Universidad de Santiago de Compostela

Facultad de Matemáticas

Máster en Técnicas Estadísticas

TRABAJO FIN DE MÁSTER

ALGORITMOS HEURÍSTICOS EN OPTIMIZACIÓN

Realizado por Aitana Vidal Esmorís

Dirigido por
María Luisa Carpente Rodríguez
Balbina Casas Méndez

1 de Julio de 2013

El presente documento recoge el Trabajo Fin de Máster para el Máster Interuniversitario en Técnicas Estadísticas, realizado por Doña Aitana Vidal Esmorís bajo el título “Algoritmos Heurísticos en Optimización”.

Ha sido realizado bajo la dirección de Doña María Luisa Carpente Rodríguez y Doña Balbina Virginia Casas Méndez, que lo consideran terminado y dan su conformidad para su presentación y defensa.

Santiago de compostela, a 1 de julio de 2013.

Fdo.: Mª Luisa Carpente Rodríguez   Fdo.: Balbina Virginia Casas Méndez

Fdo.: Aitana Vidal Esmorís

Índice general

Resumen **7**

1. Introducción a la Optimización **9**
1.1. ¿Qué es la Investigación Operativa? 9
1.2. Evolución histórica 10
1.3. Problemas de optimización y su complejidad 12
1.4. Algoritmos y complejidad computacional 18
1.4.1. Tipos de algoritmos 19
1.4.2. Medición de la complejidad de un algoritmo 20
1.5. Técnicas de resolución. Clasificación 22

2. Técnicas heurísticas de optimización **27**
2.1. Algoritmos exactos y heurísticas 27
2.2. Búsqueda Exhaustiva 33
2.2.1. Aplicación al ejemplo de referencia 33
2.3. Búsqueda Local 34
2.3.1. Introducción 34
2.3.2. Descripción del algoritmo 36
2.3.3. Aplicación al ejemplo de referencia 37
2.4. Búsqueda Tabú 38
2.4.1. Introducción 38
2.4.2. Descripción del algoritmo 40
2.4.3. Aplicación al ejemplo de referencia 42
2.5. Simulated Annealing 45
2.5.1. Introducción 45
2.5.2. Descripción del algoritmo 46
2.5.3. Aplicación al ejemplo de referencia 47
2.6. Algoritmos genéticos 50
2.6.1. Introducción 50
2.6.2. Descripción del algoritmo 50
2.6.3. Aplicación al ejemplo de referencia 54
2.7. Colonias de hormigas 56
2.7.1. Introducción 56

2.7.2. Descripción del algoritmo 57
2.7.3. Aplicación al ejemplo de referencia 60

3. Aplicación a un ejemplo real 67
3.1. Definición del problema 67
3.2. Solución del problema 70
3.3. Resolviendo el TSP 75

4. Conclusiones 83

A. Código de programación en MATLAB 85

Resumen

Dado que gran parte de los problemas prácticos que se necesitan resolver son de los denominados NP-hard, los algoritmos heurísticos o de aproximación desempeñan un papel importante en la resolución de problemas de optimización discreta con la idea de encontrar buenas soluciones factibles de manera rápida. Se pretende fundamentalmente realizar una revisión de algunos de los heurísticos más utilizados recientemente en la literatura como son la búsqueda local, búsqueda tabú, *simulated annealing*, algoritmos genéticos y algoritmos de hormigas.

Esta memoria comienza con una breve introducción a la Investigación Operativa que recoge la definición, contenido y evolución histórica. Seguidamente se presentarán algunos problemas de optimización y su complejidad, entre ellos, el problema del viajante, problemas de asignación y planificación, así como las técnicas clásicas de resolución más utilizadas.

En el siguiente apartado, se explicarán detalladamente cada uno de los métodos heurísticos de resolución de problemas de optimización que aquí se presentan. Mediante un pequeño ejemplo, común a todos ellos, se tratará de comprender las diferentes técnicas utilizadas y, finalmente, sobre un ejemplo de mayor envergadura discutiremos los resultados obtenidos en función de su valía para llegar a la mejor solución posible.

Capítulo 1

Introducción a la Optimización

1.1. ¿Qué es la Investigación Operativa?

Definición 1. *Investigación Operativa.*

*La **Investigación Operativa**, abreviando **IO**, es una ciencia moderna interdisciplinaria que mediante la aplicación de teoría, métodos y técnicas especiales, busca la "solución óptima" de problemas complejos de administración, organización y control que se producen en los diversos sistemas existentes en la naturaleza y los creados por el ser humano (sistemas organizados, sistemas físicos, económicos, ecológicos, educacionales, de servicio social, etc), permitiendo de esta forma la "toma de decisiones".*

Objetivo

El objetivo más importante de la aplicación de la investigación operativa es dar apoyo en la "toma óptima de decisiones" en los sistemas y de la planificación de sus actividades.

Enfoque

El enfoque fundamental de la investigación operativa es el enfoque de sistemas, por el cual, a diferencia del enfoque tradicional, se estudia el comportamiento de todo un conjunto de partes o sub-sistemas que interaccionan entre sí. De esta forma se identifica el problema y se analizan sus repercusiones, buscándose soluciones integrales que beneficien al sistema como un todo.

Técnica

Para hallar la solución, generalmente se representa el problema como un modelo matemático, que es analizado y evaluado previamente.

Actualmente, la Investigación Operativa incluye gran cantidad de ramas como la Programación Lineal, Programación no Lineal, Programación Dinámica, Simulación, Teoría de Colas, Teoría de Inventarios, Teoría de Grafos, etc.

1.2. Evolución histórica

Algunos historiadores consideran que el punto inicial de la Investigación de Operaciones radica en el comienzo de la segunda Guerra Mundial, tras el interés mostrado por un grupo de investigadores militares encabezados por A. P. Rowe en el uso militar de una técnica conocida como radiolocalización.

Otros consideran que su comienzo está en el análisis y solución del bloqueo naval de Siracusa que Arquímedes presentara al tirano de esa ciudad, en el siglo III A.C.

F. W. Lanchester, en Inglaterra, justo antes de la primera guerra mundial, desarrolló relaciones matemáticas sobre la potencia balística de las fuerzas opositoras, que si se resolvían tomando en cuenta el tiempo, podían determinar el resultado de un encuentro militar.

Tomás Edison también realizó estudios de guerra antisubmarina.

Ni los estudios de Lanchester ni los de Edison tuvieron un impacto inmediato. Dichos estudios, junto con los de Arquímedes, constituyen viejos ejemplos del empleo de científicos para determinar la decisión óptima en las guerras, optimizando los ataques.

Poco después de que estallara la Segunda Guerra Mundial, la Badswey Research Station, bajo la dirección de Rowe, participó en el diseño de utilización óptima de un nuevo sistema de detección y advertencia prematura, denominado radar (RAdio Detection And Ranging, detección y medición de distancias mediante radio). Este avance sirvió para el análisis de todas las fases de las operaciones nocturnas, y el estudio se constituyó en un modelo de los estudios de investigación de operaciones que siguieron.

En agosto de 1940, se organizó un grupo de 20 investigadores ("Circo de Blackett") bajo la dirección de P. M. S. Blackett, de la Universidad de Manchester, para estudiar el uso de un nuevo sistema antiaéreo controlado por radar. El grupo estaba formado, entre otros, por tres fisiólogos, dos fisicomatemáticos, un astrofísico, un oficial del ejército, un topógrafo, un físico general y dos matemáticos. Parece aceptarse comúnmente que la formación de este grupo constituye el inicio de la investigación de operaciones.

Blackett y parte de su grupo, participaron en 1941 en problemas de detección de barcos y submarinos mediante un radar autotransportado. Este estudio condujo a que Blackett fuera nombrado director de Investigación de Operación Naval del Almirantazgo Británico. Posteriormente, la parte restante de su equipo pasó a ser el grupo de Investigación de Operaciones

de la Plana de Investigación y Desarrollo de la Defensa Aérea, aunque luego se dividió de nuevo para formar el Grupo de Investigación de Operaciones del Ejército. Después de la guerra, los tres servicios tenían grupos de investigación de operaciones.

Como ejemplo de esos primeros estudios está el que planteó la Comandancia Costera que no lograba hundir submarinos enemigos con una nueva bomba antisubmarina. Las bombas se preparaban para explotar a profundidades de no menos de 30 m. Después de estudios detallados, un profesor apellidado Williams llegó a la conclusión de que la máxima probabilidad de muerte ocurriría con ajustes para profundidades entre 6 y 7 m. Entonces se prepararon las bombas para mínima profundidad posible de 10 m, y los aumentos en las tasas de muertes, según distintas estimaciones, se incrementaron entre un 400 y un 700 por ciento. De inmediato se inició el desarrollo de un mecanismo de disparo que se pudiera ajustar a la profundidad óptima de 6 a 7m. Otro problema que consideró el Almirantazgo fueron las ventajas de los convoyes grandes frente a los pequeños. Los resultados fueron a favor de los convoyes grandes.

A pocos meses de que Estados Unidos entrara en la guerra, en la fuerza aérea del ejército y en la marina se iniciaron actividades de investigación de operaciones. Para el Día D (invasión aliada de Normandía), en la fuerza aérea se habían formado veintiséis grupos de investigación de operaciones, cada uno con aproximadamente diez científicos. En la marina se dio un proceso semejante. En 1942, Philip M. Morris, del Instituto Tecnológico de Massachussets, encabezó un grupo para analizar los datos de ataque marino y aéreo en contra de los submarinos alemanes. Luego se emprendió otro estudio para determinar la mejor política de maniobrabilidad de los barcos en convoyes a fin de evadir aeroplanos enemigos, e incluso los efectos de la exactitud antiaérea. Los resultados del estudio demostraron que los barcos pequeños deberían cambiar su dirección gradualmente.

Al principio, la investigación de operaciones se refería a sistemas existentes de armas y a través del análisis, típicamente matemático, se buscaban las políticas óptimas para la utilización de esos sistemas. Hoy en día, la investigación de operaciones todavía realiza esta función dentro de la esfera militar. Sin embargo, lo que es mucho más importante, ahora se analizan las necesidades del sistema de operación con modelos matemáticos y se diseña un sistema (o sistemas) de operación que ofrezca la capacidad óptima.

El éxito de la investigación de operaciones en la esfera de lo militar quedó bastante bien documentado hacia finales de la Segunda Guerra Mundial. El general Arnold encargó a Donald Douglas, de la Douglas Aircraft Corporation, en 1946, la dirección de un proyecto, Research ANd Development (RAND, Investigación y Desarrollo), para la Fuerza Aérea. La corporación RAND desempeña hoy día un papel importante en la investigación que se lleva a cabo en la Fuerza Aérea.

A partir del inicio de la investigación de operaciones como disciplina, sus características más comunes son:

- Enfoque de sistemas
- Modelado matemático
- Enfoque de equipo

Estas características prevalecieron a ambos lados del Atlántico, a partir del desarrollo de la investigación de operaciones durante la Segunda Guerra Mundial.

Para maximizar la capacidad militar de entonces, fue necesario un enfoque de sistemas. Ya no era tiempo de tomar decisiones de alto nivel sobre la dirección de una guerra que exigía sistemas complicados frente a la estrategia de guerras anteriores o como si se tratara de un juego de ajedrez.

La computadora digital y el enfoque de sistemas fueron preludios necesarios del procedimiento matemático de los sistemas militares de operaciones. Las matemáticas aplicadas habían demostrado su utilidad en el análisis de sistemas económicos, y el uso de la investigación de operaciones en el análisis de sistemas demostró igualmente su utilidad.

Para que un análisis de un sistema militar de operaciones fuera tecnológicamente factible, era necesario tener una comprensión técnica adecuada, que tomara en cuenta todas las subcomponentes del sistema. En consecuencia, el trabajo de equipo resultó ser tan necesario como efectivo.

1.3. Problemas de optimización y su complejidad

Los problemas de optimización aparecen en multitud de ámbitos, desde la elección del camino más corto para ir al trabajo hasta el diseño óptimo de rutas de reparto de mercancías de una gran multinacional.

La optimización es una parte importante de la Investigación Operativa. Algunas técnicas como la Programación lineal (LP) o la Programación dinámica (DP) son anteriores a 1960:

- Método Simplex, LP, por Dantzig (1947)
- Principio de optimalidad, DP, por Bellman (1957)

Además, podemos encontrar un manual de dichos métodos en Hillier et al (2005) y Denardo (1982), respectivamente.

Características de la Optimización:

- Proceso algorítmico inicial muy rápido
- Elección de la mejor alternativa entre todas las alternativas posibles

Sin embargo, ciertas técnicas de investigación de operaciones se enumeran bajo el nombre de optimización o programación matemática.

Definimos a continuación algunos conceptos previos que serán de gran utilidad a lo largo de la memoria:

Definición 2. *Problema de optimización.*
*Un **problema de optimización** viene dado por un par (F, c) donde F es un dominio de puntos factibles y c una función coste. El problema consiste en encontrar un punto factible $x \in F$ tal que $\forall y \in F$ verifique, $c(x) \leq c(y)$.*

*Cada punto x verificando las condiciones anteriormente descritas se denomina **óptimo global** del problema.*

Podemos encontrar algunos conceptos básicos comunes a todos los enfoques algorítmicos para la resolución de los problemas. Dependiendo de las técnicas utilizadas necesitaremos especificar:

Espacio de búsqueda

La representación de una solución potencial y su correspondiente interpretación nos da el espacio de búsqueda y su tamaño. Este es un punto clave para el problema: el tamaño de dicho espacio no viene determinado por el problema sino por nuestra representación e interpretación. Este paso es de vital importancia para dar una correcta solución a nuestro problema, además de evitar contratiempos de duplicaciones de resultados, etc.

Vecindad

Si nos centramos en una región $N(x)$ del espacio de búsqueda S, que está "cerca" de algún punto x del espacio, podemos definir $N(x)$ como una vecindad del punto $x \in S$.

Objetivo

El objetivo del problema no es más que el estado matemático que nos interesa aplicar para que la tarea se lleve a cabo. Dependiendo del tipo del problema nos interesará maximizar o minimizar la función objetivo.

Función objetivo

Es la medida cuantitativa del rendimiento del sistema a optimizar (maximizar o minimizar). Algunos ejemplos de las funciones objetivo son: la minimización de los costes de un sistema de producción agrícola, la maximización de los beneficios netos de la venta de ciertos productos, la minimización de los materiales utilizados en la fabricación de un producto, etc.

Variables

Representan las decisiones que se pueden tomar para variar el valor de la función objetivo. Se pueden clasificar como dependientes o independientes. En el caso de un sistema de producción agrícola serán los valores de producción de las unidades generadoras o flujos a través de las líneas. En el

caso de las ventas, la cantidad de producto producido y vendido. En el caso de la fabricación de un producto, sus dimensiones físicas.

Restricciones
Representan el conjunto de relaciones que ciertas variables están obligados a cumplir. Se expresan mediante ecuaciones o desigualdades. Por ejemplo, la capacidad de producción para la fabricación de diferentes productos, las dimensiones de la materia prima del producto, etc.

Una posible clasificación de los métodos de optimización clásicos podría ser la siguiente:

- Programación no lineal
- Programación lineal
- Programación entera
- Programación lineal entera
- Programación dinámica
- Programación estocástica
- Programación multiobjetivo

En la figura 1.1 podemos ver la relación entre algunos de los métodos de optimización enumerados anteriormente. Notemos que en dicha figura no se han incluido los problemas de programación dinámica, los problemas de programación estocástica y los problemas de optimización multiobjetivo.

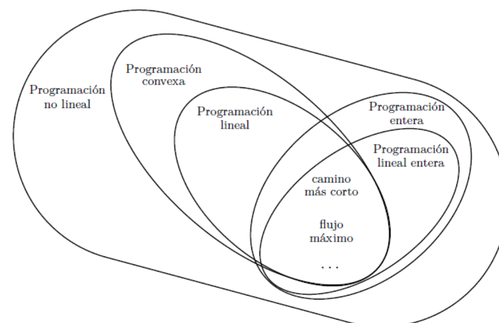

Figura 1.1: Una clasificación de los problemas de optimización

Resolver un problema de optimización consiste en encontrar el valor que deben tomar las variables para hacer óptima la función objetivo satisfaciendo el conjunto de restricciones. A continuación definiremos formalmente algunos conceptos que son de vital importancia al hablar de optimización.

Definición 3. *Programación no lineal.*
*Un problema de **programación no lineal** consiste en encontrar una solución al problema:*

$$\begin{array}{lll} \textit{minimizar} & c(x) & \\ \textit{sujeto a} & g_i(x) \geq 0 & \forall i = 1, ..., m \\ & h_j(x) = 0 & \forall j = 1, ..., l. \end{array}$$

Definición 4. *Conjunto factible*
*El **conjunto factible** asociado a un problema de programación no lineal viene dado por aquellos puntos que verifican las restricciones asociadas a las funciones g_i y h_j .*

Un problema de programación no lineal es un problema de *planificación óptima* dada una función objetivo y una serie de restricciones, las cuales determinan las soluciones factibles de entre las que habremos de encontrar la solución óptima.

A la hora de estudiar los problema de optimización es importante notar que no basta con encontrar un óptimo local, ya que un óptimo local puede estar realmente lejos del óptimo global. En general, verificar la optimalidad global de un óptimo local no es una tarea trivial.

Un problema de programación no lineal en el que la función c es convexa, las funciones g_i son cóncavas y las funciones h_j son lineales se llama *problema de programación convexa*[1]. A continuación enunciamos unos resultados muy importantes que verifica cualquier problema de programación convexa:

Lema 1. *El conjunto factible asociado a un problema de programación convexa es convexo.*[2]

Lema 2. *Dado un problema de programación convexa, todo óptimo local es un óptimo global.*

Otra propiedad interesante de los problemas de programación convexa es que dados dos óptimos, cualquier combinación convexa de ellos también será un óptimo.

Lema 3. *Dado un problema de programación convexa y dos óptimos (globales) x e y, para todo $\lambda \in [0, 1]$, el punto $z = \lambda x + (1 - \lambda)y$ también es un óptimo (global).*

[1] Una función h es lineal si para todo x e y y para todo par de escalares (números reales) a y b, $h(ax + by) = ah(x) + bh(y)$. Una función g es cóncava si para todo x e y y para todo $\lambda \in [0, 1]$, $g(\lambda x + (1 - \lambda)y) \geq \lambda g(x) + (1 - \lambda)g(y)$; es decir, la imagen de una combinación convexa de dos puntos está por encima de la recta que une sus respectivas imágenes. La función es convexa si se cumple la desigualdad contraria.

[2] Un conjunto F es convexo si para todo x, $y \in F$ y todo $\lambda \in [0, 1]$, $\lambda x + (1 - \lambda)y \in F$.

En general, a pesar de los resultados anteriores, el estudio de los problemas de programación convexa puede ser muy complejo ya que, típicamente, el conjunto factible estará formado por un continuo de puntos. De este modo, las técnicas de fuerza bruta son inviables, ya que será prácticamente imposible hacer un barrido de toda la región factible. En este punto podemos introducir los problemas de programación lineal, para los cuales podrían funcionar técnicas enumerativas para encontrar su solución.

Definición 5. *Programación lineal.*
Un problema de ***programación lineal*** *consiste en encontrar una solución al problema:*

$$\begin{array}{ll} \textit{minimizar} & c(x) \\ \textit{sujeto a} & g_i(x) \geq 0 \quad \forall i = 1, ..., m \\ & h_j(x) = 0 \quad \forall j = 1, ..., l. \end{array}$$

siendo c, g_i y h_j funciones lineales.

Como toda función lineal es tanto cóncava como convexa, todo problema de programación lineal es un problema de programación convexa y por tanto todo óptimo local equivale a un óptimo global. Hablar de programación lineal no es más que hablar de "planificación con modelos lineales".

Los problemas de programación lineal se pueden resolver utilizando sólo un conjunto finito de puntos. Hay problemas cuyo conjunto factible es un conjunto finito o numerable de puntos. Estos problemas se llaman problemas combinatorios, ya que para resolverlos es necesario enumerar todas las combinaciones posibles de los valores de cada variable y luego ver dónde se alcanza el punto óptimo. En general, estudiaremos los problemas combinatorios en la optimización lineal entera.

Definición 6. *Programación lineal entera.*
Un problema de ***programación lineal entera*** *consiste en encontrar una solución al problema:*

$$\begin{array}{ll} \textit{minimizar} & c(x) \\ \textit{sujeto a} & g_i(x) \geq 0 \quad \forall i = 1, ..., m \\ & h_j(x) = 0 \quad \forall j = 1, ..., l \\ & x \in \mathbb{Z}^n \end{array}$$

siendo c, g_i y h_j funciones lineales.

En ciertos problemas no podemos representar todas las decisiones mediante variables continuas. Por ejemplo, las decisiones de inversión y la planificación de expansión de una red, o el reclutamiento de las personas, deben estar representados por variables discretas; mientras que las decisiones de localización de plantas o almacenes deberán estar representadas por

variables binarias. De esta forma podemos clasificar los problemas de programación entera según el tipo de variables como:

- Programación entera pura (PIP), si todas las variables son enteras.
- Programación entera binaria (BIP), si todas son binarias.
- Programación entera mixta (MIP), si hay alguna entera o binaria y el resto son continuas.

