemperaturas,
    listaCostes = listaCostes,
    listaProbAceptacion = listaProbAceptacion
  )
}
```

I.4. algoritmoGenetico.R

```
algoritmo_genetico <- function(matrizDistancias, tamPoblacion, numGeneraciones,
↪  porcentajeElistismo, maxIterSinCambios) {

  core <- crear_core_tsp(matrizDistancias)

  # Tiempo de inicio
  inicio <- Sys.time()

  poblacion <- generarPoblacionInicial(tamPoblacion, core)
  elitismo <- ceiling(tamPoblacion * porcentajeElistismo)
  listaCostes <- c()
  listaDiversidad <- c()
  iteracionesSinCambios <- 0
  costeAnterior <- core$funcionObjetivo(poblacion[[which.max(sapply(poblacion,
  ↪  function(sol) calcularAptitud(sol, core)))]])

  for (gen in seq_len(numGeneraciones)) {
    # Selección de la nueva población
    nuevaPoblacion <- poblacion[order(sapply(poblacion, function(sol)
    ↪  calcularAptitud(sol, core)), decreasing = TRUE)][1:elitismo]

    if (core$funcionObjetivo(nuevaPoblacion[[1]]) - costeAnterior < 1e-6) {
      iteracionesSinCambios <- iteracionesSinCambios + 1
      if (iteracionesSinCambios == maxIterSinCambios) break
    } else {
      iteracionesSinCambios <- 0
      costeAnterior <- core$funcionObjetivo(nuevaPoblacion[[1]])
    }

    listaCostes <- c(listaCostes, core$funcionObjetivo(nuevaPoblacion[[1]]))
    listaDiversidad <- c(listaDiversidad, length(unique(sapply(poblacion, paste,
    ↪  collapse = ""))))
    iterar <- 1
    while (length(nuevaPoblacion) < tamPoblacion) {
      padres <- seleccion(poblacion, core)
```

```
      hijos <- crucePMX(padres[[1]], padres[[2]], core)

      nuevaPoblacion <- c(nuevaPoblacion, list(aplicarMutacion(hijos[[1]]),
      ↪  aplicarMutacion(hijos[[2]])))

    }

    poblacion <- nuevaPoblacion[1:tamPoblacion]
  }

  mejorSolucion <- poblacion[[which.max(sapply(poblacion, function(sol)
  ↪  calcularAptitud(sol, core)))]]
  mejorCoste <- core$funcionObjetivo(mejorSolucion)

  # Tiempo final
  fin <- Sys.time()

  return(list(
    mejorSolucion = mejorSolucion,
    mejorCoste = mejorCoste,
    tiempoCPU = difftime(fin, inicio, units = "secs"),
    listaCostes = listaCostes,
    listaDiversidad = listaDiversidad
  ))
}

# Calcular la aptitud (inversa del coste)
calcularAptitud <- function(solucion, core) {
  distanciaTotal <- core$funcionObjetivo(solucion)
  return(1 / distanciaTotal)
}

# Generar la población inicial
generarPoblacionInicial <- function(tamPoblacion, core) {
  poblacion <- list()
  for (i in seq_len(tamPoblacion)) {
    solucion <- sample(seq_len(core$numCiudades))
    poblacion[[i]] <- solucion
```

```
  }
  return(poblacion)
}

# Selección por torneo
seleccion <- function(poblacion, core) {
  tamTorneo <- 3
  torneo <- sample(poblacion, tamTorneo, replace = FALSE)
  indices <- order(sapply(torneo, function(sol) calcularAptitud(sol, core)),
  ↪   decreasing = TRUE)[1:2]
  padres <- torneo[indices]

  return(list(padres[[1]], padres[[2]]))
}

# Cruce PMX (Partial-Mapped Crossover)
crucePMX <- function(padre1, padre2, core) {
  # Generar puntos de cruce
  puntos <- sort(sample(seq_len(core$numCiudades), 2))
  punto1 <- puntos[1]
  punto2 <- puntos[2]

  hijo1 <- padre1
  hijo2 <- padre2

  mapeo <- setNames(padre2[punto1:punto2], padre1[punto1:punto2])

  for (i in names(mapeo)) {
    i <- as.numeric(i)

    if (i %in% hijo1) {
      indice1 <- which(hijo1 == i)
      valor_map <- mapeo[[as.character(i)]]

      if (valor_map %in% hijo1) {
        indice2 <- which(hijo1 == valor_map)
        hijo1[indice2] <- i
      }
```