Un caso particular de variables enteras son las variables binarias, (0/1), porque permiten la modelización por asignación o condiciones lógicas. Además, cualquier variable entera, x, puede expresarse como suma de variables binarias y_i, donde $x = \sum_{i=0}^{N} 2^i y_i$, con $0 \leq x < u$ y $2^N \leq u \leq 2^{N+1}$ siendo u una cota superior de x.

Algunos Problemas de Optimización

A continuación enumeramos una serie de problemas de optimización clásicos que se pueden encontrar en la literatura.

- El problema del viajante de comercio, TSP (*Travelling Salesman Problem*): es uno de los problemas clásicos de optimización combinatoria. Recibió este nombre porque puede describirse en términos de un agente de ventas que debe visitar cierta cantidad de ciudades en un solo viaje. Si comienza desde su ciudad de residencia, el agente debe determinar qué ruta debe seguir para visitar cada ciudad exactamente una vez antes de regresar a su casa de manera que se minimice la longitud total del viaje.

- Problemas asignación: Caso particular del problema de transporte de bienes desde unos orígenes a unos destinos donde los asignados son recursos destinados a la realización de tareas, los asignados pueden ser personas, máquinas, vehículos, plantas o períodos de tiempo. En estos problemas la oferta en cada origen es de valor 1 y la demanda en cada destino es también de valor 1.

- Problemas de rutas, *Vehicle Routing Problems* (VRP). Se trata de diseñar las rutas de una flota de transporte para dar servicio a unos clientes.

- Problemas de flujo máximo: Capacidad en redes. En una red en la que los arcos tienen limitada su capacidad, el problema del flujo máximo consiste en maximizar la cantidad de flujo que se puede pasar entre dos nodos prefijados de la red.

- El problema de la mochila. Supongamos que un excursionista debe decidir qué cosas incluir en su mochila. El problema consiste en decidir

qué objetos incluír, teniendo en cuenta que la mochila soporta un peso máximo, de forma que maximice la utilidad que le proporcionarán los objetos elegidos.

- Problemas de inventario. Se trata de optimizar la gerencia sobre cuánto producir, cuánto demorar y cuánto almacenar en cada período de un horizonte de planficación dado.

Algunas referencias que nos pueden ser de ayuda a la hora de sumergirnos en la programación matemática son, entre otros, Ríos Insua (1996), Michalewicz et al (1998) y Salazar (2001), así como Yang (2010), uno de los textos más actuales para leer sobre este tema.

1.4. Algoritmos y complejidad computacional

Definición 7. *Algoritmo.*
Un ***algoritmo*** *es un conjunto de instrucciones o pasos utilizados para realizar una tarea o resolver un problema. Formalmente, un algoritmo es una secuencia finita de operaciones que se realizan sin ambigüedades, cuya actuación ofrece una solución a un problema.*

Los algoritmos se utilizan para el cálculo, procesamiento de datos y el razonamiento automatizado. En pocas palabras, un algoritmo es un procedimiento paso a paso para los cálculos.

A partir de un estado inicial y una entrada inicial (quizás vacía), las instrucciones describen un cómputo que, cuando se ejecuta, se procederá a través de un número finito de estados sucesivos bien definidos que generalmente producen un "output" y terminan con un estado final. La transición de un estado a otro no es necesariamente determinista; algunos algoritmos, conocidos como algoritmos aleatorios, incorporan una entrada aleatoria.

Los algoritmos son esenciales para procesar los datos. Muchos programas contienen algoritmos que especifican las instrucciones específicas que un equipo debe llevar a cabo (en un orden específico) para realizar una tarea especifica, como por ejemplo el cálculo de las nóminas de los empleados o la impresión de los boletines de calificaciones de los estudiantes. Así, un algoritmo puede ser considerado como cualquier secuencia de operaciones que se puede simular mediante un sistema Turing-completo. Autores que afirman ésta tesis son Minsky (1967), y Gurevich (1999):

Minsky: " Pero también vamos a mantener, con Turing. . . que cualquier procedimiento que podría "naturalmente" denominarse efectivo, de hecho, puede ser realizado por una máquina (simple). Aunque esto puede parecer extremo, los argumentos. . . a su favor son difíciles de rebatir".

Gurevich: "...el argumento informal de Turing [3] justifica en favor a su tesis: cada algoritmo puede ser simulado por una máquina de Turing[4] ... según Savage (1987), un algoritmo es un proceso de cálculo definido por una máquina de Turing"

En la práctica, es interesante encontrar algoritmos cuyo tiempo de ejecución es, al menos, polinomial, ya que los algoritmos no polinomiales pueden ser muy poco útiles para el estudio de algunos problemas importantes. A continuación definimos qué se entiende por velocidad de un algoritmo:

Definición 8. *Velocidad de un algoritmo.*
Decimos que un algoritmo tiene ***velocidad*** $O(f(n))$ *si existen dos constantes* c *y* n_0 *tal que el tiempo necesitado por el algoritmo para resolver cualquier problema de tamaño* $n \geq n_0$ *es, al menos,* $cf(n)$.

1.4.1. Tipos de algoritmos

Podemos establecer varias clasificaciones de algoritmos. En primer lugar, podemos diferenciar entre algoritmos deterministas y algoritmos no deterministas. Un algoritmo es determinista si en un conjunto de problemas, todas las ejecuciones del algoritmo producen el mismo resultado final (y además, todos los resultados intermedios también son iguales). Un algoritmo no es determinista si se introduce algo de aleatoriedad en el proceso de encontrar la solución y por lo tanto los resultados finales e intermedios no tienen por qué coincidir.
Al resolver problemas de optimización en respuesta a la "precisión", damos la siguiente clasificación:

- **Algoritmos exactos** son algoritmos que siempre devuelven una solución óptima.

- **Algoritmos aproximados** son algoritmos que producen soluciones que están dentro de un cierto porcentaje del óptimo. Un algoritmo λ -aproximado devuelve una solución x tal que

$$\begin{cases} OPT \leq c(x) \leq \lambda.OPT & \text{si} \quad \lambda > 1 \, (\text{minimización}) \\ \lambda.OPT \leq c(x) \leq OPT & \text{si} \quad \lambda < 1 \, (\text{maximización}) \end{cases} \tag{1.1}$$

[3]En la teoría de computadoras reales e imaginarios, lenguaje y otros sistemas de programación lógica, un sistema Turing-completo es uno que tiene una potencia de cálculo equivalente a la máquina universal de Turing. En otras palabras, el sistema y la máquina universal de Turing pueden emular entre sí.

[4]Una máquina de Turing (TM) es un modelo computacional que realiza una lectura/escritura de manera automática sobre una entrada llamada cinta, generando una salida en esta misma.

- **Algoritmos heurísticos** son algoritmos que producen soluciones sin ninguna garantía de optimalidad y, a su vez, por lo general tienen un tiempo de ejecución mucho menor.

Uno de los inconvenientes de algoritmos exactos es que pueden ser muy lentos. En este caso podemos utilizar algoritmos aproximados o, si son todavía demasiado lentos, los algoritmos heurísticos. La metodología es muy diferente entre los algoritmos heurísticos y algoritmos exactos y aproximados.

Dentro del grupo de los algoritmos heurísticos podemos distinguir los denominados *métodos metaheurísticos*, que imitan fenómenos simples observados en la naturaleza y que parecen estar asociados con la inteligencia artificial. Estos algoritmos tratan de adaptar el comportamiento de diferentes especies a soluciones de problemas altamente complejos mediante optimización. Entre otros, podemos destacar los siguientes:

- Algoritmos evolutivos (genéticos): basado en modelos biológicos que emulan el proceso natural de evolución.
- Algoritmos basados en el comportamiento de las comunidades de hormigas, abejas, etc.
- *Simulated annealing*
- Búsqueda heurística (tabú, aleatorios...)
- Sistemas multiagente

En general, los métodos tradicionales se encargan de buscar y garantizar un óptimo local mientras que los métodos metaheurísticos tienen mecanismos específicos para alcanzar un óptimo global, pero no garantizan ese alcance.

1.4.2. Medición de la complejidad de un algoritmo

A la hora de trabajar con algoritmos, surge de forma natural la necesidad de saber cómo de bueno es un algoritmo dado. Además nos interesa que los algoritmos sean "rápidos" para toda una clase de problemas, no para un ejemplo en particular. A continuación discutiremos algunos enfoques que nos sirven para medir el rendimiento de un algoritmo.

- **Análisis empírico**: consiste en estimar el comportamiento del algoritmo en la práctica, probándolo en varios ejemplos del problema. *Inconvenientes*: El desarrollo del algoritmo depende del lenguaje de programación, compilador, equipo y la habilidad del programador. Llevar a cabo un análisis empírico serio por lo general conlleva mucho tiempo. Es muy difícil comparar los algoritmos a través de pruebas

empíricas, ya que el rendimiento puede depender de los problemas elegidos para las pruebas.

- **Análisis del caso promedio**: consiste en estimar el número medio de pasos que necesita el algoritmo. Por lo general, eligen una distribución de probabilidad sobre los posibles problemas y técnicas estadísticas que se utiliza para calcular los tiempos asintóticos de ejecución del algoritmo.
  Inconvenientes: El análisis depende fundamentalmente de la elección de la distribución de probabilidad. A menudo es difícil determinar la mejor distribución de probabilidad para los problemas de la práctica. El análisis suele ser bastante complejo matemáticamente, lo que hace muy difícil llevar a cabo problemas complicados. Encontramos algoritmos que tienen un rendimiento promedio muy bueno, pero hay muy pocos ejemplos con significación estadística que el algoritmo no es capaz de resolver en un tiempo razonable.

- **Análisis del peor caso**: consiste en encontrar límites superiores para el número de operaciones que requerirá el algoritmo para cualquier problema de la clase en estudio.
  Ventajas: Se trata de un análisis independiente del lenguaje de programación, compiladores, etc. Por lo general, es relativamente fácil de realizar. Se da el tiempo máximo que necesita el algoritmo para resolver un problema dado. Es capaz de comparar dos algoritmos inequívocamente dados.
  Inconvenientes: Podemos clasificar como malo un algoritmo que sólo funciona mal en ejemplos patológicos del problema en estudio.

A continuación definimos las clases de complejidad computacional para distinguir los problemas de acuerdo con su “dificultad”.

Definición 9. *Clase P*
Dado un problema, decimos que pertenece a la ***clase de complejidad P*** *si existe un algoritmo que resuelve cualquier ejemplo de su problema en tiempo polinomial.*

Definición 10. *Clase NP*
Dado un problema, decimos que pertenece a la ***clase de complejidad NP*** *si existe un algoritmo que puede verificar cualquier solución a un ejemplo de esta clase en tiempo polinomial.*

Definición 11. *Clase NP-completa*
Sea P’ un problema en la clase NP. Entonces P’ es ***NP-completo*** *si cualquier problema en clase NP se puede reducir a P’ en tiempo polinomial.*

Definición 12. *Clase NP-duro*
Un problema es ***NP-duro*** *si cualquier problema en 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.

La gran pregunta de la teoría de complejidad computacional es si las dos clases P y NP coinciden. Es decir, si $P = NP$. Si bien existe un amplio consenso de que ambas clases deben ser diferentes, nadie lo ha probado hasta ahora. En el caso de que ambas clases fuesen iguales, esto tendría un gran impacto en muchos campos. Por ejemplo, implicaría que existe un algoritmo polinomial para factorizar números primos, lo que podría comprometer seriamente a muchos protocolos de seguridad.

1.5. Técnicas de resolución. Clasificación

Hoy en día hay una gran cantidad de problemas que resolver para los cuales se han desarrollado algoritmos específicos para su resolución. Dichos algoritmos llevan a cabo una gran cantidad de tareas de cálculos, lo cual sería inimaginable al comienzo del desarrollo de la investigacion operativa. Los algoritmos que proporcionan una solución a un problema próxima a la óptima o una solución para algún caso concreto de dicho problema se les llama *algoritmos heurísticos*. Este grupo incluye una amplia abundancia de métodos basados en técnicas tradicionales, así como específicas. Comenzaremos resumiendo los principios fundamentales de los algoritmos de búsqueda tradicionales.

El algoritmo de búsqueda más simple es el de **búsqueda exhaustiva** que prueba todas las soluciones posibles de un conjunto predeterminado y elige la mejor subsecuencialmente.

Búsqueda local es una versión de búsqueda exhaustiva que sólo se centra en un área limitada del espacio de búsqueda. La búsqueda local se puede organizar de diferentes maneras. La popular técnica de *hill-climbing* pertenece a esta clase de algoritmos. A pesar de existir muchas versiones para estos algoritmos, la técnica es común a todos ellos: reemplazar constantemente la solución actual con el mejor de sus vecinos si éste mejora al actual. El proceso termina si no existe mejora posible, o si expira el tiempo de ejecución. Por ejemplo, la heurística para el problema de la replicación del intragrupo para el servicio de distribución de contenidos multimedia basado en la red Peer-to-Peer se basa en la estrategia de hill-climbing. Una simple iteración de este algoritmo sería la siguiente:

Procedimiento: iteración *hill climbing*
Empezar
$t \leftarrow 0$

inicializar *mejor*
repetir
local ← FALSO
selecciona un punto aleatorio **uactual**
evalúa **uactual**
repetir
seleccionar todos los puntos nuevos en la vecindad de **uactual**
seleccionar el punto **unuevo** del conjunto de nuevos
puntos con el mejor valor de la función objetivo
si *eval*(**unuevo**) es mejor que *eval*(**uactual**)
entonces uactual ← unuevo
si no *local* ← VERDADERO
hasta que *local*
$t \leftarrow t + 1$
si uactual es mejor que *mejor*
entonces *mejor* ← **uactual**
hasta que $t = MAX$
fin

Los algoritmos de la técnica **Divide y Vencerás** tratan de dividir un problema en problemas más pequeños que son más fáciles de resolver. La solución del problema original será una combinación de las soluciones de los problemas pequeños. Esta técnica es eficaz, pero su uso es limitado porque no hay un gran número de problemas que se puedan particionar y combinar fácilmente como tal.

Procedimiento: iteración divide y vencerás (P)
Empezar
Divide el problema P en subproblemas $P_1, P_2, ..., P_k$
Para $i = 1$ **hasta** k **hacer**
si tamaño(P_i) es menor que ρ **entonces** resolver P_i (cogiendo s_i)
si no $s_i \leftarrow$ Divide y Vencerás (P_i)
combinar las soluciones s_i en la solución final.
fin

La técnica ***branch-and-bound*** es una enumeración crítica del espacio de búsqueda. No sólo enumera sino que constantemente trata de descartar las partes del espacio de búsqueda que no pueden contener la mejor solución.

Programación dinámica es una búsqueda exhaustiva que evita la repetición de cálculos mediante el almacenamiento de las soluciones de subproblemas. El punto clave para el uso de esta técnica es formular el proceso de solución como una recursión.

Un método popular para construir sucesivamente el espacio de soluciones es la técnica ***greedy***, que se basa en el principio evidente de tomar la mejor elección (local) en cada etapa del algoritmo con el fin de encontrar el óptimo global de alguna función objetivo.

Las técnicas de *branch-and-bound* y programación dinámica son bastante eficaces, pero su tiempo de complejidad a menudo es demasiado alto e inaceptable para problemas NP-completos.

El algoritmo *hill-climbing* es eficaz, pero tiene un inconveniente importante llamado *convergencia prematura*. Debido a que es un algoritmo "codicioso (*greedy*)", siempre encuentra el óptimo local más cercano de baja calidad. El objetivo de las técnicas heurísticas modernas es superar esta desventaja.

- Algoritmo ***simulated annealing*** (recocido simulado), inventado en 1983, utiliza un enfoque similar al de *hill-climbing*, pero en ocasiones acepta soluciones que son peores que la actual. La probabilidad de dicha aceptación disminuye con el tiempo.

- **Búsqueda tabú** extiende la idea de evitar el óptimo local mediante el uso de estructuras de memoria. El problema del recocido simulado es que después de "saltar", el algoritmo simplemente puede repetir su propia lista. Búsqueda tabú prohibe la repetición de movimientos que se han hecho recientemente.

- **Inteligencia de enjambre** fue introducido en 1989. Es una técnica de inteligencia artificial basado en el estudio del comportamiento colectivo en sistemas descentralizados (auto-organizados). Dos de los tipos de mayor éxito de este enfoque son *optimización de las colonias de hormigas* (ACO) y *optimización de enjambre de partículas* (PSO). En ACO, hormigas artificiales construyen soluciones moviéndose en el gráfico del problema y cambiándolo de tal manera que las futuras hormigas puedan construir mejores soluciones. PSO se encarga de los problemas en los que podemos representar una solución mejor como un punto o una superficie en un espacio n-dimensional. La principal ventaja de las técnicas de inteligencia de enjambre es que son impresionantemente resistentes al problema local de los óptimos.

- Los **algoritmos evolutivos** abordan con éxito la convergencia prematura teniendo en cuenta, al mismo tiempo, una serie de soluciones. Una referencia bastante actual de dichos algoritmos es Haupt et al (2004). A continuación, hablaremos de este grupo de algoritmos con más detalle.

- Las **redes neuronales** se inspiran en los sistemas neuronales biológicos. Se componen de unas unidades llamadas neuronas, y las intercone-

xiones entre ellas. Después de un entrenamiento especial sobre algunos datos de determinado conjunto de redes neuronales se pueden hacer predicciones para casos que no están en el conjunto de entrenamiento. En la práctica, las redes neuronales no siempre funcionan bien porque sufren mucho de problemas de *"bajoajuste"* y *"sobreajuste"*. Estos problemas se correlacionan con la precisión de predicción. Si una red no es lo suficientemente compleja, puede simplificar las leyes a las que obedecen los datos. Desde otro punto de vista, si una red es muy compleja puede tener en cuenta el ruido que normalmente aparece en los conjuntos de datos de entrenamiento, mientras infieren las leyes. La calidad de la predicción después del entrenamiento se deterioró en ambos casos. El problema de la convergencia prematura también es fundamental para las redes neuronales.

- **Support vector machines** (SVMs) amplía las ideas de las redes neuronales. Superan con éxito la convergencia prematura ya que se utiliza una función objetivo convexa, por lo tanto, sólo existe un óptimo. La clásica técnica de "divide y vencerás" ofrece una solución elegante para los problemas separables, que, en relación con SVM proporcionan una clasificación efectiva y se convierten en un instrumento extremadamente poderoso.

Capítulo 2

Técnicas heurísticas de optimización

2.1. Algoritmos exactos y heurísticas

Son muchos los algoritmos que pueden aplicarse en busca de buenos resultados. Tal y como hemos dicho anteriormente, el estudio ha ido evolucionando desde el uso de algoritmos exactos hacia heurísticos y metaheurísticos finalmente.

Pero si queremos responder a preguntas como: ¿Cuántos caminos hay para...? ¿Listar todas las posibles soluciones para...? ¿Hay un camino para...? usualmente requiere de una búsqueda exhaustiva dentro del conjunto de todas las soluciones potenciales. Por eso, los algoritmos que resuelven este tipo de problemas reciben el nombre de **algoritmos exactos** o de **búsqueda exhaustiva**.

Por ejemplo, si se desean encontrar todos los números primos menores de 104, no hay método conocido que no requiera de alguna manera, examinar cada uno de los números enteros entre 1 y 104. De otra manera, si se desea encontrar todos los caminos de un laberinto, se deben examinar todos los caminos iniciando desde la entrada.

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

La segunda técnica es *tamiz o criba*, y es el complemento lógico de *retroceso* en que se tratan de eliminar las no-soluciones en lugar de tratar de

encontrar la solución. El método *tamiz* es útil principalmente en cálculos numéricos teóricos. Se debe tener en mente, sin embargo, que retroceso y tamiz son solamente técnicas generales. Se aplicarán en algoritmos cuyos requerimientos en tiempo son prohibitivos.

En general, la velocidad de los ordenadores no es práctica para una búsqueda exhaustiva de más de 100 elementos. Así, para que estas técnicas sean útiles, deben considerar solamente una estructura dentro de la cual se aproxima el problema. La estructura debe ser hecha a medida, a menudo con gran ingenio, para cuadrar con el problema particular, de modo que el algoritmo resultante será de uso práctico. Los métodos exactos de resolución de problemas se han aplicado con éxito a una cantidad elevada de problemas. Algunos ejemplos de estos métodos son los algoritmos voraces, algoritmos de divide y vencerás, algoritmos de ramificación y poda, *backtraking*, etc.

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

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

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

- Buscar un problema parecido que ya haya sido resuelto.
- Determinar la técnica empleada para su resolución así como la solución obtenida.

- En el caso en el que sea posible, utilizar la técnica y solución descrita en el punto anterior para resolver el problema planteado.

Existen dos interpretaciones posibles para el término heurística. La primera de ellas concibe las heurísticas como un procedimiento para resolver problemas. La segunda interpretación de heurística entiende que éstas son una función que permite evaluar la bondad de un movimiento, estado, elemento o solución.