```
      hijo1[indice1] <- valor_map
    }

    if (i %in% hijo2) {
      indice1 <- which(hijo2 == i)
      valor_map <- mapeo[[as.character(i)]]

      if (valor_map %in% hijo2) {
        indice2 <- which(hijo2 == valor_map)
        hijo2[indice2] <- i
      }
      hijo2[indice1] <- valor_map
    }
  }


  return(list(hijo1 = hijo1, hijo2 = hijo2))
}

# Mutación por intercambio
aplicarMutacion <- function(gen) {
  if (runif(1) < 0.5) {
    indices <- sample(seq_along(gen), 2)
    tmp <- gen[indices[1]]
    gen[indices[1]] <- gen[indices[2]]
    gen[indices[2]] <- tmp
  }
  return(gen)
}
```

I.5. aco.R

```
optimizacion_colonia_hormigas <- function(matrizDistancias, numHormigas,
↪  maxIteraciones, tasaEvaporacion, alfa, beta) {
  core <- crear_core_tsp(matrizDistancias)

  inicio <- Sys.time()
```

```
# Inicializar matriz de feromonas
matrizFeromonas <- matrix(1, nrow = core$numCiudades, ncol = core$numCiudades)

mejorSolucion <- NULL
mejorCoste <- Inf
listaCosteMedio <- numeric(maxIteraciones)  # Preasignamos para almacenar el
↪  coste medio

for (iter in seq_len(maxIteraciones)) {
  soluciones <- vector("list", numHormigas)
  costes <- numeric(numHormigas)

  # Construir soluciones para todas las hormigas
  for (hormiga in seq_len(numHormigas)) {
    solucion <- construirSolucion(core, matrizFeromonas, matrizDistancias,
    ↪  alfa, beta)
    coste <- core$funcionObjetivo(solucion)

    soluciones[[hormiga]] <- solucion
    costes[hormiga] <- coste

    # Actualizar la mejor solución
    if (coste < mejorCoste) {
      mejorCoste <- coste
      mejorSolucion <- solucion
    }
  }

  # Actualizar la matriz de feromonas de forma vectorizada
  matrizFeromonas <- actualizarMatrizFeromonas(soluciones, costes,
  ↪  matrizFeromonas, tasaEvaporacion)

  # Guardar el coste promedio de esta iteración
  listaCosteMedio[iter] <- mean(costes)
}

fin <- Sys.time()
```

```
  return(list(
    mejorSolucion = mejorSolucion,
    mejorCoste = mejorCoste,
    listaCosteMedio = listaCosteMedio,
    matrizFeromonas = matrizFeromonas,
    tiempoCPU = difftime(fin, inicio, units = "secs")
  ))
}

# Construir una solución
construirSolucion <- function(core, matrizFeromonas, matrizDistancias, alfa,
↪  beta) {
  solucion <- numeric(core$numCiudades)
  visitadas <- logical(core$numCiudades)
  solucion[1] <- sample(seq_len(core$numCiudades), 1)  # Ciudad inicial
  ↪  aleatoria
  visitadas[solucion[1]] <- TRUE

  for (i in 2:core$numCiudades) {
    ciudadActual <- solucion[i - 1]
    solucion[i] <- seleccionarSiguienteCiudad(core, ciudadActual, visitadas,
    ↪  alfa, beta, matrizFeromonas, matrizDistancias)
    visitadas[solucion[i]] <- TRUE
  }

  return(solucion)
}

# Seleccionar la siguiente ciudad
seleccionarSiguienteCiudad <- function(core, ciudadActual, visitadas, alfa,
↪  beta, matrizFeromonas, matrizDistancias) {

  # Calcular probabilidades para todas las ciudades en una línea
  probabilidades <- ifelse(
    visitadas,
    0,  # Asignar 0 si la ciudad ya fue visitada
```

```
    (matrizFeromonas[ciudadActual, ]^alfa) * ((1 /
    ↪  matrizDistancias[ciudadActual, ])^beta)
  )

  # Normalizar las probabilidades
  probabilidades <- probabilidades / sum(probabilidades)

  # Seleccionar una ciudad no visitada según las probabilidades
  return(sample(seq_len(core$numCiudades), 1, prob = probabilidades))

}

# Actualizar la matriz de feromonas
actualizarMatrizFeromonas <- function(soluciones, costes, matrizFeromonas,
↪  tasaEvaporacion) {
  # Evaporación de las feromonas de todas las rutas
  matrizFeromonas <- matrizFeromonas * (1 - tasaEvaporacion)

  # Actualización de feromonas basadas en las soluciones recorridas
  for (i in seq_along(soluciones)) {
    solucion <- soluciones[[i]]
    coste <- costes[i]
    for (j in seq_len(length(solucion) - 1)) {
      ciudad1 <- solucion[j]
      ciudad2 <- solucion[j + 1]
      matrizFeromonas[ciudad1, ciudad2] <- matrizFeromonas[ciudad1, ciudad2] +
      ↪  (1 / coste)
    }
  }

  return(matrizFeromonas)
}
```

Anexo II

Tablas

II.1. Resultados búsqueda tabú

Instancia	Numero de ciudades	Coste	Tiempo
gr21	21	3235	0.3506 s
swiss42	42	1418	1.4912 s
brazil58	58	29178	3.1843 s
eil101	101	711	11.6203 s
d493	493	40274	631.9568 s
rat783	783	10765	2875.1402 s
vm1748	1748	NaN	NaN
rl1889	1889	NaN	NaN
pla7397	7397	NaN	NaN

II.2. Resultados templado simulado

Instancia	**Numero de ciudades**	**Coste**	**Tiempo**
gr21	21	3187	0.0397 s
gr21	21	3224	0.0553 s
gr21	21	3042	0.0393 s
gr21	21	3271	0.0392 s
gr21	21	3028	0.0411 s
gr21	21	2707	0.0390 s
gr21	21	3022	0.0392 s

Instancia	**Numero de ciudades**	**Coste**	**Tiempo**
gr21	21	2709	0.0390 s
gr21	21	3565	0.0393 s
gr21	21	3283	0.0391 s
gr21	**21**	**3103.8**	**0.0410 s**
swiss42	42	2020	0.0414 s
swiss42	42	2274	0.0402 s
swiss42	42	2127	0.0403 s
swiss42	42	2005	0.0402 s
swiss42	42	1984	0.0431 s
swiss42	42	2094	0.0404 s
swiss42	42	1980	0.0405 s
swiss42	42	2007	0.0401 s
swiss42	42	2061	0.0399 s
swiss42	42	2155	0.0401 s
swiss42	**42**	**2070.7**	**0.0406 s**
brazil58	58	29404	0.0543 s
brazil58	58	29761	0.0421 s
brazil58	58	29176	0.0419 s
brazil58	58	29850	0.0422 s
brazil58	58	29959	0.0425 s
brazil58	58	29176	0.0422 s
brazil58	58	29944	0.0455 s
brazil58	58	29833	0.0419 s
brazil58	58	29683	0.0418 s
brazil58	58	29577	0.0418 s
brazil58	**58**	**29636.3**	**0.0436 s**