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

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

    - Estrategia voraz: Partiendo de una semilla, se va construyendo paso a paso una solución factible. En cada paso se añade un elemento constituyente de dicha solución, que se caracteriza por ser el que produce una mejora más elevada en la solución parcial para ese paso concreto. Este tipo de algoritmos se dice que tienen una visión "miope" ya que eligen la mejor opción actual sin que les importe qué ocurrirá en el futuro.
    - Estrategia de descomposición: Se divide sistemáticamente el problema en subproblemas más pequeños. Este proceso se repite (generalmente de forma recursiva) hasta que se tenga un tamaño de problema en el que la solución a dicho subproblema es trivial. Después, el algoritmo combina las soluciones obtenidas hasta que se tenga la solución al problema original. Los algoritmos más representativos de los métodos de descomposición son los algoritmos de divide y vencerás tanto en su versión exacta como aproximada.
    - Métodos de reducción: Identifican características que contienen las soluciones buenas conocidas y se asume que la solución óptima también las tendrá. De esta forma, se puede reducir drásticamente el espacio de búsqueda.
    - Métodos de manipulación del modelo: Consisten en simplificar el modelo del problema original para obtener una solución al problema simplificado. A partir de esta solución aproximada, se extrapola la solución al problema original. Entre estos métodos se pueden destacar: la linealización, la agrupación de variables, introducción de nuevas restricciones, etc.

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

   - Estrategia de búsqueda local 1: Parte de una solución factible y la mejora progresivamente. Para ello examina su vecindad y selecciona el primer movimiento que produce una mejora en la solución actual (*first improvement*).
   - Estrategia de búsqueda local 2: Parte de una solución factible y la mejora progresivamente. Para ello examina su vecindad y todos los posibles movimientos seleccionando el mejor movimiento de ellos, es decir aquél que produzca un incremento (en el caso de maximización) más elevado en la función objetivo (*best improvement*).
   - Estrategia aleatorizada: Para una solución factible dada y una vecindad asociada a esa solución, se seleccionan aleatoriamente soluciones vecinas de esa vecindad.

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

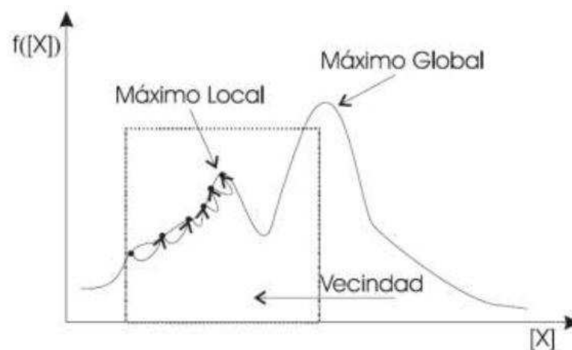

Figura 2.1: Encasillamiento de los algoritmos heurísticos en óptimos locales

Los algoritmos heurísticos no poseen ningún mecanismo que les permita escapar de los óptimos locales. Para solventar este problema se introducen otros algoritmos de búsqueda más inteligentes que eviten en la medida de lo posible quedar atrapados. Estos algoritmos de búsqueda más inteligentes, son los denominados metaheurísticos.

El término metaheurística o meta-heurística fue acuñado por F. Glover en el año 1986. Con este término, pretendía definir un "procedimiento maestro de alto nivel que guía y modifica otras heurísticas para explorar soluciones más allá de la simple optimalidad local". Actualmente, existe una cantidad muy importante de trabajos científicos publicados que abordan problemas de optimización a través de las metaheurísticas, investigaciones sobre nuevas metaheurísticas o extensiones de las metaheurísticas ya conocidas. Existen bastantes foros donde se publican todos estos trabajos de investigación. La idea básica general es siempre la misma: enriquecer a los algoritmos heurísticos de forma que éstos no se queden atrapados en óptimos locales. La evolución de las metaheurísticas durante los últimos 25 años ha tenido un comportamiento prácticamente exponencial. En el tiempo que transcurre desde las primeras reticencias (por su supuesta falta de rigor científico) hasta la actualidad, se han encontrado soluciones de muy alta calidad a problemas que hace tiempo parecían inabordables. De modo general, se puede decir que las metaheurísticas combinan ideas que provienen de cuatro campos de investigación bien distintos:

- Las técnicas de diseño de algoritmos (resuelven una colección de problemas)
- Algoritmos específicos (dependientes del problema que se quiere resolver).
- Fuente de inspiración (del mundo real).
- Métodos estadísticos.

Una primera conclusión que se puede extraer de las definiciones dadas es que, en muchos casos, son reglas de sentido común que permiten hacer una búsqueda inteligente. Debido a esta característica, para bastantes metaheurísticas no existe un marco teórico que las sustente, sino que es a través de los buenos resultados experimentales donde encuentran su justificación. Definir un marco general en el que definir las metaheurísticas resulta un poco complicado hoy en día aunque se esta estudiando la manera de englobarlas a todas. Por el momento se pueden clasificar de la siguiente manera:

- Atendiendo a la inspiración: Natural: algoritmos que se basan en un simil real, ya sea biológico, social, cultural, etc. Sin inspiración: algoritmos que se obtienen directamente de sus propiedades matemáticas.
- Atendiendo al número de soluciones: Poblacionales: buscan el óptimo de un problema a través de un conjunto de soluciones. Trayectoriales: trabajan exclusivamente con una solución que mejoran iterativamente.

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

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

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

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

Según el teorema NFL (No Free Lunch Theorem), que demuestra que al mismo tiempo que una metaheurística es muy eficiente para una colección de problemas, es muy ineficiente para otra colección), los métodos generales de búsqueda, entre los que se encuentran las metaheurísticas, se comportan exactamente igual cuando se promedian sobre todas las funciones objetivo posibles, de tal forma que si un algoritmo A es más eficiente que un algoritmo B en un conjunto de problemas, debe existir otro conjunto de problemas de igual tamaño para los que el algoritmo B sea más eficiente que el A. Esta aseveración establece que, en media, ninguna metaheurística (algoritmos genéticos, búsqueda dispersa, búsqueda tabú, etc.) es mejor que la búsqueda completamente aleatoria. Una segunda característica que presentan las metaheurísticas es que existen pocas pruebas sobre su convergencia hacia un óptimo global; es decir, que a priori no se puede asegurar ni que la metaheurística converja ni la calidad de la solución obtenida. Por último, las metaheurísticas más optimizadas son demasiado dependientes del problema o al menos necesitan tener un elevado conocimiento heurístico del problema. Esto hace que, en general, se pierda la genericidad[1] original con la que fueron concebidas.

A pesar de estos aparentes problemas, la realidad es que el comportamiento experimental de la mayoría de las metaheurísticas es extraordinario, convirtiéndose para muchos problemas difíciles de resolver en la única alternativa factible para encontrar una solución de calidad en un tiempo razonable. En general, las metaheurísticas se comportan como métodos muy

[1] El término genericidad se refiere a una serie de técnicas que permitan escribir algoritmos o definir contenedores de forma que puedan aplicarse a un amplio rango de tipos de datos

robustos y eficientes que se pueden aplicar con relativa facilidad a una colección amplia de problemas. Además, la demostración del teorema NFL se basa en que el algoritmo de búsqueda no visita dos veces la misma solución y en que no se introduce conocimiento heurístico en el diseño del método metaheurístico. Estas hipótesis habitualmente no son ciertas.

A continuación definiremos los siguientes métodos de resolución de problemas de optimización:

- Búsqueda Exhaustiva
- Búsqueda Local
- Búsqueda Tabú
- *Simulated Annealing*
- Algoritmos Genéticos
- Colonias de Hormigas

2.2. Búsqueda Exhaustiva

Tal y como su nombre indica, la búsqueda exhaustiva consiste en evaluar cada una de las soluciones del espacio de búsqueda hasta encontrar la mejor solución global. Esto significa que si no conoces el valor correspondiente a la mejor solución global, no hay forma de asegurarse de si has encontrado la mejor solución utilizando esta técnica, a menos que examines todas las posibles. Como el tamaño del espacio de búsqueda de problemas reales es usualmente grande, es probable que se requiera mucho tiempo computacional para probar cada una de las soluciones.

Los algoritmos exhaustivos (o enumerativos) son interesantes en algunos casos por su simplicidad; el único requisito es generar sistemáticamente cada posible solución al problema. Además, existen métodos para reducir esta cantidad de operaciones, como por ejemplo la técnica de *backtracking.* Algunos algoritmos clásicos de optimización que construyen la solución completa de soluciones parciales (por ejemplo *branch and bound*) están basados en búsqueda exhaustiva.

2.2.1. Aplicación al ejemplo de referencia

A continuación mostraremos cómo funciona dicho algoritmo sobre un ejemplo en concreto, de elaboración propia.

Ejemplo 1 (Ejemplo de referencia)**.** *El ayuntamiento de una ciudad desea contruir tres parques de ocio infantil. Para ello dispone de cinco fincas ubicadas en distintos puntos urbanísticos de los cuales deberá escoger tres*

que minimicen el coste de compra y construcción. Además como las fincas 4 *y* 5 *están relativamente próximas, el ayuntamiento considera que no se deben construír simultáneamente parques en ambas fincas. Los costes de compra y contrucción de cada finca se detallan en la siguiente tabla:*

	Finca 1	*Finca* 2	*Finca* 3	*Finca* 4	*Finca* 5
Coste (Miles de euros)	2	2.4	3	4	4.4

Definamos las variables $x_i = 1$ *si se construye un parque en la finca* i *y* $x_i = 0$ *si no se construye en la finca* i, $i = 1, \ldots, 5$. *De esta forma el problema de optimización a resolver sería el siguiente:*

$$\begin{array}{ll} \textit{minimizar} & c(x) = 2x_1 + 2.4x_2 + 3x_3 + 4x_4 + 4.4x_5 \\ \textit{sujeto a} & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & x_4 + x_5 \leq 1 \\ & x_i \in \{0, 1\}, i = 1, \ldots, 5. \end{array}$$

Para tener en cuenta las restricciones del problema a la hora de buscar el mínimo de la función objetivo lo que haremos será penalizar en la función objetivo la violación de dichas restricciones. En concreto, penalizaremos con 18 *unidades la violación de la primera restricción y con* 8 *unidades si no se verifica la segunda. De esta forma el algoritmo se moverá intentando encontrar aquellas soluciones que verifiquen las restricciones, puesto que tendrán menos valor en la función objetivo.*

- Solución

Consideremos el espacio de las posibles soluciones S.
Notemos que $S = \{(x_1, \ldots, x_5)$ tal que $x_i \in \{0, 1\}$, $i = 1, \ldots, 5\}$. Entonces:
$S = \{(1,1,1,1,1), (1,1,1,1,0), (1,1,1,0,1), (1,1,1,0,0), (1,1,0,1,1),$
$(1,1,0,1,0), (1,1,0,0,1), (1,1,0,0,0), (1,0,1,1,1), (1,0,1,1,0), (1,0,1,0,1),$
$(1,0,1,0,0), (1,0,0,1,1), (1,0,0,1,1), (1,0,0,1,0), (1,0,0,0,1), (1,0,0,0,0),$
$(0,1,1,1,1), (0,1,1,1,0), (0,1,1,0,1), (0,1,1,0,0), (0,1,0,1,1), (0,1,0,1,0),$
$(0,1,0,0,1), (0,1,0,0,0), (0,0,1,1,1), (0,0,1,1,0), (0,0,1,0,1), (0,0,1,0,0),$
$(0,0,0,1,1), (0,0,0,1,1), (0,0,0,1,0), (0,0,0,0,1), (0,0,0,0,0)\}$

Evaluando cada posible solución en la función de evaluación (función objetivo) construimos el Cuadro 2.1.

Por tanto, la solución óptima del problema es $x = (1,1,1,0,0)$ con función de evaluación $c(x) = 7.4$

2.3. Búsqueda Local

2.3.1. Introducción

La búsqueda local es un método metaheurístico para resolver problemas de optimización computacionalmente complejos. Los algoritmos de búsqueda

$c(1,1,1,1,1) = 41.8$	$c(1,1,1,1,0) = 29.4$	$c(1,1,1,0,1) = 29.8$
$c(1,1,1,0,0) = 7.4$	$c(1,1,0,1,1) = 38.8$	$c(1,1,0,1,0) = 8.4$
$c(1,1,0,0,1) = 8.8$	$c(1,1,0,0,0) = 22.4$	$c(1,0,1,1,1) = 39.4$
$c(1,0,1,1,0) = 9$	$c(1,0,1,0,1) = 9.4$	$c(1,0,1,0,0) = 23$
$c(1,0,0,1,1) = 18.4$	$c(1,0,0,1,0) = 24$	$c(1,0,0,0,1) = 24.4$
$c(1,0,0,0,0) = 20$	$c(0,0,0,0,0) = 18$	$c(0,1,1,1,1) = 39.8$
$c(0,1,1,1,0) = 9.4$	$c(0,1,1,0,1) = 9.8$	$c(0,1,1,0,0) = 23.4$
$c(0,1,0,1,1) = 18.8$	$c(0,1,0,1,0) = 24.4$	$c(0,1,0,0,1) = 24.8$
$c(0,1,0,0,0) = 20.4$	$c(0,0,1,1,1) = 19.4$	$c(0,0,1,1,0) = 25$
$c(0,0,1,0,1) = 25.4$	$c(0,0,1,0,0) = 21$	$c(0,0,0,1,1) = 34.4$
$c(0,0,0,1,0) = 22$	$c(0,0,0,0,1) = 22.4$	

Cuadro 2.1: Valores de la función objetivo en el espacio de soluciones

local utilizan problemas que pretenden encontrar una solución óptima entre un número de soluciones candidatas. Estos algoritmos se mueven de una solución a otra en el espacio de las posibles soluciones (el espacio de búsqueda) mediante la aplicación de cambios locales, hasta que se encuentra la solución considerada óptima o transcurrido un plazo establecido.

La mayoría de los problemas pueden ser formulados en términos de espacio de búsqueda y de destino de varias maneras diferentes. Por ejemplo, para el problema del viajante una solución puede ser un ciclo y el criterio de maximizar puede ser una combinación del número de nodos y la longitud del ciclo. Pero una solución también puede ser un camino, siendo el ciclo el objetivo.

El algoritmo de búsqueda local comienza a partir de una solución candidata, moviéndose, iterativamente, a una solución vecina. Esto sólo es posible si se define una relación de vecindad en el espacio de búsqueda. Por ejemplo, el vecino de una solución dada por un conjunto de vértices podría ser otra solución que difiere de la anterior en un nodo. Esto es, si tenemos una solución dada por el conjunto $(1-3-4-5)$, un posible vecino sería el conjunto $(1-2-4-5)$. El mismo problema puede tener múltiples vecinos definidos en él. La optimización local con vecinos que implican cambios hasta k componentes de la solución se refiere a menudo como k-opt.

Por lo general, cada solución candidata tiene más de una solución vecina, la elección de cada una se hace tomando sólo la información de las soluciones vecinas de la actual, de ahí el nombre de búsqueda local. Cuando la elección de la solución vecina se hace tomando el criterio de maximización local, la metaheurística toma el nombre de *hill climbing*. En el caso de que no haya configuraciones de mejora presentes en los vecinos, la búsqueda local se queda atascada en un punto óptimo a nivel local. Este problema de óptimos locales se puede arreglar mediante el reinicio (búsqueda local repetida con diferentes condiciones iniciales), o sistemas más complejos basados en: iteraciones, como la búsqueda local iterada; en memoria, como

la optimización de la búsqueda reactiva; simulación de fenómenos físicos, como recocido simulado.

La convergencia de la búsqueda local puede llevarse a cabo en un tiempo limitado. Una opción común es terminar cuando no se mejore en un número dado de pasos la mejor solución encontrada por el algoritmo. La búsqueda local es un algoritmo que puede devolver una solución válida incluso si se interrumpe en cualquier momento antes de que termine. Los algoritmos de búsqueda local son algoritmos normalmente de aproximación o incompletos, pues la búsqueda se puede detener aunque la mejor solución encontrada por el algoritmo no sea la óptima. Esto puede suceder incluso si la convergencia es debida a la imposibilidad de mejorar la solución, como en el caso de que la solución óptima esté lejos de la zona de las soluciones consideradas por el algoritmo.

Para problemas específicos, es posible idear vecinos muy grandes, posiblemente de tamaño exponencial. Si la mejor solución dentro de la zona se puede encontrar de manera eficiente, tales algoritmos se denominan, a grande escala, *algoritmos de búsqueda vecinales*.

2.3.2. Descripción del algoritmo

Un algoritmo genérico de búsqueda local puede describirse por un conjunto $\mathcal{S}$ de todas las soluciones factibles, una función de coste *cost*: $\mathcal{S} \rightarrow \mathbb{R}$, una estructura de *vecindad* $B : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ y un oráculo que, dada una solución S, encuentra (si es posible) una solución $S' \in B(S)$ tal que $cost(S') < cost(S)$.

Diremos que una solución $S \in \mathcal{S}$ es *óptima localmente* si $cost(S) \leq cost(S')$ para todo $S' \in B(S)$. Los algoritmos de búsqueda local siempre devuelven una solución de este tipo. Notemos que si S es una solución óptima local, entonces para todo $S' \in B(S)$, $cost(S') - cost(S) \geq 0$. La función de coste *cost* y la estructura de vecindad B será diferente para cada problema y algoritmo en cuestión.

En un problema de minimización la idea básica de los algoritmos de búsqueda local es la siguiente:

(I) Iniciamos el proceso con una solución inicial x que, por ejemplo, podría ser elegida aleatoriamente. Otra posibilidad sería diseñar un algoritmo específico para el problema en cuestión que proporciona una solución inicial razonable.

(II) Calcular un vecino y de la solución actual de forma que se mejore la solución actual; es decir, $cost(y) < cost(x)$ (Si el problema es de maximizar basta cambiar la desigualdad). Esto se consigue aplicando a la solución actual alguna transformación de algún conjunto de tranformaciones dado, que serán las que establezcan la forma de calcular el vecino.

La solución mejorada se convierte en la solución actual; es decir, reemplazamos x por y.

(III) Repetir lo anterior hasta que ninguna transformación del conjunto mejore la solución actual o hasta que se rezalice un número máximo de iteraciones sin encontrar una solución mejor.

A pesar de su sencillez, hay varios elementos clave a tener en cuenta a la hora de diseñar el algoritmo:

- Elección adecuada de los entornos: esto es crucial para el funcionamiento del algoritmo. Por un lado interesa que los entornos sean relativamente pequeños para que el proceso de búsqueda de vecinos sea rápido, pero por otro lado también queremos que sean lo suficientemente grandes para que el algoritmo no quede estancado en óptimos locales.

- Solución inicial: el resultado final del algoritmo puede depender crucialemente de la solución inicial escogida. Para mitigar la dependencia de la solución inicial se suele ejecutar el algoritmo múltiples veces con diferentes soluciones iniciales.

- Criterio de mejora: en el paso (II) hay dos formas principales de elegir la nueva solución y. Se puede elegir la primera solución y que está en un entorno de x y mejora a x. Otra opción sería seleccionar de entre todas las soluciones que están en el entorno de x la mejor. Con la primera opción el algoritmo realiza más rápido cada iteración, pero la segunda permite dar pasos más grandes en cada iteración.

2.3.3. Aplicación al ejemplo de referencia

A continuación mostraremos cómo funciona dicho algoritmo sobre el Ejemplo 1.

Tomamos como solución inicial $x = (1, 1, 0, 1, 1)$, cuyo valor de la función objetivo es $c(x) = 2 + 2.4 + 4 + 4.4 + 18 + 8 = 38.8$. Notemos que se están violando las dos restricciones, por lo que hemos añadido ambas penalizaciones en la función objetivo. Para evitar no incumplir la primera restricción y así conseguir que la función objetivo disminuya, lo que haremos será calcular un vecino x' de forma que sólo contenga 3 unos, por ejemplo $x' = (1, 0, 0, 1, 1)$ con $c(x) = 18.4$. Como hemos mejorado la función objetivo, actualizamos $x = x'$. El proceso continua calculando vecinos de x y seleccionando aquéllos que mejoran la función objetivo:

Vecinos de x	Valor función objetivo
$(1,0,1,0,1)$	9.4
$(1,1,0,0,1)$	8.8
$(1,1,0,1,0)$	8.4
$(1,0,1,1,0)$	9
$(0,1,0,1,1)$	18.8
$(0,0,1,1,1)$	19.4

Para calcular los vecinos lo que hemos hecho ha sido simplemente intercambiar un 0 por un 1 en el vector x. De los vecinos que podemos observar en la tabla vemos que el mejor es $x' = (1,1,0,1,0)$, porque tiene el menor valor de la función objetivo $c(x) = 8.4$. Por lo tanto reemplazamos $x = x' = (1,1,0,1,0)$ y calculamos los vecinos de esta nueva solución para ver si hay alguno mejor.

Vecinos de x	Valor función objetivo
$(1,1,1,0,0)$	7.4
$(1,1,0,0,1)$	8.8
$(1,0,1,1,0)$	9
$(1,0,0,1,1)$	18.4
$(0,1,1,1,0)$	9.4
$(0,1,0,1,1)$	18.8

De los vecinos que podemos observar en la tabla vemos que el mejor es $x = (1,1,1,0,0)$ con $c(x) = 7.4$. La idea del algoritmo es volver a calcular vecinos de esta solución y ver si hay alguno mejor. Se puede comprobar fácilmente que $x = (1,1,1,0,0)$ ya es el óptimo global del problema, por lo tanto no se podría encontrar otro vecino que mejore la solución y no tiene sentido continuar ejecutando el algoritmo.