eil101	101	2980	0.0211 s
eil101	101	3093	0.0177 s
eil101	101	2815	0.0207 s
eil101	101	3024	0.0177 s
eil101	101	3378	0.0206 s
eil101	101	3248	0.0207 s
eil101	101	3055	0.0177 s
eil101	101	3021	0.0207 s
eil101	101	3248	0.0177 s
eil101	101	3252	0.0206 s

Instancia	**Numero de ciudades**	**Coste**	**Tiempo**
eil101	**101**	**3111.4**	**0.0195 s**
d493	493	41896	0.0325 s
d493	493	41959	0.0319 s
d493	493	42704	0.0316 s
d493	493	42599	0.0314 s
d493	493	42424	0.0331 s
d493	493	42250	0.0310 s
d493	493	42671	0.0304 s
d493	493	42170	0.0316 s
d493	493	42745	0.0315 s
d493	493	42609	0.0323 s
d493	**493**	**42402.7**	**0.0317 s**
rat783	783	12384	0.0444 s
rat783	783	14574	0.0432 s
rat783	783	14512	0.0452 s
rat783	783	13780	0.0437 s
rat783	783	13483	0.0431 s
rat783	783	13553	0.0426 s
rat783	783	12816	0.0421 s
rat783	783	15119	0.0423 s
rat783	783	13685	0.0419 s
rat783	783	13852	0.0425 s
rat783	**783**	**13775.8**	**0.0431 s**
vm1748	1748	408102	0.1142 s
vm1748	1748	408184	0.1140 s
vm1748	1748	408102	0.1099 s
vm1748	1748	408365	0.0946 s
vm1748	1748	408102	0.1137 s
vm1748	1748	408102	0.0952 s
vm1748	1748	408102	0.1123 s
vm1748	1748	408102	0.0926 s
vm1748	1748	408102	0.1110 s
vm1748	1748	408102	0.0935 s
vm1748	**1748**	**408136.5**	**0.1051 s**
rl1889	1889	389425	0.1061 s
rl1889	1889	389270	0.1244 s

Instancia	Numero de ciudades	Coste	Tiempo
rl1889	1889	389270	0.1054 s
rl1889	1889	389270	0.1054 s
rl1889	1889	389261	0.1060 s
rl1889	1889	389270	0.1071 s
rl1889	1889	389270	0.1032 s
rl1889	1889	389270	0.1021 s
rl1889	1889	389270	0.1051 s
rl1889	1889	389270	0.1055 s
rl1889	**1889**	**389284.6**	**0.1070 s**

II.3. Resultados algoritmo genético

Instancia	Numero de ciudades	Coste	Tiempo
gr21	21	4277	0.7189 s
gr21	21	3487	0.7405 s
gr21	21	4281	0.7150 s
gr21	21	3501	0.7121 s
gr21	21	3559	0.7091 s
gr21	21	3887	0.7165 s
gr21	21	3862	0.7120 s
gr21	21	3768	0.7332 s
gr21	21	3852	0.7127 s
gr21	21	3344	0.7131 s
gr21	**21**	**3781.8**	**0.7183 s**
swiss42	42	2114	1.1716 s
swiss42	42	2327	1.1598 s
swiss42	42	2532	1.1710 s
swiss42	42	2216	1.2057 s
swiss42	42	2790	1.1828 s
swiss42	42	2699	1.1836 s
swiss42	42	2259	1.2127 s
swiss42	42	2443	1.2081 s
swiss42	42	2616	1.1901 s
swiss42	42	2672	1.1782 s

Instancia	Numero de ciudades	Coste	Tiempo
swiss42	**42**	**2466.8**	**1.1864 s**
brazil58	58	63942	1.4948 s