2.4. Búsqueda Tabú

2.4.1. Introducción

La búsqueda tabú, a diferencia de las meta-heurísticas analizadas anteriormente, incorpora la memoria de las soluciones que han llevado a la última solución antes de decidir cuál es la siguiente. Está basada en principios generales de la Inteligencia Artificial y fue desarrollada independientemente por Glover (1986) y Hansen (1986).

Con el fin de introducir esta meta-heurística y sus elementos más importantes, se estudia un ejemplo de optimización combinatoria que es resuelto con un algoritmo de búsqueda tabú. El problema a resolver es el siguiente: Dada la función

$$f : X \to \mathbb{R}$$

determinar

$$s^* \in X \text{ tal que } f(s^*) = \text{mín} \, f(s) \text{ para } s \in X$$

Movimiento

El esquema general de la búsqueda tabú es el siguiente: a partir de una solución inicial, se determina una próxima o vecina que la mejore hasta llegar a una solución aceptable. Se denota por $N(s) \subset X$ al conjunto de vecinos de la solución $s \in X$ del problema de optimización.
En la búsqueda tabú conviene hablar más que de vecinos, de una solución del conjunto de movimientos que permite construir una solución a partir de otra.

Definición 13. *Dada una solución $s \in X$, se define un movimiento como un elemento $m \in M(s)$ de forma que permite construir una solución vecina:*

$$s^{'} = s \oplus m \in N(s) \ \forall m \in M(s)$$

Definición 14. *Dado un movimiento $m \in M(s)$, se define su valor, y se denota por v(m) a la mejora que se produce en la valoración de la solución vecina obtenida por dicho movimiento:*

$$f(s^{'}) = f(s \oplus m) = f(s) + v(m) \ \forall m \in M(s), \forall s \in X$$

Movimiento tabú

En un proceso de búsqueda constituido por una sucesión de movimientos, parece razonable exigir que no se repitan dos movimientos en un período suficientemente corto. Para ello hay que registrar los movimientos que han modificado las soluciones anteriores.

De ahí el término de memoria a corto plazo que identifica esta meta-heurística que prohibe determinados movimientos utilizados en las últimas iteraciones. La memoria de los últimos movimientos trata de evitar repeticiones sistemáticas en el recorrido por el espacio de soluciones para diversificar así el esfuerzo de búsqueda.

Se introduce entonces el siguiente concepto que da nombre a la meta-heurística.

Definición 15. *Dado un movimiento $m \in M(s)$, se denomina tabú, y se denota por $m \in T(s)$, cuando no interese utilizarlo en las próximas iteraciones.*
Se denomina duración al número de iteraciones que un movimiento tabú no puede ser utilizado.

Función de aspiración

Con el fin de permitir movimientos prohibidos para construir soluciones mejores, se introduce el siguiente concepto.

Definición 16. *Un movimiento tabú $m \in T(s)$, puede ser aceptado si verifica ciertas restricciones impuestas por la función de aspiración, que depende de la valoración de la solución actual $f(s)$ y se denota por $A(f(s))$:*

$$f(s \oplus m) < A(f(s))$$

Ejemplos de funciones de aspiración son las siguientes:

- $A(z) = z$
  Con esta función de aspiración se aceptan movimientos tabúes cuando la solución obtenida por el movimiento sea mejor que la anterior.

- $A(z) = f(s^*)$
  Con esta función de aspiración sólo se aceptan aquellos movimientos tabúes que mejoren la mejor solución obtenida hasta el momento $s^* \in X$.

Definición 17. *Se denota frecuencia de un movimiento m al número de veces que se ha aplicado hasta la iteración actual.*

Con el fin de utilizar la frecuencia, se amplía la matriz de datos tabúes incorporando en su triángulo inferior la frecuencia absoluta de los movimientos hasta la iteración actual. Se incorpora así en la búsqueda tabú una estrategia de diversificación.

Hay varias formas de penalizar aquellos movimientos que han sido utilizados más a menudo en las iteraciones anteriores. Se denotará por $p(m)$ la penalización de cada movimiento $m \in M(s)$.

La estrategia de *diversificación* considera la evolución del proceso de búsqueda con el fin de evitar aquellos movimientos más utilizados. De forma análoga, se puede introducir una estrategia de *intensificación* que favorezca la elección de movimientos con mejores valores.

2.4.2. Descripción del algoritmo

Como podemos ver en Aarts et al (2003), la búsqueda tabú puede considerarse como un caso particular de un algoritmo de búsqueda de óptimo para un problema de optimización combinatoria:

$$\min f(x), x \in X$$

Presentemos el esquema general de un algoritmo de búsqueda:

Las condiciones de parada pueden ser:

1. Solución óptima encontrada,

2. $N(s, k+1) = \emptyset$, siendo k el número de iteración.

Inicialización:
$s \in X$
$s^* = s$
$k = 0$
Proceso:
do
$\quad k = k + 1$
$\quad$ Generar $V^* \subset N(s,k)$ (subconjunto de soluciones vecinas)
$\quad$ Elegir la mejor de ellas $s^{'} \in V^*$
$\quad s = s^{'}$
$\quad$ if $(f(s^{'}) < f(s^*))$
$\quad\quad s^* = s^{'}$
$\quad$ end if
while (Condición de parada)

Cuadro 2.2: Algoritmo de búsqueda

3. $k \geq K$, siendo K una cota prefijada, entre otros.

El esquema general de un algoritmo de búsqueda tabú generaliza el algoritmo anterior incorporando las características introducidas previamente: la memoria de corto plazo a través de los movimientos, la posibilidad de relajar la prohibición de los movimientos tabúes con la función de aspiración, etc. Concretamente, se tienen las siguientes características:

1. La definición del conjunto $N(s,k)$ identifica el proceso de búsqueda tabú:

$$N(s,k) = N(s) \setminus T$$

donde T representa el conjunto de las últimas soluciones obtenidas y a las que se llega con un movimiento tabú.

2. Formulación de los movimientos:
En lugar de trabajar con el conjunto de vecinos de una solución concreta s, es más eficaz trabajar con movimientos válidos, de forma que una iteración se representa por:

$$s^{'} = s \oplus m,\ m \in M(s)$$

y el conjunto de vecinos se define según:

$$N(s) = \left\{ s^{'} \in X : \exists m \in M(s) \text{ y } s^{'} = s \oplus m \right\}.$$

El operador $\oplus$ permite relajar la prohibición de los movimientos tabúes.

3. Función de aspiración:
   Para poder aceptar movimientos tabúes cuando, por ejemplo, permitan alcanzar valores de la función objetivo mejores que los obtenidos hasta el momento, se introducen las condiciones de nivel de aspiración:

   Un movimiento tabú m es aceptado si su nivel de aspiración respecto de la solución actual s, $a(s,m)$, es mejor que un determinado umbral fijado $A(s,m)$.

   Por ejemplo, $A(s,m) = f(s)$ permitirá movimientos tabúes cuando mejoren la solución actual s. $A(s,m) = f(s^*)$ permitirá movimientos tabúes sólo si se mejora la mejor solución obtenida hasta el momento s^*.

   La actualización de la función de aspiración permite flexibilizar la búsqueda variando el nivel de prohibición según evoluciona el algoritmo. Una función de aspiración $A(s,m) = -\infty$ no relajaría nunca la prohibición de los movimientos tabúes.

El esquema general de búsqueda tabú puede verse en el Cuadro 2.3.

```
Inicialización:
s ∈ X
s* = s
k = 0
T = ∅ A(z) = z Proceso:
do
    M(s,k) = {m ∈ M(s) : m ∉ T y f(s ⊕ m) < A(f(s))} (Movimientos válidos)
    V' = {s' ∈ X : s' = s ⊕ m, m ∈ M(s,k)} (Conjunto de vecinos)
    Elegir la mejor solución s' ∈ V*
    if (f(s') < f(s*))
        s* = s'
    end if
    Actualizar T y A(·)
    k = k + 1
    s = s'
while (Condición de parada)
```

Cuadro 2.3: Pseudocódigo algoritmo tabú

2.4.3. Aplicación al ejemplo de referencia

Consideremos el Ejemplo 1 y apliquemos el método tabú para calcular la solución óptima. Pararemos en cuatro iteraciones y fijaremos el tamaño de la lista tabú en 3, de modo que cuando se llene la lista, el primer movimiento que entró será el primero en salir:

- Iteración 1

    - Paso 1
    Consideramos la solución inicial: $x^0 = (1,0,1,0,1)$ con función objetivo $c(x) = 9.4$
    - Paso 2
    Presentamos los posibles movimientos:
    $m_1(x^0) : x_1 = 0 \Rightarrow x = (0,0,1,0,1)$ y $c(x) = 25.4$
    $m_2(x^0) : x_2 = 1 \Rightarrow x = (1,1,1,0,1)$ y $c(x) = 29.8$
    $m_3(x^0) : x_3 = 0 \Rightarrow x = (1,0,0,0,1)$ y $c(x) = 24.4$
    $m_4(x^0) : x_4 = 1 \Rightarrow x = (1,0,1,1,1)$ y $c(x) = 39.8$
    $m_5(x^0) : x_5 = 0 \Rightarrow x = (1,0,1,0,0)$ y $c(x) = 23$
    - Paso 3
    El mejor vecino es $x^1 = (1,0,1,0,0)$ con valor $c(x^1) = 23$

        x_1 no es tabú

        La lista tabú es: $L = \{x_5 = 0\}$

- Iteración 2

    - Paso 1
    Consideramos la solución actual: $x^1 = (1,0,1,0,0)$ con función objetivo $c(x) = 23$
    - Paso 2
    Presentamos los posibles movimientos:
    $m_1(x^1) : x_1 = 0 \Rightarrow x = (0,0,1,0,0)$ y $c(x) = 21$
    $m_2(x^1) : x_2 = 1 \Rightarrow x = (1,1,1,0,0)$ y $c(x) = 7.4$
    $m_3(x^1) : x_3 = 0 \Rightarrow x = (1,0,0,0,0)$ y $c(x) = 20$
    $m_4(x^1) : x_4 = 1 \Rightarrow x = (1,0,1,1,0)$ y $c(x) = 9$
    $m_5(x^1) : x_5 = 1 \Rightarrow x = (1,0,1,0,1)$ y $c(x) = 9.4$
    - Paso 3
    El mejor vecino es $x^2 = (1,1,1,0,0)$ con valor $c(x^2) = 7.4$

        x_2 no es tabú

        La lista tabú es: $L = \{x_5 = 0, x_2 = 1\}$

- Iteración 3

  - Paso 1
    Consideramos la solución inicial: $x^2 = (1,1,1,0,0)$ con función objetivo $c(x) = 7.4$
  - Paso 2
    Presentamos los posibles movimientos:
    $m_1(x^2) : x_1 = 0 \Rightarrow x = (0,1,1,0,0)$ y $c(x) = 23.4$
    $m_2(x^2) : x_2 = 0 \Rightarrow x = (1,0,1,0,0)$ y $c(x) = 23$
    $m_3(x^2) : x_3 = 0 \Rightarrow x = (1,1,0,0,0)$ y $c(x) = 22.4$
    $m_4(x^2) : x_4 = 1 \Rightarrow x = (1,1,1,1,0)$ y $c(x) = 29.4$
    $m_5(x^2) : x_5 = 1 \Rightarrow x = (1,1,1,0,1)$ y $c(x) = 29.8$

  - Paso 3
    El mejor vecino es $x^3 = (1,1,0,0,0)$ con valor $c(x^3) = 22.4$

    x_3 no es tabú

    La lista tabú es: $L = \{x_5 = 0, x_2 = 1, x_3 = 0\}$

- Iteración 4

  - Paso 1
    Consideramos la solución inicial: $x^3 = (1,1,0,0,0)$ con función objetivo $c(x) = 22.4$
  - Paso 2
    Presentamos los posibles movimientos:
    $m_1(x^3) : x_1 = 0 \Rightarrow x = (0,1,0,0,0)$ y $c(x) = 20.4$
    $m_2(x^3) : x_2 = 0 \Rightarrow x = (1,0,0,0,0)$ y $c(x) = 20$
    $m_3(x^3) : x_3 = 1 \Rightarrow x = (1,1,1,0,0)$ y $c(x) = 7.4$
    $m_4(x^3) : x_4 = 1 \Rightarrow x = (1,1,0,1,0)$ y $c(x) = 8.4$
    $m_5(x^3) : x_5 = 1 \Rightarrow x = (1,1,0,0,1)$ y $c(x) = 8.8$

  - Paso 3
    El mejor vecino es $x^4 = (1,1,1,0,0)$ con valor $c(x^4) = 7.4$

    x_4 no es tabú

    La lista tabú es: $L = \{x_2 = 1, x_3 = 0, x_3 = 1\}$

Por lo tanto la solución óptima es $x = (1,1,1,0,0)$ con valor de la función objetivo $c(x) = 7.4$

2.5. Simulated Annealing

2.5.1. Introducción

El algoritmo de recocido simulado, *simulated annealing* en inglés, se basa en principios de la termodinámica y el proceso de recocido del acero. Considerando los problemas de optimización combinatoria como problemas que buscan un óptimo global, se pueden incluir procedimientos de búsqueda estocástica, como es éste, como alternativas heurísticas.

Explicación física del método

El nombre de "recocido simulado" se justifica por el templado o enfriado controlado con el que se producen determinadas sustancias; es el caso, por ejemplo, de la cristalización del vidrio o el recocido del acero.

Inicialmente, a temperaturas muy elevadas se produce un amalgama líquido en el que las partículas se configuran aleatoriamente. El estado sólido se caracteriza por tener una configuración concreta de mínima energía (el mínimo global). Para alcanzar esa configuración es necesario enfriar el amalgama lentamente ya que un enfriamiento brusco paralizaría el proceso y se llegaría a una configuración distinta de la buscada (un mínimo local distinto del mínimo global).

Las diferentes configuraciones que se pueden obtener corresponden con las distintas soluciones en el problema de optimización combinatoria, y el óptimo es el mínimo global. Podemos ver el templado simulado como una variación de la simulación de Monte Carlo, cuyo estudio se basa en la simulación del comportamiento de una colección de átomos a una cierta temperatura. En cada iteración, cada átomo es sometido a un desplazamiento aleatorio que provoca un cambio global en la energía del sistema (δ). Si $\delta < 0$, se acepta el cambio; en caso contrario, el cambio se acepta con probabilidad $exp(-\delta/K_BT)$, siendo K_B la denominada constante de Boltzman y T la temperatura absoluta. Para un número grande de iteraciones el sistema alcanza el equilibrio en cada temperatura, y la distribución de probabilidad del sistema sigue la distribución de Boltzman:

$$Prob\{X_T = i\} = \frac{1}{Z(T)} exp(\frac{-E_i}{K_BT})$$

siendo E_i la energía del estado i y $Z(T) = \sum_i exp(\frac{-E_i}{K_BT})$ la constante de normalización. A la función $exp(\frac{-E_i}{K_BT})$ se la denomina función de aceptación y asegura el que el sistema converja a la distribución de Boltzman.

2.5.2. Descripción del algoritmo

Partiendo del esquema general de un algoritmo de minimización local, el recocido simulado introduce una variable de control T denominada temperatura, que permite avanzar en las iteraciones empeorando la función objetivo con cierta probabilidad para así poder salir, si es el caso, de un bucle donde la función objetivo haya podido quedar atrapada en un un mínimo local. El esquema de minimización general es el siguiente:

Sea $x \in S$
Repetir
{
    Genera $y \in V(x) \subset S$
    Evalúa $\delta = C(y) - C(x)$
    Si $\delta < 0$, $x = y$
}
Hasta que $C(y) \geq C(x)$, $\forall y \in V(x)$.

Un problema siguiendo este esquema depende fuertemente de la solución inicial elegida. En la figura 2.2 se observa como la solución final x^* podría haber quedado atrapada en el mínimo local y no podría ir nunca al mínimo global x^{**}.

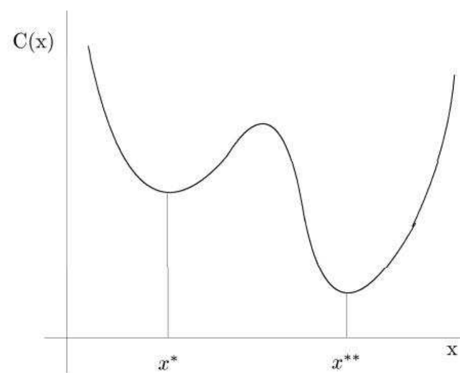

Figura 2.2: Dependencia de la solución inicial

Introduciendo la variable de control T evitaríamos con cierta probabilidad quedar atrapados en el mínimo local. Esta probabilidad se denomina función de aceptación que suele formularse como $exp(-\delta/T)$, siendo δ el incremento o empeoramiento de la función objetivo. La probabilidad de

aceptar incrementos del coste es mayor para incrementos pequeños que grandes, y, a igual incremento, la probabilidad de aceptación es mayor para valores de T altos. En el caso límite de $T = \infty$ aceptamos cualquier empeoramiento del coste; y para el caso de $T = 0$ no se acepta ningún incremento de la función objetivo y se tendría el esquema normal de minimización local.

La probabilidad se introduce a partir de números aleatorios $u \in \mathbb{U}(0, 1)$. El esquema básico del recocido simulado sería el que resulta de incluir esta modificación (haciendo tender a cero el parámetro positivo T, es decir, "enfriando" o "templando" a lo largo del proceso) en el algoritmo anterior.

Algoritmo de recocido simulado
Sea $x \in S$ la configuración inicial.
Sea $T > 0$ la temperatura inicial.
Sea $N(T)$ el número máximo de iteraciones.
Repetir
{
    Repetir
    {
    Genera solución $y \in V(x) \subset S$
    Evalúa $\delta = C(y) - C(x)$
    o $\begin{cases} \delta < 0 \Rightarrow x = y \\ \delta \geq 0 \text{ y } u < exp(-\delta/T) \Rightarrow x = y \end{cases}$
    $n = n + 1$
    } mientras $n \leq N(T)$.
Disminuir T
} mientras no se haya alcanzado el criterio de parada.

2.5.3. Aplicación al ejemplo de referencia

Consideremos el Ejemplo 1 y apliquemos el algoritmo descrito para calcular la solución óptima. En primer lugar lo adaptaremos al esquema descrito anteriormente guiándonos por los siguientes aspectos:

Adaptación del problema:

- Conjunto S de configuraciones: serán los vectores $x = (x_1, \ldots, x_5)$ tal que $x_i \in \{0, 1\}$, $i = 1, \ldots, 5$.

- Función de coste: $c(x) = 2x_1 + 2.4x_2 + 3x_3 + 4x_4 + 4.4x_5 + 18y_1 + 8y_2$, siendo $y_1 = 1$, si la solución x no verifica la primera restricción y 0 en caso contrario. De forma análoga se define y_2 pero teniendo en cuenta si se verifica la segunda restricción o no.

- Vecindad de cada configuración: Definiremos un vecino cambiando

la posible localización de un parque de una finca a otra, es decir, intercambiando 0 por 1 en el vector solución.

- Configuración inicial: empezaremos con una solución inicial que sea razonable, por ejemplo, $x = (1, 0, 0, 1, 1)$.

Parámetros del algoritmo:

- Sabemos que el peor empeoramiento que se puede producir es que no se verifique ninguna restricción; es decir, que se construyan más de 3 parques y que se contruyan parques en las fincas 4 y 5 simultáneamente. En este caso dicho empeoramiento sería $\delta = 18+8 = 26$. A partir de éste, calculamos el valor inicial de la variable de control T (temperatura inicial) de forma que sea lo suficientemente grande para inicialmente movernos por todo el conjunto de la región factible. De esta forma queremos que:

$$exp(-\frac{\delta}{T}) = 0.99 \Leftrightarrow T = -\frac{\delta}{log(0.99)} = 2586.98.$$

- Como factor de disminución de la temperatura tomamos $\alpha = 0.3$, e iremos actualizando la temperatura de la siguiente forma:

$$T_{new} = \alpha T_{old} \, (0 < \alpha < 1).$$

- Actualizaremos la temperatura cada $N(T) = 1$ iteración.

- Criterio de parada: no mejora de la función objetivo en 2 iteraciones consecutivas.

Iteracion 1:
Comenzamos con la solución inicial $x = (1, 0, 0, 1, 1)$, $c(x) = 18.4$. Calculamos un vecino $y = (0, 1, 0, 1, 1)$, para el que obtenemos el valor $c(y) = 18.8$. De esta forma como $\delta = c(y) - c(x) = 0.4 > 0$, quiere decir que no estamos mejorando la función objetivo; entonces generamos un número aleatorio u de una distribución uniforme $(0, 1)$ y obtenemos $u = 0.6$. Como $u = 0.6 < exp(-\frac{\delta}{T}) = 0.99$ aceptamos el empeoramiento y hacemos la actualización $x = y$. Por último actualizamos la temperatura $T = \alpha \cdot T = 0.3 \cdot 2586.98 = 776.094$.