brazil58	58	64498	1.4389 s
brazil58	58	62208	1.4678 s
brazil58	58	64518	1.4692 s
brazil58	58	67015	1.4332 s
brazil58	58	58406	1.4325 s
brazil58	58	71493	1.4482 s
brazil58	58	61624	1.4447 s
brazil58	58	69466	1.4162 s
brazil58	58	59517	1.4361 s
brazil58	**58**	**64268.7**	**1.4482 s**
eil101	101	2154	2.3123 s
eil101	101	2249	2.3476 s
eil101	101	2046	2.3920 s
eil101	101	2207	2.3456 s
eil101	101	2013	2.3524 s
eil101	101	2079	2.3657 s
eil101	101	1934	2.3029 s
eil101	101	2082	2.3498 s
eil101	101	2080	2.3559 s
eil101	101	2051	2.3330 s
eil101	**101**	**2089.5**	**2.3457 s**
d493	493	325852	14.4483 s
d493	493	342729	14.8932 s
d493	493	345007	14.6421 s
d493	493	360809	14.8290 s
d493	493	359003	14.7410 s
d493	493	334220	14.3449 s
d493	493	353793	14.5610 s
d493	493	357111	14.3304 s
d493	493	344747	14.5646 s
d493	493	334266	14.7675 s
d493	**493**	**345753.7**	**14.6122 s**
rat783	783	145635	27.7830 s
rat783	783	138948	27.4090 s

Instancia	**Numero de ciudades**	**Coste**	**Tiempo**
rat783	783	146878	27.5916 s
rat783	783	146936	27.4517 s
rat783	783	139768	28.1861 s
rat783	783	149487	27.6365 s
rat783	783	144831	27.1987 s
rat783	783	142068	26.7346 s
rat783	783	141870	27.6375 s
rat783	783	142230	26.6626 s
rat783	**783**	**143865.1**	**27.4291 s**
vm1748	1748	12964555	99.0749 s
vm1748	1748	13132112	98.0372 s
vm1748	1748	13041723	96.7814 s
vm1748	1748	13162911	97.6930 s
vm1748	1748	12937450	98.6296 s
vm1748	1748	12907323	99.0362 s
vm1748	1748	12746069	98.9399 s
vm1748	1748	13713897	100.4360 s
vm1748	1748	13241972	99.9386 s
vm1748	1748	13089198	94.5042 s
vm1748	**1748**	**13093721**	**98.3071 s**
rl1889	1889	13040056	114.7976 s
rl1889	1889	12908591	112.9227 s
rl1889	1889	12729194	114.3345 s
rl1889	1889	12918641	110.4529 s
rl1889	1889	13073494	111.3346 s
rl1889	1889	13194478	116.4296 s
rl1889	1889	12927679	111.8917 s
rl1889	1889	12821379	112.9199 s
rl1889	1889	12966997	113.4594 s
rl1889	1889	12958895	115.1654 s
rl1889	**1889**	**12953940.4**	**113.3708 s**

II.4. Resultados optimización de la colonia de hormigas

Instancia	Numero de ciudades	Coste	Tiempo
gr21	21	2707	2.1510 s
gr21	21	2758	2.1184 s
gr21	21	2707	2.0409 s
gr21	21	2709	2.0948 s
gr21	21	2707	2.0404 s
gr21	21	2707	2.0552 s
gr21	21	2707	2.0295 s
gr21	21	2795	2.0377 s
gr21	21	2707	1.9909 s
gr21	21	2795	2.0087 s
gr21	**21**	**2729.9**	**2.0568 s**
swiss42	42	1291	6.6922 s
swiss42	42	1293	6.6901 s
swiss42	42	1309	6.6876 s
swiss42	42	1349	6.6562 s
swiss42	42	1286	6.7327 s
swiss42	42	1304	6.8697 s
swiss42	42	1308	7.1459 s
swiss42	42	1303	7.0700 s
swiss42	42	1315	7.0060 s
swiss42	42	1287	6.9980 s
swiss42	**42**	**1304.5**	**6.8548 s**
brazil58	58	26115	11.8698 s
brazil58	58	25993	11.8231 s
brazil58	58	26276	13.4303 s
brazil58	58	26284	14.6522 s
brazil58	58	25906	15.1313 s
brazil58	58	25993	12.9738 s
brazil58	58	26161	11.8114 s
brazil58	58	25876	11.7411 s
brazil58	58	25769	11.8327 s
brazil58	58	26284	11.6123 s
brazil58	**58**	**26065.7**	**12.6878 s**
eil101	101	699	33.9193 s
eil101	101	698	32.9963 s
eil101	101	696	33.9302 s

Instancia	Numero de ciudades	Coste	Tiempo
eil101	101	694	34.1524 s
eil101	101	690	38.3350 s
eil101	101	685	34.0744 s
eil101	101	701	35.6417 s
eil101	101	705	35.3879 s
eil101	101	668	32.8283 s
eil101	101	700	34.1222 s
eil101	**101**	**693.6**	**34.5388 s**
d493	493	40617	822.0743 s
d493	493	41974	947.8438 s
d493	493	42497	1100.2957 s
d493	493	41733	972.6206 s
d493	493	41579	820.4895 s
d493	493	41634	871.2218 s
d493	493	41744	978.2920 s
d493	493	41434	1158.8328 s
d493	493	41506	818.7255 s
d493	493	41834	818.2062 s
d493	**493**	**41655.2**	**930.8602 s**
rat783	783	11118	2771.3868 s
rat783	783	10929	2711.5941 s
rat783	783	11200	2733.2714 s
rat783	783	11339	2750.8232 s
rat783	783	11353	2705.6146 s
rat783	783	11168	2750.8217 s
rat783	**783**	**11184.5**	**2737.2520 s**

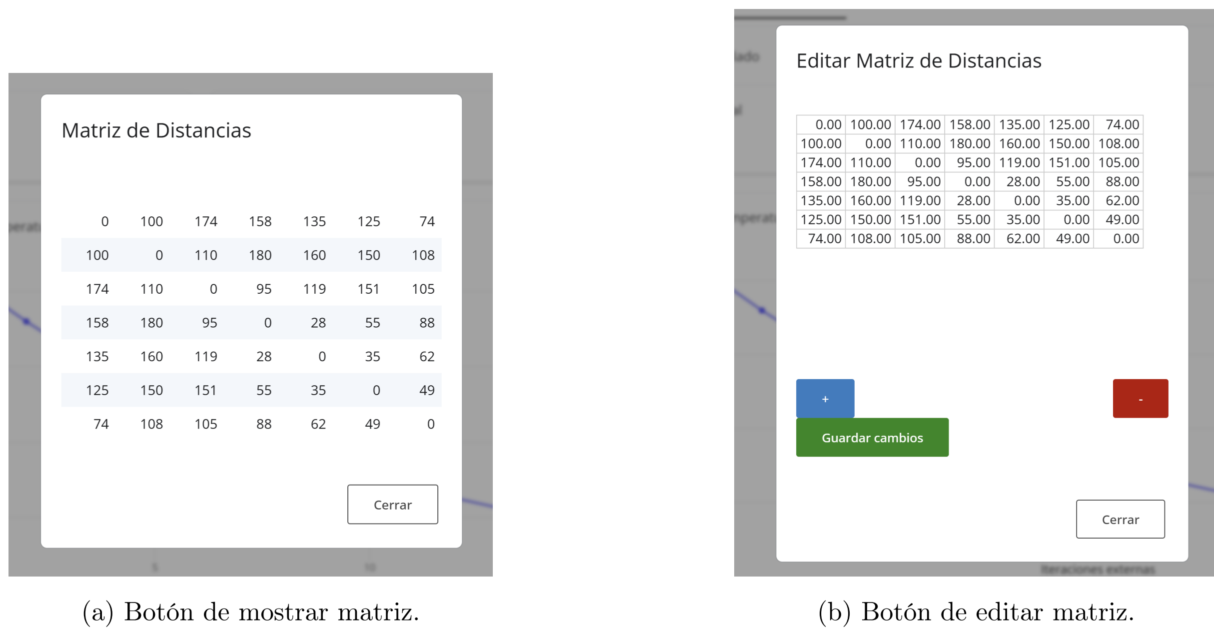
Anexo III

Exploración visual de parámetros en el problema del viajante

Para ayudar a visualizar, como los diferentes parámetros de cada metaheurística influyen en el resultado, se ha confeccionado una interfaz sencilla que se basa en el código ya estudiado. El objetivo es que los usuarios sin una base de programación sólida puedan utilizar el código ya tratado para sus propias instancias del TSP, de forma fácil. La herramienta fue creada con Shiny que es un paquete que permite crear aplicaciones de visualización de datos de forma rápida. La interfaz se puede usar en el siguiente enlace: https://elena24.shinyapps.io/TFGM/

Para usar esta interfaz, simplemente hay que introducir la matriz de distancias que se desea utilizar o usar la que viene por defecto. Los paneles de visualización y de edición de la matriz se pueden ver en la Figura III.1.