Iteración 2:
Partimos de $x = (0, 1, 0, 1, 1)$ con $c(x) = 18.8$ y calculamos un vecino $y = (0, 0, 1, 1, 1)$ obteniendo $c(y) = 19.4$. De esta forma $\delta = c(y) - c(x) = 0.6 > 0$, por lo tanto generamos una uniforme para ver si aceptamos el empeoramiento, obteniendo $u = 0.75$. Como $u = 0.75 < exp(-\frac{\delta}{T}) = 0.99$, aceptamos el empeoramiento y actualizamos: $x = y$, $T = 0.3 \cdot 776.094 = 232.8282$.

Iteración 3:
Calculamos un vecino de $x = (0, 0, 1, 1, 1)$ con $c(x) = 19.4$, por ejemplo $y = (0, 1, 1, 0, 1)$. Tenemos que $c(y) = 9.8$ y $\delta = -9.6 < 0$, luego hemos encontrado un vecino mejor y hacemos $x = y$. Actualizamos la temperatura $T = 0.3 \cdot 232.8282 = 69.84$.

Iteración 4:
Calculamos un vecino de $x = (0, 1, 1, 0, 1)$ con $c(x) = 9.8$, $y = (1, 0, 1, 0, 1)$. Tenemos que $c(y) = 9.4$ y $\delta = -0.4 < 0$, como es un vecino mejor actualizamos $x = y$. Actualizamos la temperatura $T = 0.3 \cdot 69.84 = 20.9545$.

Iteración 5:
Calculamos un vecino de $x = (1, 0, 1, 0, 1)$ con $c(x) = 9.4$, $y = (1, 0, 1, 1, 0)$. Tenemos que $c(y) = 9$ y $\delta = -0.4 < 0$, entonces mejora la solución y actualizamos $x = y$. Actualizamos la temperatura $T = 0.3 \cdot 20.9545 = 6.2863$.

Iteración 6:
Calculamos un vecino de $x = (1, 0, 1, 1, 0)$ con $c(x) = 9$, $y = (1, 1, 1, 0, 0)$. Tenemos que $c(y) = 7.4$ y $\delta = -1.6 < 0$, entonces como mejora la solución actualizamos $x = y$. Además $T = 0.3 \cdot 6.2863 = 1.859$.

Iteración 7:
Calculamos un vecino de $x = (1, 1, 1, 0, 0)$ con $c(x) = 7.4$, $y = (1, 1, 0, 1, 0)$. Se verifica que $c(y) = 8.4$ y $\delta = 1 > 0$, entonces como no mejora la solución generamos una uniforme y obtenemos $u = 0.3 < exp(-\frac{\delta}{T}) = 0.59$. Por lo tanto actualizamos la solución y hacemos $T = 0.3 \cdot 1.859 = 0.56577$.

Iteración 8:
Calculamos un vecino de $x = (1, 1, 0, 1, 0)$ con $c(x) = 8.4$, $y = (0, 1, 1, 1, 0)$, $c(y) = 9.4$ y $\delta = 1 > 0$. Entonces como no mejora la solución generamos una uniforme y obtenemos $u = 0.15 < exp(-\frac{\delta}{T}) = 0.17$. De este modo actualizamos la solución y hacemos $T = 0.3 \cdot 0.56577 = 0.1697$.

Como hemos completado dos iteraciones consecutivas sin mejorar la solución, el algoritmo terminaría ya que verifica el criterio de parada que habíamos impuesto. Por lo tanto, la solución final obtenida es $x = (1, 1, 1, 0, 0)$ con un valor óptimo de $c(x) = 7.4$. Cabe destacar que obtenemos convergencia al óptimo global del problema.

2.6. Algoritmos genéticos

2.6.1. Introducción

Los algoritmos genéticos están basados en los mecanismos de la genética y de la selección natural. Como se verá más adelante, es la única metaheurística que trabaja simultáneamente con dos conjuntos de soluciones factibles, que las considerará como individuos de una población que se cruza, reproduce y puede incluso mutar para sobrevivir. Fueron introducidos por Holland (1975) en su trabajo *Adaptation in Natural and Artificial Systems*. El objetivo de su investigación era doble: por un lado, explicar de forma rigurosa los procesos de adaptación natural en los seres vivos; y por otro lado, diseñar programas de ordenador basados en dichos mecanismos naturales. La naturaleza nos enseña que los mecanismos de adaptación y supervivencia funcionan bien en los seres vivos. ¿Podrían adaptarse estos mecanismos a los complejos sistemas artificiales?

Como se ha comentado anteriormente, se parte de un conjunto de soluciones iniciales. Sean en nuestro ejemplo de referencia, Ejemplo 1, las siguientes soluciones iniciales:

$$\begin{array}{ll} s^1 = (0,1,1,0,1) & f(s^1) = 9.8 \\ s^2 = (1,1,0,0,0) & f(s^2) = 22.4 \\ s^3 = (0,1,0,0,0) & f(s^3) = 20.4 \\ s^4 = (1,0,0,1,1) & f(s^4) = 18.4 \end{array}$$

2.6.2. Descripción del algoritmo

Operadores básicos

Con los algoritmos genéticos se consideran estas soluciones como individuos de una especie que evolucionan para conseguir adaptarse y mejorar dicha especie. La calidad de los individuos y, en definitiva de la especie, se mide por la función de aptitud. En nuestro ejemplo, la función de cada solución s es $c(s)$, la función objetivo del problema teniendo en cuenta la penalización añadida si no se verifica alguna de las restricciones.

Las relaciones entre los individuos se ajustan a unos operadores que se analizan a continuación.

- Operador de selección
  No todos los individuos sobreviven durante un determinado tiempo, por ejemplo, hasta la madurez; sólo lo harán los más aptos, es decir, los que tengan una mejor función de aptitud. Una posibilidad para realizar esto sería considerando que sólo sobrevive la mitad de los individuos siendo la probabilidad de supervivencia de cada individuo proporcional a su función de aptitud. En el ejemplo de referencia sólo sobrevivirán 2 de los 4 individuos y la probabilidad con la que serán elegidos viene indicada por la siguiente tabla:

Individuo	Aptitud	% Total	Prob. Repr.
s^1	9.8	13.8	0.138
s^2	22.4	31.6	0.316
s^3	20.4	28.7	0.287
s^4	18.4	25.9	0.259
Suma	71	100.0	1.000

Los individuos elegidos son s^1 y s^4. Se permite la repetición.

- Operador de cruce
Los dos individuos seleccionados y que han llegado, por tanto, a la madurez, se pueden cruzar para mejorar la especie. La forma en que se cruzan y, en definitiva, que permite transmitir sus cualidades a las generaciones futuras, es lo que distinguirá unos algoritmos de otros.
En el ejemplo, se supone que los cruces de dos individuos se realizan creando dos nuevos individuos, uno con las 3 primeras componentes del vector que representa el padre y las dos últimas de la madre; y otro con las 3 primeras componentes del vector que representa la madre y las 2 últimas del padre. Los dos hijos así creados y los padres serán la siguiente generación:

$$\begin{array}{ll} s^1 = (0,1,1,0,1) & f(s^1) = 9.8 \\ s^4 = (1,0,0,1,1) & f(s^4) = 18.4 \\ s^5 = (0,1,1,1,1) & f(s^5) = 39.8 \\ s^6 = (1,0,0,0,1) & f(s^6) = 24.4 \end{array}$$

Obsérvese que tenemos un individuo con el mínimo coste de 9.8, al igual que en la generación anterior, pero globalmente, esta nueva generación es menos apta que la anterior porque la suma total de costes es mayor.

- Operador de mutación
Aunque globalmente se mejore, hay determinadas características que no pueden modificarse sólo con los dos operadores vistos anteriormente. En el ejemplo de referencia, nunca podrá conseguirse a partir de la segunda generación obtener dos 0 en la cuarta y quinta componente simultáneamente, por lo que nunca se obtendrá el individuo con la aptitud máxima por este procedimiento.
Se introduce así un operador que perturba alguna de las soluciones obtenidas. Evidentemente, esta perturbación puede empeorar la aptitud del individuo mutado, pero si éste es el caso, los otros dos operadores anularán esta disminución y sólo sobrevivirán aquellas

mutaciones positivas.
El operador mutación se aplica muy poco, es decir, con una probabilidad muy baja. En el ejemplo de referencia podría ser de la siguiente forma: con una probabilidad 0.01 alterar las componentes del vector que representa a los individuos.
Observación: En el caso de problemas de minimización hay que modificar la función de adaptación para que sean más probables los de menor valor.

A continuación consideraremos el problema del viajante y mostraremos 3 posibles operadores distintos de cruce. Para comprender mejor la diferencia, se considera un ejemplo con 12 ciudades y cada recorrido se representará como una de las 12! permutaciones posibles. El padre y la madre que se cruzarán de 3 formas serán:

$$(h, k, c, e, f, d, b, l, a, i, g, j)$$
$$(a, b, c, d, e, f, g, h, i, j, k, l)$$

Cruce de orden

Se realiza de la siguiente forma:

1. Se eligen dos puntos de corte aleatoriamente.
2. Entre estos dos puntos de corte se sitúan los elementos del padre.
3. El resto se van eligiendo de la madre siempre que no hayan sido seleccionados previamente. Se comienza a partir del segundo punto de corte.

Si los puntos de corte son el sexto y el noveno, el hijo será:

$$(d, e, f, g, h, i\,|b, l, a|\,j, k, c).$$

Observación: entre el sexto y el noveno, coinciden todos los del padre; a partir del décimo, se sitúan j y k, al intentar colocar l, se observa que ya ha sido introducido, por lo que se pasa al siguiente, que es a, y luego b, y les ocurre lo mismo, por lo que se coloca el siguiente, c, y así sucesivamente.

Cruce parcial

Se realiza de la siguiente forma:

1. Se eligen dos puntos de corte aleatoriamente.
2. Los caracteres de la zona cortada del padre permutan con los correspondientes de la madre.

Si en el ejemplo, la zona cortada sigue siendo la anterior, el hijo sería:

$$(i, g, c, d, e, f\,|b, l, a|\,j, k, h).$$

Observación: no se desplazan como en el cruce anterior, sino que mantienen la posición del padre en la zona cortada y los de la madre en el resto salvo los que son permutados.

Cruce de ciclo

Genera un hijo de forma que cualquiera de sus caracteres mantiene la posición del padre o de la madre, de acuerdo con sus posiciones en un ciclo. Se opera completando ciclos de sucesión, de ahí el nombre del cruce.

En el ejemplo anterior, para la primera posición hay dos opciones para el hijo: h como el padre ó a como la madre; se elige la h. Por lo tanto tendríamos $(h, -, -, -, -, -, -, -, -, -, -, -)$.

Al hacer esta elección, está claro que a debe ir en la posicion del padre obteniendo $(h, -, -, -, -, -, -, -, a, -, -, -)$. Esto obliga a seleccionar la ciudad i del vector padre, ciudad bajo la a en el vector madre, obteniendo $(h, -, -, -, -, -, -, -, a, i, -, -)$.

Por la misma razón, los caracteres j y l deben mantener la posición del padre, llegando a $(h, -, -, -, -, -, -, l, a, i, -, j)$.

Se ha definido así un ciclo al elegir h en la primera posición: $h \to a \to i \to j \to l$, que denotaremos por ciclo 1.

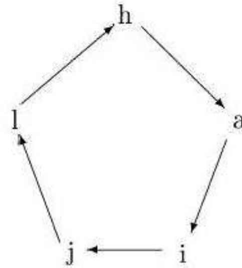

Figura 2.3: Ciclo 1

Si se hubiera elegido el carácter a para la primer posición, el ciclo sería el mismo con el orden inverso. Se definen así varios ciclos, en el ejemplo son 3 más el ciclo unitario formado en la tercera posición al coincidir el carácter c en el padre y la madre.

Denominando los ciclos $1, 2, 3$ y U para el unitario, las posiciones de las 12 ciudades se asignan a estos ciclos:

$$(1, 2, U, 3, 3, 3, 2, 1, 1, 1, 2, 1).$$

La selección aleatoria fija uno de los ciclos con los caracteres del padre y el resto para la madre. En el ejemplo, se elige el ciclo 1 con los caracteres del padre y se completa el resto con lo de la madre quedando:

$$(h, b, c, d, e, f, g, l, a, i, k, j).$$

2.6.3. Aplicación al ejemplo de referencia

Consideremos el Ejemplo 1 y apliquemos el algoritmo descrito para calcular la solución óptima. Como se ha comentado anteriormente, se parte de un conjunto de soluciones iniciales:

$$\begin{array}{ll} s^1 = (0,1,1,0,1) & f(s^1) = 9.8 \\ s^2 = (1,1,0,0,0) & f(s^2) = 22.4 \\ s^3 = (0,1,0,0,0) & f(s^3) = 20.4 \\ s^4 = (1,0,0,1,1) & f(s^4) = 18.4 \end{array}$$

La probabilidad con la que serán elegidos viene indicada por la siguiente tabla:

Individuo	Aptitud	% Total	Prob. Repr.
s^1	9.8	13.8	0.138
s^2	22.4	31.6	0.316
s^3	20.4	28.7	0.287
s^4	18.4	25.9	0.259
Suma	71	100.0	1.000

Suponemos que los cruces de dos individuos se realizan creando dos nuevos individuos, uno con los 3 primeros bits del padre y los dos últimos de la madre y otro con los 3 primeros bits de la madre y los 2 últimos del padre. Los hijos así creados serán la siguiente generación:

Iteración 1

$$\begin{array}{ll} s^5 = (0,1,1,1,1) & f(s^5) = 39.8 \\ s^6 = (1,0,0,0,1) & f(s^6) = 24.4 \\ s^7 = (1,1,0,0,0) & f(s^7) = 22.4 \\ s^8 = (0,1,0,0,0) & f(s^8) = 20.4 \end{array}$$

Iteración 2

$$\begin{array}{ll} s^9 = (1,1,0,0,0) & f(s^9) = 22.4 \\ s^{10} = (0,1,0,0,0) & f(s^{10}) = 20.4 \\ s^{11} = (1,0,0,1,1) & f(s^{11}) = 18.4 \\ s^{12} = (0,1,1,0,1) & f(s^{12}) = 9.8 \end{array}$$

Iteración 3

$$\begin{array}{ll} s^{13} = (1,0,0,0,1) & f(s^{13}) = 24.4 \\ s^{14} = (0,1,1,1,1) & f(s^{14}) = 39.8 \\ s^{15} = (1,1,0,0,0) & f(s^{15}) = 22.4 \\ s^{16} = (0,1,0,0,0) & f(s^{16}) = 20.4 \end{array}$$

Iteración 4

$$\begin{array}{ll} s^{17} = (1,1,0,0,0) & f(s^{17}) = 22.4 \\ s^{18} = (0,1,0,0,0) & f(s^{18}) = 20.4 \\ s^{19} = (1,0,0,1,1) & f(s^{19}) = 18.4 \\ s^{20} = (0,1,1,0,1) & f(s^{20}) = 9.8 \end{array}$$

Iteración 5

$$\begin{array}{ll} s^{21} = (1,0,0,0,1) & f(s^{21}) = 24.4 \\ s^{22} = (0,1,1,1,1) & f(s^{22}) = 39.8 \\ s^{23} = (1,1,0,0,0) & f(s^{23}) = 22.4 \\ s^{24} = (0,1,0,0,0) & f(s^{24}) = 20.4 \end{array}$$

Iteración 6

$$\begin{array}{ll} s^{25} = (1,1,0,0,0) & f(s^{25}) = 22.4 \\ s^{26} = (0,1,0,0,0) & f(s^{26}) = 20.4 \\ s^{27} = (1,0,0,1,1) & f(s^{27}) = 18.4 \\ s^{28} = (0,1,1,0,1) & f(s^{28}) = 9.8 \end{array}$$

Iteración 7

$$\begin{array}{ll} s^{29} = (0,1,1,1,1) & f(s^{29}) = 39.8 \\ s^{30} = (1,0,0,0,1) & f(s^{30}) = 24.4 \\ s^{31} = (1,1,0,0,0) & f(s^{31}) = 22.4 \\ s^{32} = (0,1,0,0,0) & f(s^{32}) = 20.4 \end{array}$$

Iteración 8

$$\begin{array}{ll} s^{33} = (1,1,0,0,0) & f(s^{33}) = 22.4 \\ s^{34} = (0,1,0,0,0) & f(s^{34}) = 20.4 \\ s^{35} = (1,0,0,1,1) & f(s^{35}) = 18.4 \\ s^{36} = (0,1,1,0,1) & f(s^{36}) = 9.8 \end{array}$$

Después de 8 iteraciones, hemos visto que la mejor solución es $x = (0,1,1,0,1)$, que sabemos, de otros algoritmos, que no es el óptimo global, por lo tanto hemos llegado a un óptimo local.

2.7. Colonias de hormigas

2.7.1. Introducción

Los algoritmos ACO (*Ant Colony Optimization*) son modelos inspirados en el comportamiento de colonias de hormigas reales. Los estudios realizados explican cómo animales casi ciegos, como son las hormigas, son capaces de seguir la ruta más corta en su camino de ida y vuelta entre la colonia y una fuente de abastecimiento. Esto es debido a que las hormigas pueden "transmitirse información" entre ellas gracias a que cada una de ellas, al desplazarse, va dejando un rastro de una sustancia llamada feromona a lo largo del camino seguido.

Así, mientras una hormiga aislada se mueve de forma esencialmente aleatoria, los "agentes" de una colonia de hormigas detectan el rastro de feromona dejado por otras hormigas y tienden a seguir dicho rastro. Éstas a su vez van dejando su propia feromona a lo largo del camino recorrido y por tanto lo hacen más atractivo, puesto que se ha reforzado el rastro de feromona. Sin embargo, la feromona también se va evaporando con el paso del tiempo provocando que el rastro de feromona sufra y se debilite. De esta manera se limita el crecimiento de los rastros por lo que la solución adoptada podría corresponder a un óptimo local.

En definitiva, puede decirse que el proceso se caracteriza por una retroalimentación positiva, en la que la probabilidad con la que una hormiga escoge un camino aumenta con el número de hormigas que previamente hayan elegido el mismo camino.

El primer algoritmo de este tipo fue denominado "*Ant System*" para la resolución del Problema del Viajante, obteniendo unos resultados no muy afortunados, pero que causaron la curiosidad de gran número de investigadores que fueron modificando el algoritmo y aplicando esta técnica a este conjunto de problemas.

Los algoritmos de optimización basados en colonias de hormigas se han aplicado fundamentalmente a problemas de optimización combinatoria. Dentro de este conjunto de problemas, se han utilizado en la resolución de problemas de optimización combinatoria NP-duros en los que las técnicas clásicas ofrecen resultados no muy convincentes, y problemas de caminos mínimos donde la instancia del problema varía en el tiempo.

Como ya se ha mencionado, el primer problema donde se aplicaron este tipo de algoritmos fue el Problema del Viajante (*Traveling Salesman Problem*), instancia de un problema NP-duro, que además incluye la resolución de un camino mínimo. Este problema se ha convertido en un estándar para la realización de pruebas en modelos posteriores al inicial planteado por Bonabeau et al (1999).

Otros conjuntos de problemas tratados posteriormente por este tipo de algoritmos, fueron el problema de asignación cuadrática (*Quadratic*

Assignment Problem) y el problema de ordenamiento secuencial (*Sequential Ordering Problem*). Posteriormente, una de las aplicaciones donde mejor se han adaptado este tipo de algoritmos ha sido en el enrutamiento de redes (*Telecommunications Networks*), siendo los primeros trabajos desarrollados los de Schoonderwoerd (1996) y Di Caro et al (1998).

Otro problema muy estudiado por este tipo de algoritmos ha sido el de determinación de rutas de vehículos (*Vehicle Routing Problem*), donde los resultados proporcionados por estos algoritmos han sido aceptables. Además de los anteriores ejemplos de problemas donde este tipo de algoritmos han sido probados, existen otros muchos campos de aplicación donde los resultados están siendo prometedores, como por ejemplo el diseño de circuitos lógicos.

2.7.2. Descripción del algoritmo

Los algoritmos ACO son procesos iterativos. En cada iteración se "lanza" una colonia de m hormigas y cada una de las hormigas de la colonia construye una solución al problema. Las hormigas construyen las soluciones de manera probabilística, guiándose por un rastro de feromona artificial y por una información calculada a priori de manera heurística.

Goss et al (1989), a partir de los resultados obtenidos, desarrolló un modelo para reflejar el comportamiento observado. Si una hormiga se sitúa en una bifurcación después de t unidades de tiempo desde el inicio del experimento, y sabiendo que m_1 hormigas han utilizado la ruta 1 y m_2 la ruta 2, la probabilidad de que la hormiga elija una de las dos rutas, respectivamente, es la siguiente:

$$p_{1(m+1)} = \frac{(m_1 + k)^h}{(m_1 + k)^h + (m_2 + k)^h} \qquad p_{2(m+1)} = 1 - p_{1(m+1)}$$

donde k y h son parámetros del modelo que sirven de ajuste a los datos del experimento. Además, una vez que las hormigas han construido una solución, debe actualizarse la feromona, en el sentido que se explica más adelante. Notemos que, en la explicación que hacemos del método, suponemos que queremos diseñar una ruta de mínimo coste, partiendo de un conjunto de nodos y un conjunto de arcos que los conectan.