Posteriormente, se seleccionaría la metaheurística que se desee en el panel superior. Por último, se seleccionarán en el panel derecho los parámetros y se observará el resultado en la parte izquierda. Un ejemplo de cómo se ve la interfaz se puede observar en la Figura III.2.



(a) Botón de mostrar matriz.

(b) Botón de editar matriz.

Figura III.1: Visualización y edición de la matriz de distancias.

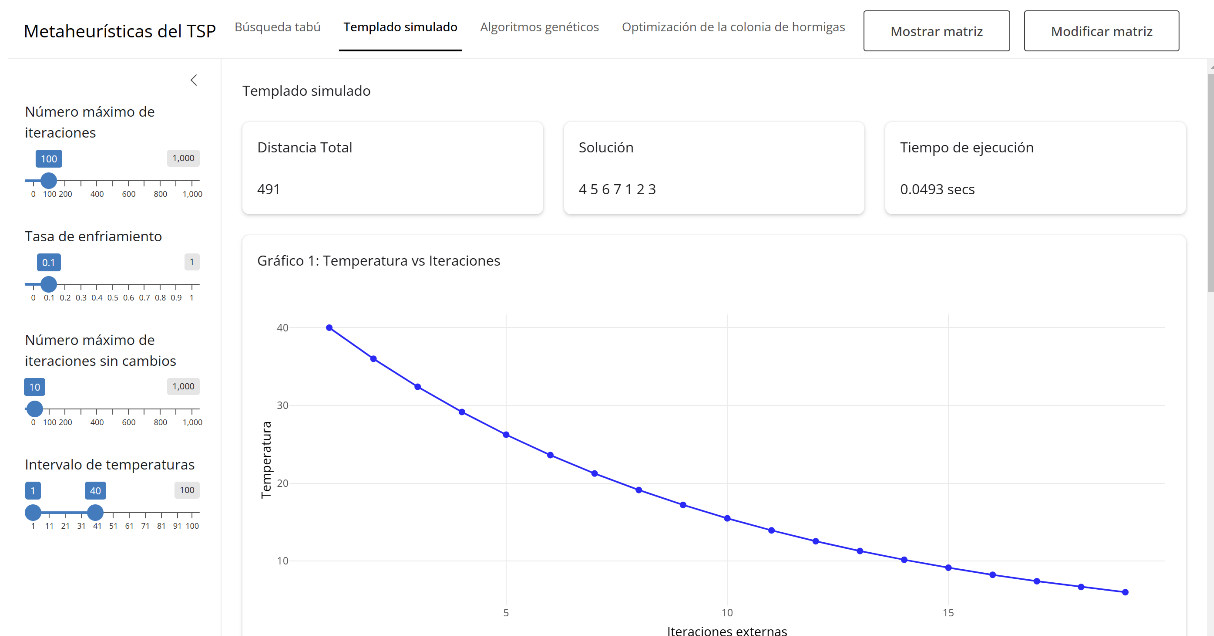


Figura III.2: Ejemplo de la interfaz.

Bibliografía

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall.

Anily, S., & Federgruen, A. (1987). Simulated annealing methods with general acceptance probabilities. *Journal of Applied Probability*, *24*(3), 657-667.

Applegate, D. L. (2006). *The Traveling Salesman Problem: a Computational Study*. Princeton University Press.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. *International Conference on Genetic Algorithms*, 14-21.

Çinlar, E. (1975). *Introduction to Stochastic Processes*. Prentice-Hall.

Dorigo, M., & Gambardella, L. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, *1*(1), 53-66.

Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms* [Tesis doctoral, Politecnico di Milano].

Gendreau, M., & Potvin, J.-Y. (2010). *Handbook of Metaheuristics*. Springer.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, *13*(5), 533-549.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems* [second edition, 1992]. University of Michigan Press.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science (American Association for the Advancement of Science)*, *220*(4598), 671-680.

Korte, B. H., & Vygen, J. (2002). *Combinatorial Optimization*. Springer Science; Business Media.

Mitra, D., Romeo, F., & Sangiovanni-Vincentelli, A. (1986). Convergence and finite-time behavior of simulated annealing. *Advances in Applied Probability*, *18*(3), 747-771.

Reeves, C. R. (1993). Using Genetic Algorithms with Small Populations. *Proceedings of the 5th International Conference on Genetic Algorithms*, 92-99.

Reina, D. G., Córdoba, A. T., & Del Nozal, A. R. (2020). *Algoritmos Genéticos con Python.* Marcombo.

Reinelt, G. (s.f.). TSPLIB. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/