El algoritmo de las hormigas trata de simular el comportamiento de las hormigas naturales. Éste debe de ser capaz de admitir las tareas que realiza la hormiga y simular el comportamiento del conjunto. En función del tipo de depósito de la feromona en el medio, existen tres variantes del algoritmo:

1. En función de la densidad: el depósito se realiza durante el transcurso del recorrido y la cantidad depositada de feromona es constante.

2. En función de la cantidad: el depósito se realiza durante el transcurso

del recorrido y la cantidad depositada de feromona está relacionada con la deseabilidad heurística del tramo.

3. En función del ciclo: el depósito se realiza al finalizar el recorrido.

La variante de ciclo fue la que mejor resultados proporcionó y la que se conoce por Ant System (AS) ó Sistema de Hormigas (SH).

El SH se caracteriza porque la actualización de la feromona se realiza una vez que todas las hormigas han completado sus soluciones, y se lleva a cabo como sigue: primero, todos los rastros de feromona se reducen en un factor constante, implementándose de esta manera la evaporación de feromona. A continuación, cada hormiga de la colonia deposita una cantidad de feromona que es función de la calidad de su solución. Inicialmente, el SH no usaba ninguna acción en un segundo plano, pero es relativamente fácil, por ejemplo, añadir un procedimiento de búsqueda local para refinar las soluciones generadas por las hormigas. A grandes rasgos el algoritmo presenta el desarrollo mostrado en el Cuadro 2.4.

```
Procedimiento Ant System
   Inicializar parámetros
   Mientras condición do
      Inicializar nueva_hormiga()
      Mientras estado <> estado fin
         Por cada_arco_posible_movimiento
            Calcular probabilidad_elección
         Fin Por
         Siguiente_posición = política_decisión
         Lista_posiciones = + siguiente_posición
      Fin Mientras
      Realizar evaporación
      Deposito feromona (lista_posiciones)
   Fin Mientras
Fin Procedimiento
```

Cuadro 2.4: Pseudocódigo del algoritmo de las hormigas

Tal y como hemos comentado anteriormente, las hormigas utilizan el depósito de feromonas para recordar su comportamiento. En un primer momento, todos los arcos presentan la misma probabilidad y para ello se considera oportuno introducir un valor pequeño de feromona, lo que hace posible que caminos sin explorar también tengan probabilidad de ser recorridos.

Como se ha indicado, en el algoritmo existen dos procesos fundamentales en su ejecución:

- Elección del movimiento

Una hormiga k elige ir al siguiente nodo con una probabilidad que viene dada con la fórmula:

$$p_{rs}^k = \frac{[r_{rs}]^\alpha \cdot [\eta_{rs}]^\beta}{\sum_{u \in N_r^k} [r_{ru}]^\alpha \cdot [\eta_{ru}]^\beta}, \text{ si } s \in N_k(r)$$

donde:
$N_k(r)$ son los nodos alcanzables por la hormiga k desde el nodo r.
α y β son parámetros que ponderan la importancia de la heurística utilizada y los valores de feromona detectados.
r_{rs} representa el rastro de feromona entre los puntos r y s.
η_{rs} representa el valor de la función heurística elegida.

Cada hormiga k almacena el recorrido realizado para no repetir visitas al mismo nodo.
Se debe tener en cuenta que al definir unos valores de los parámetros α y β puede hacer variar bastante los resultados. Si hacemos $\alpha = 0$ sólo damos importancia a la función heurística elegida, mientras que con $\beta = 0$ sólo se tiene en cuenta los rastros de feromona detectados por la hormiga. Esto último provoca que la probabilidad de construir óptimos locales se fortalezca, cosa que no debe ocurrir para un buen funcionamiento del algoritmo.

- Actualización de la feromona

  La actualización de la feromona se presenta en dos subprocesos:

  - Evaporación: Los rastros de feromona se reducen un valor constante. Esto es lo que representa la evaporación de la feromona del sistema natural.
    Los diferentes arcos sufren una disminución de su valor de feromona que viene dado por la expresión:

    $$r_{rs} \leftarrow (1-\rho) \cdot r_{rs}, \text{ siendo } \rho \in (0,1] \text{ el factor de evaporación.}$$

  - Deposición: A continuación se realiza el depósito de feromona en la ruta seguida por la hormiga en función de la solución obtenida

    $$r_{rs} \leftarrow (1-\rho) \cdot r_{rs} + \Delta r_{rs}^k, \forall r_{rs} \in S_k$$

    donde S_k es la solución generada por la hormiga k y $\Delta r_{rs}^k = f(C(S_k))$ es la cantidad de feromona que se deposita, función de la calidad de la solución obtenida. La calidad de una solución normalmente viene representada por el inverso de la distancia recorrida.

Cuando todas las hormigas han construido una solución debe actualizarse la feromona en cada arco. Tras la actualización de la feromona puede comenzarse una nueva iteración. De este modo, el resultado final es la mejor solución encontrada a lo largo de todas las iteraciones realizadas.

2.7.3. Aplicación al ejemplo de referencia

Dada la eficiencia del algortitmo de las hormigas para problemas de rutas, presentaremos un ejemplo sencillo de dicho algoritmo aplicado al TSP.

Consideremos una población de cuatro hormigas y cuatro ciudades ($m = n = 4$), cada una de las hormigas partiendo de una ciudad distinta. A continuación definimos la matriz de distancias D y su correspondiente heurística η (inversa, término a término, de cada d_{ij}, $\eta_{ij} = \frac{1}{d_{ij}}$):

$$D = \begin{pmatrix} \infty & 1 & 4 & \sqrt{3} \\ 1 & \infty & 2 & 4 \\ 4 & 2 & \infty & \pi \\ \sqrt{3} & 4 & \pi & \infty \end{pmatrix} \qquad \eta = \begin{pmatrix} - & 1 & 0.25 & 0.577 \\ 1 & - & 0.5 & 0.25 \\ 0.25 & 0.5 & - & 0.318 \\ 0.577 & 0.25 & 0.318 & - \end{pmatrix}$$

Nuestro objetivo será minimizar el camino recorrido por cada hormiga visitando todas las ciudades una sola vez.

Inicializamos la feromona a $\tau_0 = 10$ y empezamos la resolución del problema:

Iteración 1:

Hormiga 1

Consideramos la primera hormiga partiendo de la ciudad 1. Las probabilidades de transición para la primera hormiga se calculan de la siguiente forma:

$$P_1(1,2) = \frac{10 \cdot 1}{10 \cdot 1 + 10 \cdot 0.25 + 10 \cdot 0.577} = \frac{1}{1.827} = 0.547$$
$$P_1(1,3) = \frac{10 \cdot 0.25}{10 \cdot 1 + 10 \cdot 0.25 + 10 \cdot 0.577} = \frac{0.25}{1.827} = 0.136$$
$$P_1(1,4) = \frac{10 \cdot 0.577}{10 \cdot 1 + 10 \cdot 0.25 + 10 \cdot 0.577} = \frac{0.577}{1.827} = 0.315$$
$$P_1(2,3) = \frac{10 \cdot 0.5}{10 \cdot 0.5 + 10 \cdot 0.25} = \frac{0.5}{0.75} = 0.666$$
$$P_1(2,4) = \frac{10 \cdot 0.25}{10 \cdot 0.5 + 10 \cdot 0.25} = \frac{0.25}{0.75} = 0.333$$

A continuación mostramos una tabla con las probabilidades de transición y las soluciones:

Para la elección de cada ciudad de destino, generamos un número aleatorio uniforme en (0,1), $u \in \mathbb{U}(0,1)$, de forma que si partimos de la ciudad i, iremos a la ciudad j que verifique que $\sum_{k=1}^{j} p_m(i,k) > u$. En nuestro caso, $p_1(1,2) = 0.547 > 0.279 = u_1 \in \mathbb{U}(0,1)$, por lo tanto elegimos la ciudad 2 como primera ciudad de destino. En el siguiente caso

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
1	-	0.547	0.136	0.315	0.279	(1,2,-)
2	-	-	0.666	0.333	0.671	(1,2,4,-)
4	-	-	1	-	0.931	(1,2,4,3)

tendremos $p_1(2,3) = 0.666 < 0.671 = u_2 \in \mathbb{U}(0,1)$, entonces hacemos $p_1(2,3) + p_1(2,4) = 1 > 0.671 = u_2$ por lo que la siguiente ciudad elegida será la ciudad 4.

Por lo tanto el camino recorrido por la primera hormiga es $(1,2,4,3)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$$C(S_1) = d_{12} + d_{24} + d_{43} + d_{31} = 1 + 4 + \pi + 4 = 12.14$$

$$\Delta\tau_{13}^1 = \frac{1}{12.14} = 0.082$$

Por lo que la aportación de feromona de esta hormiga en el camino $(1,2,4,3)$, $\Delta\tau_{13}^1$, para la próxima iteración es:

$$\Delta\tau_{13}^1 = \begin{pmatrix} 0 & 0.082 & 0 & 0 \\ 0 & 0 & 0 & 0.082 \\ 0.082 & 0 & 0 & 0 \\ 0 & 0 & 0.082 & 0 \end{pmatrix}.$$

Hormiga 2

Consideramos la segunda hormiga partiendo de la ciudad 2.

Análogamente a la primera hormiga, las probabilidades de transición para la segunda hormiga son las siguientes:

$$P_2(2,1) = \frac{10 \cdot 1}{10 \cdot 1 + 10 \cdot 0.5 + 10 \cdot 0.25} = \frac{1}{1.75} = 0.571$$
$$P_2(2,3) = \frac{10 \cdot 0.5}{10 \cdot 1 + 10 \cdot 0.5 + 10 \cdot 0.25} = \frac{0.5}{1.75} = 0.285$$
$$P_2(2,4) = \frac{10 \cdot 0.25}{10 \cdot 1 + 10 \cdot 0.5 + 10 \cdot 0.25} = \frac{0.25}{1.75} = 0.142$$
$$P_2(1,3) = \frac{10 \cdot 0.25}{10 \cdot 0.25 + 10 \cdot 0.577} = \frac{0.25}{0.827} = 0.302$$
$$P_2(1,4) = \frac{10 \cdot 0.577}{10 \cdot 0.25 + 10 \cdot 0.577} = \frac{0.577}{0.827} = 0.697$$

A continuación mostramos una tabla con las probabilidades de transición y las soluciones:

Por lo tanto el camino recorrido por la segunda hormiga es $(2,1,3,4)$, cuyo coste asociado (distancia recorrida) es el siguiente:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
2	0.571	-	0.285	0.142	0.000	(2,1,-)
1	-	-	0.302	0.697	0.031	(2,1,3,-)
3	-	-	-	1	0.637	(2,1,3,4)

$C(S_2) = d_{21} + d_{13} + d_{34} + d_{42} = 1 + 4 + \pi + 4 = 12.14$

$\Delta\tau_{24}^2 = \dfrac{1}{12.14} = 0.082$

Por lo que la aportación de feromona de esta hormiga en el camino $(2, 1, 3, 4)$, $\Delta\tau_{24}^2$, para la próxima iteración es:

$$\Delta\tau_{24}^2 = \begin{pmatrix} 0 & 0 & 0.082 & 0 \\ 0.082 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.082 \\ 0 & 0.082 & 0 & 0 \end{pmatrix}.$$

Hormiga 3

Consideramos la tercera hormiga partiendo de la ciudad 3.

Probabilidades de transición para la tercera hormiga:

$$P_3(3,1) = \frac{10 \cdot 0.25}{10 \cdot 0.25 + 10 \cdot 0.5 + 10 \cdot 0.318} = \frac{0.25}{1.068} = 0.234$$
$$P_3(3,2) = \frac{10 \cdot 0.5}{10 \cdot 0.25 + 10 \cdot 0.5 + 10 \cdot 0.318} = \frac{0.5}{1.068} = 0.468$$
$$P_3(3,4) = \frac{10 \cdot 0.318}{10 \cdot 0.25 + 10 \cdot 0.5 + 10 \cdot 0.318} = \frac{0.318}{1.068} = 0.297$$
$$P_3(2,1) = \frac{10 \cdot 1}{10 \cdot 1 + 10 \cdot 0.25} = \frac{1}{1.25} = 0.8$$
$$P_3(2,4) = \frac{10 \cdot 0.25}{10 \cdot 1 + 10 \cdot 0.25} = \frac{0.25}{1.25} = 0.2$$

A continuación mostramos una tabla con las probabilidades de transición y las soluciones:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
3	0.234	0.468	-	0.297	0.431	(3,2,-)
2	0.800	-	-	0.200	0.021	(3,2,1,-)
1	-	-	-	1	0.963	(3,2,1,4)

Por lo tanto el camino recorrido por la tercera hormiga es $(3, 2, 1, 4)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$C(S_3) = d_{32} + d_{21} + d_{14} + d_{43} = 2 + 1 + \sqrt{3} + \pi = 7.87$

$\Delta\tau_{34}^3 = \dfrac{1}{7.87} = 0.127$

Por lo que la aportación de feromona de esta hormiga, $\Delta\tau_{34}^3$, para la siguiente iteración es:

$$\Delta\tau_{34}^3 = \begin{pmatrix} 0 & 0 & 0 & 0.127 \\ 0.127 & 0 & 0 & 0 \\ 0 & 0.127 & 0 & 0 \\ 0 & 0 & 0.127 & 0 \end{pmatrix}.$$

Hormiga 4

Consideramos la cuarta hormiga partiendo de la ciudad 4. Las probabilidades de transición son las siguientes:

$$P_4(4,1) = \frac{10 \cdot 0.577}{10 \cdot 0.577 + 10 \cdot 0.25 + 10 \cdot 0.318} = \frac{0.577}{1.145} = 0.503$$
$$P_4(4,2) = \frac{10 \cdot 0.25}{10 \cdot 0.577 + 10 \cdot 0.25 + 10 \cdot 0.318} = \frac{0.25}{1.145} = 0.218$$
$$P_4(4,3) = \frac{10 \cdot 0.318}{10 \cdot 0.577 + 10 \cdot 0.25 + 10 \cdot 0.318} = \frac{0.318}{1.145} = 0.277$$
$$P_4(2,1) = \frac{10 \cdot 1}{10 \cdot 1 + 10 \cdot 0.5} = \frac{1}{1.5} = 0.666$$
$$P_4(2,3) = \frac{10 \cdot 0.5}{10 \cdot 1 + 10 \cdot 0.5} = \frac{0.5}{1.5} = 0.333$$

A continuación, mostramos una tabla con las probabilidades de transición y las soluciones:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
4	0.503	0.218	0.277	-	0.524	(4,2,-)
2	0.666	-	0.333	-	0.211	(4,2,1,-)
1	-	-	1	-	0.827	(4,2,1,3)

Por lo tanto el camino recorrido por la cuarta hormiga es $(4, 2, 1, 3)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$C(S_4) = d_{42} + d_{21} + d_{13} + d_{34} = 4 + 1 + 4 + \pi = 12.14$

$\Delta\tau_{43}^4 = \dfrac{1}{12.14} = 0.082$

Por lo que la aportación de feromona de esta hormiga, $\Delta\tau_{43}^4$, para la siguiente iteración es:

$$\Delta\tau_{43}^4 = \begin{pmatrix} 0 & 0 & 0.082 & 0 \\ 0.082 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.082 \\ 0 & 0.082 & 0 & 0 \end{pmatrix}.$$

Agrupando las soluciones en una tabla, vemos que la mejor solución encontrada hasta el momento es el camino $(3, 2, 1, 4)$

Hormiga	$C(S_k)$	Solución
1	12.14	(1,2,4,3)
2	12.14	(2,1,3,4)
3	7.87	(3,2,1,4)
4	12.14	(4,2,1,3)

Una vez acabada la iteración actualizamos la feromona y procedemos con la siguiente iteración.

Actualización de la feromona:

$r_{rs}_nuevo = (1-\rho)\cdot r_{rs}_viejo + \Delta r_{rs}^k,\ \forall r_{rs} \in S_k$
Tomando el factor de evaporación, ρ, igual a 0.2 tenemos:

$$\tau_{rs}^2 = 0.8 \cdot \begin{pmatrix} 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 \end{pmatrix} \cdot 1 + \begin{pmatrix} 0 & 0.082 & 0.164 & 0.127 \\ 0.291 & 0 & 0 & 0.082 \\ 0.082 & 0.127 & 0 & 0.164 \\ 0 & 0.164 & 0.209 & 0 \end{pmatrix}$$

Por lo que la nueva feromona τ_{rs}^2 es:

$$\Delta\tau_{43}^4 = \begin{pmatrix} 8 & 8.082 & 8.164 & 8.127 \\ 8.291 & 8 & 8 & 8.082 \\ 8.082 & 8.127 & 8 & 8.164 \\ 8 & 8.164 & 8.209 & 8 \end{pmatrix}.$$

Iteración 2:

Hormiga 1

Análogamente a la iteración 1, consideramos la primera hormiga partiendo de la ciudad 1, la segunda partiendo de la ciudad 2, la tercera de la ciudad 3 y la cuarta de la ciudad 4. Entonces las probabilidades de transición son las siguientes:

$$P_1(1,2) = \frac{8.082\cdot 1}{8.082\cdot 1 + 8.164\cdot 0.25 + 8.127\cdot 0.577} = \frac{8.082}{14.81} = 0.545$$

$$P_1(1,3) = \frac{8.164 \cdot 0.25}{8.082 \cdot 1 + 8.164 \cdot 0.25 + 8.127 \cdot 0.577} = 0.137$$
$$P_1(1,4) = \frac{8.127 \cdot 0.577}{8.082 \cdot 1 + 8.164 \cdot 0.25 + 8.127 \cdot 0.577} = 0.316$$
$$P_1(2,3) = \frac{8 \cdot 0.5}{8 \cdot 0.5 + 8.082 \cdot 0.25} = \frac{4}{6.02} = 0.664$$
$$P_1(2,4) = \frac{8.082 \cdot 0.25}{8 \cdot 0.5 + 8.082 \cdot 0.25} = 0.335$$

A continuación mostramos una tabla con las probabilidades de transición y las soluciones:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
1	-	0.545	0.137	0.318	0.321	(1,2,-)
2	-	-	0.664	0.335	0.425	(1,2,3,-)
3	-	-	-	1	0.968	(1,2,3,4)

Por lo tanto el camino recorrido por la primera hormiga es $(1,2,3,4)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$$C(S_1) = d_{12} + d_{23} + d_{34} + d_{41} = 1 + 2 + \pi + \sqrt{3} = 7.87$$

Hormiga 2

Consideramos la segunda hormiga partiendo de la ciudad 2. Análogamente se obtiene la siguiente tabla con las probabilidades de transición y solución para la segunda hormiga:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
2	0.579	-	0.279	0.141	0.660	(2,3,-)
3	0.476	-	-	0.563	0.700	(2,3,4,-)
4	1	-	-	-	0.770	(2,3,4,1)

Por lo tanto el camino recorrido por la segunda hormiga es $(2,3,4,1)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$$C(S_2) = d_{23} + d_{34} + d_{41} + d_{12} = 2 + \pi + \sqrt{3} + 1 = 7.87$$

Hormiga 3

Consideramos la tercera hormiga partiendo de la ciudad 3. Análogamente se obtiene la siguiente tabla con las probabilidades de transición y solución para la tercera hormiga:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
3	0.230	0.468	-	0.290	0.070	(3,1,-)
1	-	0.630	-	0.367	0.150	(3,1,2,-)
2	-	-	-	1	0.650	(3,1,2,4)

Por lo tanto el camino recorrido por la tercera hormiga es $(3, 1, 2, 4)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$$C(S_3) = d_{31} + d_{12} + d_{24} + d_{43} = 4 + 1 + 4 + \pi = 12.14$$

Hormiga 4

Consideramos la cuarta hormiga partiendo de la ciudad 4. Análogamente se obtiene la siguiente tabla con las probabilidades de transición y solución para la cuarta hormiga:

	Probabilidad de transición				Uniforme	Solución
	1	2	3	4		
4	0.498	0.220	0.281	-	0.300	(4,1,-)
1	-	0.798	0.201	-	0.590	(4,1,2,-)
2	-	-	1	-	0.770	(4,1,2,3)

Por lo tanto el camino recorrido por la cuarta hormiga es $(4, 1, 2, 3)$, cuyo coste asociado (distancia recorrida) es el siguiente:

$$C(S_4) = d_{41} + d_{12} + d_{23} + d_{34} = \sqrt{3} + 1 + 2 + \pi = 7.87$$

Agrupando las soluciones en una tabla, vemos que la mejor solución encontrada hasta el momento es el camino $(1, 2, 3, 4)$

Hormiga	$C(S_k)$	Solución
1	7.87	(1,2,3,4)
2	7.87	(2,3,4,1)
3	12.14	(3,1,2,4)
4	7.87	(4,1,2,3)

Por lo tanto, en dos iteraciones hemos llegado a la solución óptima: (1,2,3,4), cuyo coste asociado es 7.87.

Capítulo 3

Aplicación a un ejemplo real

El objetivo de este capítulo es aplicar todas las técnicas heurísticas que presentamos en las secciones anteriores sobre un ejemplo real. Además, compararemos todos los resultados obtenidos para tener una idea de la efectividad de cada una de las técnicas en nuestro ejemplo en concreto. Los algoritmos se han programado en lenguaje MATLAB y el código se presenta en el Apéndice A. Comenzaremos definiendo formalmente el problema que nos interesa resolver para posteriormente ir presentando los resultados con cada una de las técnicas.

3.1. Definición del problema

Dispersas a lo largo del territorio nacional podemos encontrar 20 granjas de producción lechera que pertenecen a una misma empresa del sector lácteo.

La comunidad europea impone restricciones sanitarias de forma que la empresa tiene el deber de hacer análisis de la leche de sus granjas periódicamente garantizando que el producto que suministran cumple todos los requisitos impuestos por las regulaciones alimentarias. Dicha empresa está interesada en construir un cierto número de plantas de análisis de leche de forma que ninguna granja esté a más de 180 kilómetros de la planta más cercana.

Por las condiciones geográficas y administrativas las plantas pueden estar en la misma localización que las granjas o en ciertos terrenos que la empresa ya dispone actualmente. En la figura 3.1 podemos observar la localización de las granjas (con una chincheta amarilla), así como los terrenos de los que dispone la empresa para poder construir las plantas (con una chincheta roja).

Por lo tanto el problema consiste en:

- Determinar dónde se deben construir las plantas de forma que se verifique la restricción de que cada granja no debe estar a más de

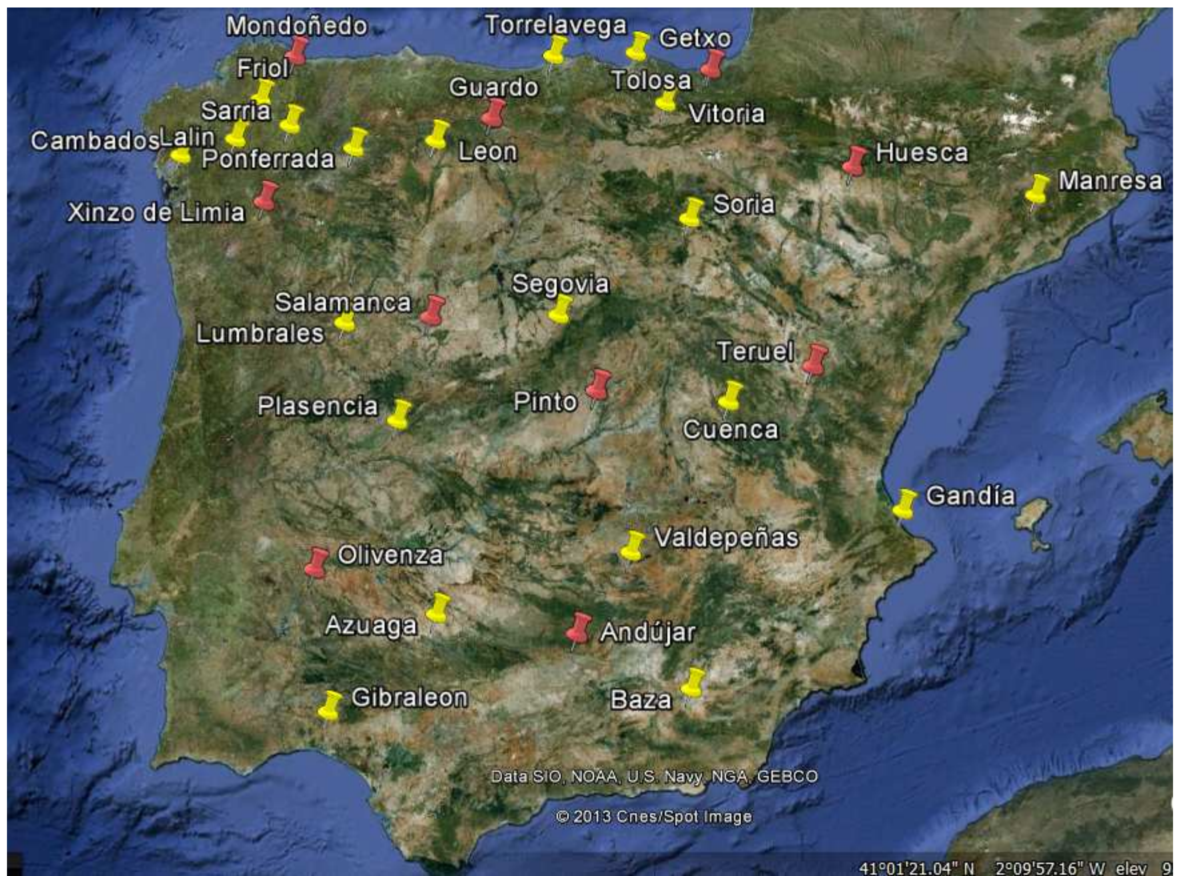

Figura 3.1: Localización de granjas (amarillo) y terrenos disponibles (rojo)

180 kilómetros de distancia de su planta más cercana y además nos interesa que esté lo más próxima posible.

- Notemos que un problema derivado del anterior es determinar cuál es el número mínimo de plantas que se deben construir para que el problema tenga solución.

Definamos el problema formalmente desde el punto de vista de la programación matemática. Denotemos por $P = \{i = 1, \ldots, 30 : i$ es una posible localización para construir una planta $\}$ y por $G = \{j = 1, \ldots, 20 : j$ es una localización de las granjas $\}$. Definamos la variable:

$$x_i = \begin{cases} 1 & \text{si construimos una planta en la localización } i \\ 0 & \text{en caso contrario} \end{cases} , \forall i \in P$$

Sea d_{ij} la distancia desde cada posible localización i donde se puede construir una planta hasta cada granja j. El problema de optimización

asociado será:

$$\begin{aligned}
\text{minimizar} \quad & \sum_{j\in G} d_j \\
\text{sujeto a} \quad & C_{ij} = d_{ij}\cdot x_i + M\cdot(x_i - 1) && ,\ \forall i \in P, j \in G \\
& d_j = \min_{i\in P}\{C_{ij}\} \leq 180 && ,\ \forall j \in G \\
& \sum_{i=1,\ldots,30} x_i > 1 \\
& C_{ij} \in \mathbb{R}, d_j \in \mathbb{R}, x_i \in \{0,1\} && ,\ \forall i \in P, j \in G
\end{aligned}$$

Siendo $M = 10^6$ una penalización que se aplicará a las soluciones que no verifiquen las restricciones. Cabe destacar que tenemos una gran cantidad de posibles soluciones para el problema, exactamente $2^{30} - 1$ posibilidades. Por lo tanto, es prácticamente imposible resolver el problema mediante búsqueda exhaustiva de todo el espacio de soluciones.

Supongamos que a priori conocemos el mínimo número de plantas necesarias que hay que construir verificando las restricciones, y lo denotamos por k. En este caso el número de posibles soluciones que tendríamos pasaría a ser $\binom{30}{k}$. De esta forma, una vez fijado el número de plantas a construir, reducimos enormemente el espacio de búsqueda. Entonces, lo que haremos será ir fijando k previamente con distintos valores hasta encontrar el k óptimo, es decir, el número mínimo de plantas que se deben construir. Para ello necesitamos añadir como restricción:

$$\sum_{i=1,\ldots,30} x_i = k$$

Notemos que k es un número entero que sólo puede tomar valores entre 0 y 30.

A la hora de aplicar los algoritmos heurísticos para resolver el problema, para no restringir el espacio de búsqueda únicamente a las soluciones factibles (ya que complicaría mucho su resolución), incorporaremos en la función objetivo una penalización para aquellas soluciones que no verifiquen las restricciones. Concretamente, dada una solución penalizaremos con 10^6 la función objetivo tantas veces como el número de granjas que no tengan una planta a menos de 180 kilómetros. Es decir, dada una solución x calcularíamos:

$$d_j = \min_{i\in P}\{d_{ij}\cdot x_i : x_i = 1\},\ \forall j \in G.$$

Entonces, si denotamos por $p = |\{j \in P : d_j > 180\}|$, añadiríamos a la función objetivo la penalización $10^6 \cdot p$.

3.2. Solución del problema

En primer lugar, comenzaremos definiendo cómo vamos a calcular un vecino dada una solución $x \in \{0,1\}^{30}$, ya que es necesario para poder aplicar las heurísticas presentadas en el Capítulo 2. Supongamos que conocemos el número de plantas a construir, k, entonces sabemos que una posible solución será, obligatoriamente, un vector que contenga k unos y $30-k$ ceros. De este modo, para calcular un vecino simplemente intercambiamos en el vector un cero con un uno. Supongamos que $k = 2$ y

$$x_i = \begin{cases} 1 & \text{si } i = 5, 8 \\ 0 & \text{en caso contrario} \end{cases}$$

Entonces un posible vecino podría ser:

$$\widehat{x}_i = \begin{cases} 1 & \text{si } i = 20, 8 \\ 0 & \text{en caso contrario} \end{cases}$$

Es importante destacar que dada una solución x y fijado el número de plantas a construir k, en total habría $k \cdot (30-k)$ posibles vecinos de x si los calculamos según el esquema anterior.

De aquí en adelante supondremos fijado el número de plantas a construir k.

Algoritmo voraz

Lo primero que vamos a presentar es un algoritmo voraz con la idea de utilizar la solución que nos proporciona como solución inicial de los siguientes algoritmos. La idea es la siguiente:

Para cada posible localización de la central lechera, calculamos todas las distancias de la misma a cada una de las granjas, que será una matriz $D = (d_{ij})_{i \in P, j \in G}$ (cada fila representará una localización y cada columna, una granja). A continuación sumamos por filas la matriz D, quedándonos con las k menores sumas de distancias. Dichas filas denotarán las k localizaciones elegidas como solución voraz que no tiene porqué ser necesariamente factible.

Formalmente tendríamos que calcular:

$$v_i = \sum_{j \in G} d_{ij}, \forall i \in P.$$

Si denotamos por V los índices de las k plantas que tienen menor v_i, la solución obtenida con el algoritmo voraz sería:

$$x_i = \begin{cases} 1 & \text{si } i \in V \\ 0 & \text{en caso contrario} \end{cases}$$

Lo que hace este algoritmo simplemente es calcular aquellas posibles localizaciones de plantas que están más cercanas a todas las granjas, con lo

cual serán localizaciones centrales en el mapa.

Algoritmo de búsqueda local

A la hora de emplear el algoritmo de búsqueda local hay que tener en cuenta las siguientes consideraciones:

- Tomaremos como solución de partida la proporcionada por el algoritmo voraz.
- Dada una solución x, fijado el número de plantas a construir, k, sabemos que contendrá k unos y $30-k$ ceros. Entonces, para calcular un vecino y generaremos dos números aleatorios, uno entre 1 y k para seleccionar un 1 del vector x, y otro entre 1 y $30-k$ para seleccionar un 0 del vector x, e intercambiamos dichos cero y uno. Por lo tanto, es una forma aleatoria de ir generando los vecinos. Otra posibilidad sería calcular todos los vecinos posibles de x y quedarse con el mejor.

Pongamos un ejemplo sencillo para entender el cálculo de los vecinos de forma aleatoria. Supongamos que tenemos el vector $(1,1,0,1,0,0,0)$, entonces generamos un número aleatorio u_1 entre 1 y 3, y otro, u_2, entre 1 y 4. Si $u_1 = 3$ y $u_2 = 4$, tendríamos que intercambiar el 1 que se encuentra en la posición 4 del vector con el 0 que se encuentra en la posición 7, obteniendo como vecino $(1,1,0,0,0,0,1)$.

Simulated annealing

Sabemos que el peor empeoramiento que se puede producir es que no se verifique ninguna restricción; es decir, que todas las granjas estén a más de 180 kilómetros de su planta más cercana. En este caso, dicho empeoramienro sería $\delta = 20 \cdot 10^6$. A partir de éste, calculamos el valor inicial de la variable de control T (temperatura inicial) de forma que sea lo suficientemente grande para inicialmente movernos por todo el conjunto de la región factible. De esta forma queremos que:

$$exp(-\frac{\delta}{T}) = 0.99 \Leftrightarrow T = -\frac{\delta}{log(0.99)}.$$

Como factor de disminución de la temperatura tomamos $\alpha = 0.3$, de forma que iremos actualizando la temperatura de la forma:

$$T = \alpha T \quad (0 < \alpha < 1).$$

Además, actualizaremos la temperatura cada $N(T) = 100$ iteraciones. Impondremos como criterio de parada que la función objetivo no mejore en 10000 iteraciones consecutivas.

Algoritmo de búsqueda tabú

Consideraremos como duración (número de iteraciones que un movimiento tabú no puede ser utilizado) 15 iteraciones. Además, consideramos como función de aspiración la mejor solución obtenida hasta el momento; es decir, un movimiento tabú podrá ser aceptado si su aplicación mejora el mínimo valor de la función objetivo que se ha alcanzado hasta ahora en la búsqueda tabú.

A diferencia de los algoritmos anteriores, en donde simplemente generábamos un vecino de forma aleatoria en cada iteración, ahora en cada iteración generamos todos los posibles vecinos de la solución y nos quedamos con el mejor.

Además, como éste es un algoritmo de memoria a corto plazo, necesitamos guardar en cada iteración la mejor solución obtenida hasta el momento.

En la figura 3.2 podemos ver la mejor función objetivo que hemos encontrado con cada uno de los algoritmos en función del número de plantas k que se contruyen (que hemos fijado previamente como restricción). Cabe destacar que en la función objetivo se han añadido las penalizaciones derivadas de no verificar la restricción de máxima distancia, de ahí la escala tan alta de la función objetivo.

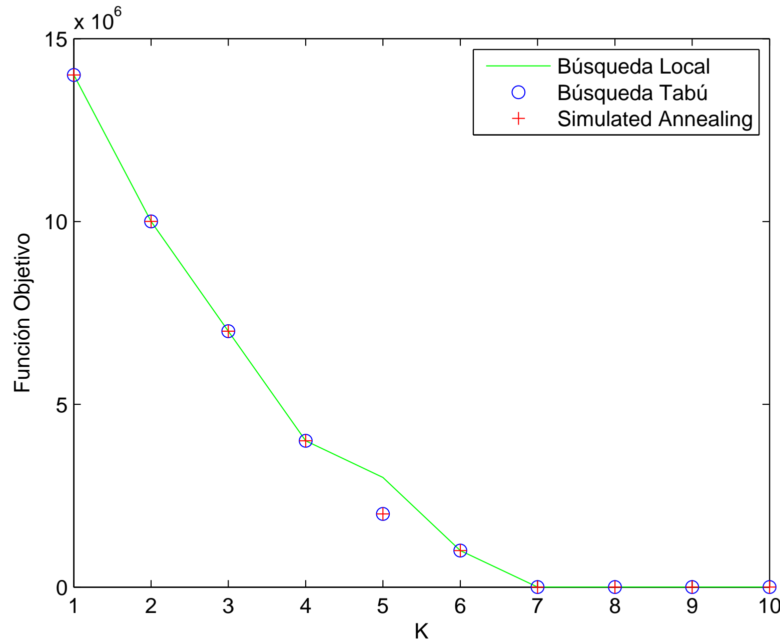

Figura 3.2: Evolución de la función objetivo dependiendo del número de plantas contruidas k

A la vista de la figura 3.2 se observa que el número óptimo de plantas que se deben construir de forma que se verifiquen las restricciones impuestas debe estar cerca de 7, ya que es a partir de donde la función objetivo parece que no contiene penalizaciones. Tras hacer varias pruebas con nuestros algoritmos encontramos que dicho número debe ser exactamente 7, ya que si consideramos $k = 6$, no somos capaces de encontrar una solución factible.

En la tabla 3.1 presentamos las soluciones obtenidas con los diferentes algoritmos. Obsérvese que hemos obtenido exactamente el mismo óptimo con los tres algoritmos y además las soluciones proporcionadas por el algoritmo de búsqueda local y búsqueda tabú son idénticas. La solución proporcionada por el algoritmo de *simulated annealing* se diferencia de las anteriores en que en lugar de construir una planta en Valdepeñas lo hace en Baza. Es importante destacar que las únicas plantas que se contruyen en lugares donde previamente no había ya localizada una granja son las de Salamanca y Teruel.

Algoritmo	**Localización plantas**	**Función objetivo (km)**
Búsqueda local	Vitoria, Salamanca, Manresa,Teruel Valdepeñas, Azuaga, Soria	1661.3362
Búsqueda tabú	Vitoria, Salamanca, Manresa,Teruel Valdepeñas, Azuaga, Soria	1661.3362
Simulated annealing	Vitoria, Salamanca, Manresa,Teruel Baza, Azuaga, Soria	1661.3362

Cuadro 3.1: Soluciones obtenidas con los algoritmos

Por otro lado, en la tabla 3.2, podemos ver cuáles serían las granjas de las que se debería encargar cada planta si se actuara de forma óptima teniendo en cuenta las distancias. Además detallamos cual sería la distancia entre ambos puntos. Recordemos que la solución de *simulated annealing* se diferenciaba por construir una planta en Baza en lugar de Valdepeñas, pero viendo dicha tabla es evidente deducir que es una solución equivalente. También es inmediato comprobar que la mayor distancia de una granja a su planta más cercana es de 173.03 km, que se corresponde a la distancia entre Teruel y Gandía.

En la figura 3.3 podemos ver representada gráficamente la solución en google maps. Con el nombre en rojo denotamos las localizaciones donde deben ser contruidas las plantas. Además hemos diferenciado con colores las granjas por zonas en función de cuál será la planta que se debe encargar de cada granja.

Plantas	Granjas	Distancia (km)
Vitoria	Vitoria	0
	Torrelavega	125.97
	Soria	123.05
	Getxo	61.87
Salamanca	Segovia	131.39
	Plasencia	111.60
	Lumbrales	89.41
Manresa	Manresa	0
Teruel	Cuenca	93.88
	Gandía	173.03
Valdepeñas	Valdepeñas	0
	Baza	152.13
Azuaga	Gibraleon	151.16
	Azuaga	0
Sarria	Lalín	59.54
	Cambados	119.46
	Ponferrada	72.56
	León	153.48
	Sarria	0
	Friol	42.81

Cuadro 3.2: Asignación de granjas a las plantas

Figura 3.3: Solución obtenida con los algoritmos

3.3. Resolviendo el TSP

Una vez contruidas las plantas, a la empresa le interesa que un inspector visite todas las instalaciones para verificar su correcto funcionamiento, tanto las granjas como las plantas. De esta forma le surge la siguiente cuestión, en qué orden debe el inspector visitar todas las instalaciones de forma que recorra el menor número de kilómetros posible.

Claramente, estamos ante un problema de viajante ya que se trata de determinar la ruta óptima que debe realizar el inspector de forma que la longitud de dicha ruta sea lo mínima posible. En este caso vamos a tener 22 lugares que visitar; 20 se corresponden con granjas (en alguna de las cuales también se contruyó una planta) y 2 con dos nuevas localizaciones de plantas (Salamanca y Teruel).

Comenzaremos resolviendo el problema utilizando un solver online gratuito que se encuentra en la página web:

http://www.neos-server.org/neos/solvers/co:concorde/TSP.html

Dicha página utiliza tanto solvers exactos como heurísticos, según elijamos, y además sólo resuelve problemas simétricos. En nuestro caso, utilizaremos el solver exacto. Además, como estamos considerando distancias euclídeas, el problema es simétrico y lo podemos aplicar.

Para resolverlo, definiremos el siguiente fichero en formato TSPLIB:

```
NAME : ProblemaGranjas
COMMENT : Recorremos todas las granjas y plantas
COMMENT : 1 Lalin 2 Cambados 3 Ponferrada 4 Leon 5 Vitoria 6 Segovia
7 Salamanca 8 Plasencia 9 Cuenca 10 Manresa 11 Teruel 12 Gandia
COMMENT : 13 Valdepeñas 14 Baza 15 Gibraleon 16 Azuaga 17 Lumbrales
18 Torrelavega 19 Soria 20 Getxo 21 Sarria 22 Friol
TYPE : TSP
DIMENSION : 22
EDGE_WEIGHT_TYPE : GEO
NODE_COORD_SECTION
1 42.40 -8.07
2 42.31 -8.49
3 42.33 -6.36
4 42.36 -5.34
5 42.51 -2.40
6 40.57 -4.07
7 40.58 -5.40
8 40.02 -6.05
9 40.04 -2.08
10 41.44 1.49
11 40.21 -1.06
```

```
12 38.58 -0.11
13 38.46 -3.23
14 37.29 -2.47
15 37.23 -6.58
16 38.15 -5.41
17 40.56 -6.43
18 43.21 -4.03
19 41.46 -2.28
20 43.20 -3.00
21 42.47 -7.25
22 43.02 -7.48
EOF
```

Como podemos observar, la definición del fichero es muy sencilla. Su estructura es la siguiente:

- En "NAME" ponemos un nombre al problema.
- Si queremos añadir comentarios los escribimos precedidos de la palabra "COMMENT".
- En "TYPE" especificamos el tipo de problema, en este caso se trata de un TSP.
- En "DIMENSION" ponemos el tamaño del problema, en este caso consta de 22 lugares que visitar.
- En "EDGE_WEIGHT_TYPE" especificamos el tipo de datos que tenemos. En este caso disponemos de las coordenadas[1] geográficas de los lugares, lo que se corresponde con la opción GEO.
- En "NODE_COORD_SECTION" ponemos las coordenadas de los 22 lugares en el formato GGG.MM, donde GGG denota los grados y MM los minutos.

En la página mencionada anteriormente se puede encontrar información más detallada acerca de la estructura de los ficheros TSPLIB.

La ruta óptima que obtenemos, especificando como algoritmo del solver Concorde(CPLEX), es la siguiente:

$$1- > 22- > 21- > 3- > 4- > 18- > 20- > 5- > 19- > 6- > 9- > 11- > 10- > 12- > 14- > 13- > 16- > 15- > 8- > 7- > 17- > 2- > 1$$

[1]Las coordenadas se han obtenido a través del servidor: http://www.agenciacreativa.net/coordenadas_google_maps.php que utiliza google maps; y a su vez, mediante el convertidor de decimales a grados, minutos y segundos: http://transition.fcc.gov/mb/audio/bickel/DDDMMSS-decimal.html

con una longitud total de 3369 km.

Los números se corresponden con la numeración de las ciudades especificadas en el fichero presentado previamente. En la Figura 3.4 podemos ver representado gráficamente, sobre google maps, la ruta óptima.

Figura 3.4: Solución obtenida con el solver online

A continuación, mostraremos los resultados obtenidos empleando algoritmos genéticos y algoritmos basados en colonias de hormigas implementados por nosotros.

Algoritmos genéticos

El operador de cruce que utilizaremos es una modificación de los presentados en la Sección 2.6, que detallamos ahora sobre un sencillo ejemplo. Supongamos que tenemos los dos siguientes cromosomas de tamaño 5 que queremos cruzar:

$$\text{Padre} = (3\ 5\ 2\ 4\ 1)$$
$$\text{Madre} = (4\ 2\ 3\ 1\ 5)$$

Seleccionamos aleatoriamente una posición del cromosoma, por ejemplo la posición 2, e intercambiamos el cromosoma de ambos padres.

$$\text{Hijo 1} = (3\ 2\ 2\ 4\ 1)$$
$$\text{Hijo 2} = (4\ 5\ 3\ 1\ 5)$$

A menos que ambos cromosomas coincidan, cada hijo va a tener un entero duplicado. A continuación seleccionamos el entero repetido del hijo 1 y lo intercambiamos con el del hijo 2, situado en esa misma posición:

Hijo 1 = (3 2 3 4 1)
Hijo 2 = (4 5 2 1 5)

Ahora tenemos repetido un entero distinto al anterior, el 3, así que el proceso continúa hasta que seleccionemos del cromosoma del hijo 2 el primer cromosoma seleccionado del padre. En nuestro ejemplo, hasta seleccionar el entero 5 del hijo 2.

Hijo 1 = (4 2 3 4 1)
Hijo 2 = (3 5 2 1 5)

Hijo 1 = (4 2 3 1 1)
Hijo 2 = (3 5 2 4 5)

Hijo 1 = (4 2 3 1 5)
Hijo 2 = (3 5 2 4 1)

Al final del proceso vemos que cada hijo ya no tendrá ningún entero repetido, y por lo tanto estarán bien definidos.

Para llevar a cabo la mutación, seleccionamos aleatoriamente dos posiciones en el cromosoma e intercambiamos los enteros. Por ejemplo, supongamos que tenemos x =(3 1 4 2 5) y que las dos posiciones seleccionadas al azar son la 2 y la 5, la mutación de x será (3 5 4 2 1).

Algunos parámetros que consideramos a la hora de lanzar el algoritmo fueron los siguientes:

- Una población inicial de tamaño 50.
- La probabilidad de supervivencia de la población del 50 %.
- La probabilidad de que se produjese una mutación del 1 %.

En la figura 3.5 podemos ver la evolución de la población a lo largo de las distintas generaciones. Destacamos en color verde la longitud media de las rutas representadas por la población en cada iteración y en azul la longitud de la ruta más corta encontrada hasta cada instante. Como podemos observar hasta la iteración 300 la población va mejorando considerablemente, sobre todo en las iteraciones iniciales, pero a partir de dicha iteración ya no se consigue mejorar. Este comportamiento es razonable porque partimos de una población inicial generada aleatoriamente, con lo cual en las iteraciones iniciales resulta sencillo mejorarlas.

La mejor solución que obtenemos, tras ejecutar el algoritmo genético varias veces, coincide con la presentada en la Figura 3.4.

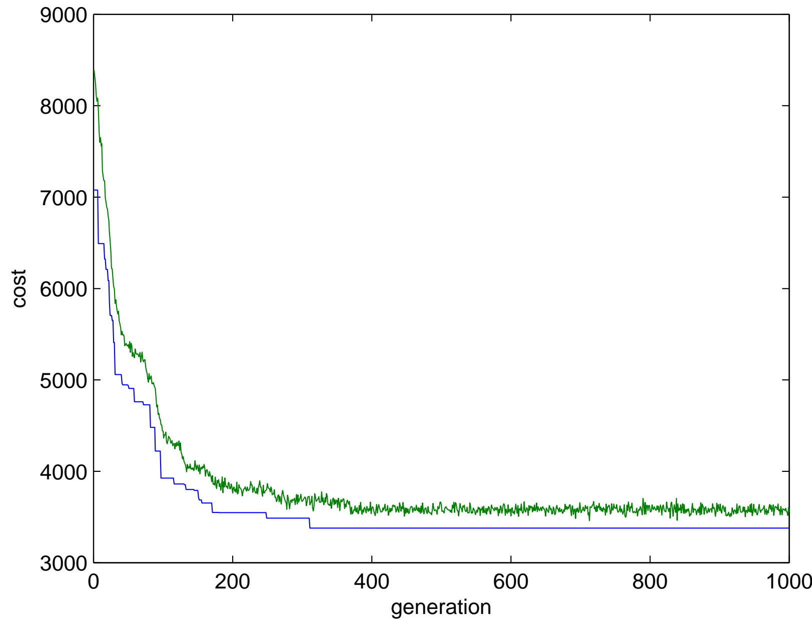

Figura 3.5: Evolución de la población a lo largo de las distintas generaciones

Algoritmos basados en colonias de hormigas

Los parámetros que hemos considerado a la hora de ejecutar el algoritmo son:

$$\alpha = 1, \beta = 4, r_{rs} = 0.1 \quad \forall r, s, \quad \eta_{rs} = \frac{1}{d_{rs}}.$$

siendo d_{rs} la distancia desde la ciudad r a la s. De esta forma, los caminos más cortos en los que se haya depositado más feromena serán los que tengan mayor probabilidad de ser elegidos. En las iteraciones iniciales, las hormigas recorrerán rutas ineficientes y dejarán feromonas en dichos caminos, por lo tanto será necesario que ciertas feromonas se vayan evaporando para que el algoritmo no converja a una ruta no óptima.

La actualización de la feromona la llevaremos a cabo basándonos en la fórmula dada en Bonabeau et al. (1999):

- Primero llevamos a cabo la evaporación de la feromona:

  $$r_{rs} \leftarrow (1-\rho) \cdot r_{rs}, \text{ siendo } \rho = 0.5 \in (0,1] \text{ el factor de evaporación.}$$

- A continuación se realiza la deposición. Empezamos calculando la feromona temporal:

  $$r_{rs}^{temp} = \sum_k \frac{Q}{\text{Distancia ruta hormiga } k}$$

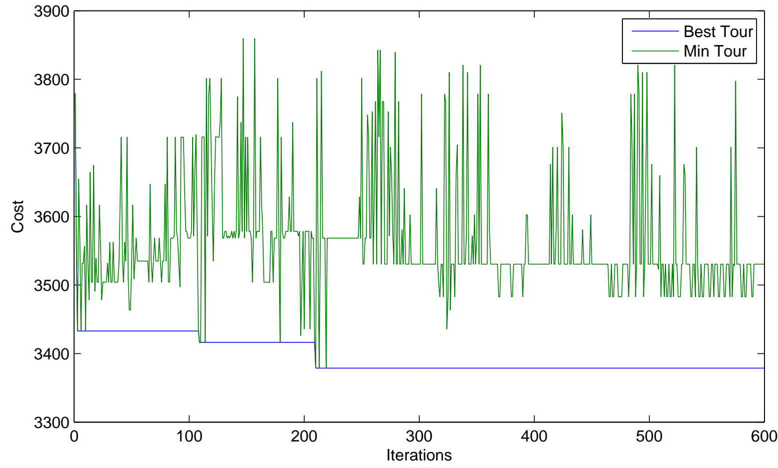

Figura 3.6: Convergencia del algoritmo de las hormigas

siendo Q un número próximo a lo que se piensa que puede ser la longitud de la ruta óptima. En nuestro caso le hemos dado el valor 2000.
Posteriomente, calculamos la feromona depositada por la hormiga que recorre la mejor ruta:

$$r_{rs}^{opt} = \frac{Q}{\text{Distancia de la mejor ruta recorrida por las hormigas}}.$$

Finalmente, actualizamos la feromona de la siguiente manera:

$$r_{rs} = 0.5 \cdot r_{rs} + r_{rs}^{temp} + \xi \cdot r_{rs}^{opt}$$

siendo ξ una constante que pondera la feromona dejada por la hormiga que recorrió la mejor ruta. Para nuestras ejecuciones hemos considerado $\xi = 5$.

En la Figura 3.6, podemos ver la convergencia del algoritmo de las hormigas aplicado a nuestro problema. En azul hemos representado la distancia de la mejor ruta encontrada hasta cada iteración y en verde la mejor ruta encontrada por las hormigas en cada iteración. Como podemos observar, las hormigas encuentran la ruta óptima en una iteración cercana a la 300, y a partir de dicha iteración vemos que las hormigas no encuentran ninguna ruta mejor. Resulta curioso obervar que en lugar de que las hormigas mejoren sus rutas una vez encuentran la mejor solución, la longitud de las rutas oscila considerablemente y sólo en tres ocasiones hay hormigas que recorren la ruta óptima. Además esto ocurre en iteraciones contiguas, lo

que tiene sentido ya que el rastro de feromonas de ese camino será alto en iteraciones próximas, pero después se irá evaporando. Recordemos que hemos considerado como parámetro de evaporación 0.5, por lo que el rastro de feromonas disminuye fuertemente entre dos iteraciones consecutivas.

La mejor solución obtenida con este algoritmo, tras ejecutarlo varias veces, coincide con la presentada en la figura 3.4.

Por lo tanto, hemos obtenido la misma solución tanto con el algoritmo genético como con el de las hormigas, que además sabemos que coincide con el óptimo global del problema.

Capítulo 4

Conclusiones

El objetivo de este TFM ha sido realizar una recopilación y revisión bibliográfica de los principales métodos heurísticos de resolución de problemas de optimización, que encontramos en la vida real en ámbitos como, por ejemplo, la logística, procesos industriales, telecomunicaciones, etc. Debido al elevado número de variables y restricciones existentes en dichos problemas, en ocasiones, es de gran dificultad resolverlos de forma exacta, por lo que estas técnicas son de vital importancia para obtener buenas soluciones en tiempos aceptables.

La bibliografía en este campo es muy extensa, tanto en libros como en revistas de investigación, y tanto en el plano metodológico como en el de las aplicaciones. Hemos incluído amplias referencias, desde los textos más clásicos a los más recientes. A lo largo del trabajo hemos optado por la presentación de cada uno de los algoritmos en una modalidad básica, con la finalidad de que puedan ser adaptados sin mucha dificultad a cada problema real específico.

Con el fin de dar un carácter más pedagógico al trabajo hemos explicado las distintas técnicas de manera intuitiva, ilustrándolas con un ejemplo sencillo común a todas ellas. Y, finalmente, hemos comparado cada una de las técnicas utilizando un ejemplo real de mayor extensión y complejidad. Se han implementado los algoritmos en lenguaje MATLAB, a la vez que nos hemos familiarizado con la herramienta gratuita online CONCORDE.

Este TFM supone una continuación a las asignaturas de investigación operativa estudiadas en el Máster de Técnicas Estadísticas y ha sido realizado de forma paralela a la actividad laboral que desempeño desde febrero de 2012 en *BrandScience*, consultoría estratégica de *OmnicomMediaGroup*.

Apéndice A

Código de programación en MATLAB

- Algoritmo voraz

```
1  function [x,ind,dist,obj]=voraz(f,k)
2  %INPUT:
3  %f: funcion objetivo del problema
4  %k: numero de centrales a construir
5  %OUTPUT:
6  %x: solucion voraz del problema
7  %ind: identificadores de las localizaciones escogidas
8  %dist: vector de distancias de cada granja a su central mas
       cercana
9  %obj: valor de la funcion objetivo
10
11 global d
12 x=zeros(1,30);
13 sumafilas=zeros(1,30);
14 for i=1:30
15     sumafilas(i)=sum(d(i,:));
16 end
17 %Ordenamos las localizaciones de menor a mayor suma de
18 %distancias a todas las granjas
19 [Y,I] = sort(sumafilas);
20 %Nos quedamos con los indices de las k primeras localizaciones
21 ind(1:k)=I(1:k);
22 x(ind)=1;
23 [dist,obj]=feval(f,x);
24 return
```

- Búsqueda Local

```
1  function [x,ind_unos,dist,obj]=vecinal(x0,f,k,nitmax)
2  %INPUT:
3  %x0: solucion inicial
```

```
4  % f: funcion objetivo del problema
5  % k: numero de centrales
6  % nitmax: numero maximo de iteraciones del algoritmo vecinal
7  % OUTPUT:
8  % x: solucion final del algoritmo vecinal
9  % ind_unos: indices de los 1's del vector solucion
10 % dist: vector de distancias de cada granja a su central mas
       cercana
11 % obj: valor de la funcion objetivo
12
13 x=x0; % Partimos de la solucion inicial(voraz en nuestro caso)
14 [dist,obj]=feval(f,x);
15 i=0;
16 while(i<=nitmax)
17     y=vecino(x,k);
18     [dy,fy]=feval(f,y);
19     delta=fy-obj;
20     if(delta < 0)
21         x=y;
22         obj=fy;
23         dist=dy;
24         i=0;
25     else
26         i=i+1;
27     end
28 end
29 [F,I] = sort(x,'descend');
30 ind_unos=I(1:k);
31 return
```

- Búsqueda Tabú

```
1
2  function [xmejor,ind_unos,fmejor_absoluto]=tabu(x0,f,nitmax,k)
3  % INPUT:
4  % x0: solucion inicial
5  % f: funcion objetivo del problema
6  % k: numero de centrales
7  % nitmax: numero maximo de iteraciones del algoritmo vecinal
8  % OUTPUT:
9  % xmejor: solucion final del algoritmo vecinal
10 % ind_unos: indices de los 1's del vector solucion
11 % fmejor_absoluto: valor de la funcion objetivo
12
13 listatabu=zeros(30,30);
14 frecuencias=zeros(30,30);
15
16 x=x0;
17 [dx,fmejor_absoluto]=feval(f,x);
18
19 i=0;
20 while( i <= nitmax )
21         % Generamos los vecinos y nos quedamos con el mejor:
```

```
22          [y,fy,listatabu_sal,frecuencias_sal]=vecinotabu(x,
                listatabu,frecuencias,f,fmejor_absoluto,k);
23          if(fy < fmejor_absoluto)
24                  fmejor_absoluto=fy;
25                  xmejor= y;
26                  i=0;
27          else
28                  i=i+1;
29          end
30          % Actualizamos:
31          x=y;
32          listatabu=listatabu_sal;
33          frecuencias=frecuencias_sal;
34  end
35  [F,I] = sort(xmejor,'descend');
36  ind_unos=I(1:k);
37  return
```

- *Simulated annealing*

```
1   function [x,ind_unos,dist,obj]=templesimulado(x0,f,k,nitmax,
        delta,alfa)
2   %INPUT:
3   %x0: solucion inicial
4   %f: funcion objetivo del problema
5   %k: numero de centrales
6   %nitmax: numero maximo de iteraciones del algoritmo vecinal
7   %delta: parametro de empeoramiento de la funcion objetivo
8   %alfa: factor (entre 0 y 1) de disminucion de la temperatura
9   %OUTPUT:
10  %x: solucion final del algoritmo vecinal
11  %ind_unos: indices de los 1's del vector solucion
12  %dist: vector de distancias de cada granja a su central mas
        cercana
13  %obj: valor de la funcion objetivo
14
15  % Variable de control TEMPERATURA, cuyo valor inicial depende de
        delta
16  T=-delta/log(0.99);
17  % Partimos de una configuracion inicial (la voraz en nuestro
        caso)
18  x=x0;
19  [dist,obj]=feval(f,x);
20  count=0;
21  i=0;
22  % Criterio de parada
23  while(count<=nitmax)
24      while(i <= 100)
25          i=i+1;
26          % Generamos un vecino de manera aleatoria
27          y=vecino(x,k);
28          [dy,fy]=feval(f,y);
29          c=fy-obj;
```

```
30          if(c > 0)
31              prob=exp(-delta/T);
32              u=rand(1);
33              if(u < prob)
34                  x=y;
35                  obj=fy;
36                  dist=dy;
37              end
38              count=count+1;
39          else
40              x=y;
41              obj=fy;
42              dist=dy;
43              count=0;
44          end
45      end
46      i=0;
47      % Disminuimos la temperatura
48      T=T*alfa;
49  end
50  [F,I] = sort(x,'descend');
51  ind_unos=I(1:k);
52  return
```

- Algoritmo Genético

```
1  function [sol,obj,minc,meanc]=genetico(nvar,maxit,popsize,
       mutrate,selection)
2
3  %INPUT:
4  %nvar: numero de variables de optimizacion
5  %maxit: número máximo de iteraciones
6  %popsize: tamaño poblacional
7  %mutrate: tasa de mutación
8  %selection: fracción de la población
9
10 %OUTPUT:
11 %sol: solucion inicial
12 %obj: funcion objetivo del problema
13 %minc: minimo de la poblacion en cada iteracion
14 %meanc: media de la poblacion en cada iteracion
15
16 %Cargamos el GA
17 ff='tspfun'; %función objetivo
18
19 npar = nvar; %# variables de optimizacion
20 Nt   = npar; %# columnas en la matriz poblacional
21
22
23
24 keep=floor(selection*popsize); %#miembros de la poblacion que
       sobrevive
25 M=ceil((popsize-keep)/2); %numero de apareamientos
```

```
26 odds=1;
27 for ii=2:keep
28     odds=[odds ii*ones(1,ii)];
29 end
30 Nodds=length(odds);
31
32 % Crea la población inicial
33 iga=0; % inicia contador de las iteraciones de las generaciones
34 for iz=1:popsize
35     pop(iz,:)=randperm(npar); % poblacion aleatoria
36 end
37 cost=feval(ff,pop); % calcula el coste de poblacion
38 [cost,ind]=sort(cost); % minimo coste en el elemento 1
39 pop=pop(ind,:); % poblacion de una especie con el coste mas bajo
40 minc(1)=min(cost);
41 meanc(1)=mean(cost);
42
43 % Iterar a traves de las generaciones
44 while iga<maxit
45     iga=iga+1;
46     % Empareja y aparea
47     pick1=ceil(Nodds*rand(1,M)); % compañero #1
48     pick2=ceil(Nodds*rand(1,M)); % compañero #2
49
50     ma=odds(pick1); % indices de los padres
51     pa=odds(pick2); % indices de las madres
52     % Apareamiento
53     for ic=1:M
54         mate1=pop(ma(ic),:);
55         mate2=pop(pa(ic),:);
56         indx=2*(ic-1)+1; % empieza en uno y recorre todos
57         xp=ceil(rand*npar); % valor aleatorio entre 1 y N
58         temp=mate1;
59         x0=xp;
60         % Primer cambio:
61         mate1(xp)=mate2(xp);
62         mate2(xp)=temp(xp);
63         xs=find(temp==mate1(xp));
64         xp=xs;
65         while xp~=x0
66             mate1(xp)=mate2(xp);
67             mate2(xp)=temp(xp);
68             xs=find(temp==mate1(xp));
69             xp=xs;
70         end
71         pop(keep+indx,:)=mate1;
72         pop(keep+indx+1,:)=mate2;
73     end
74     % Mutacion
75     nmut=ceil(popsize*npar*mutrate);
76     for ic = 1:nmut
77         row1=ceil(rand*(popsize-1))+1;
78         col1=ceil(rand*npar);
79         col2=ceil(rand*npar);
```

```
80            temp=pop(row1,col1);
81            pop(row1,col1)=pop(row1,col2);
82            pop(row1,col2)=temp;
83            im(ic)=row1;
84        end
85        cost=feval(ff,pop);
86        % Ordena los costes y parámetros asociados
87        part=pop; costt=cost;
88        [cost,ind]=sort(cost);
89        pop=pop(ind,:);
90        % Estadísticas
91        minc(iga)=min(cost);
92        meanc(iga)=mean(cost);
93    end
94
95    % Muestra la salida
96    day=clock;
97    disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6))
          ,0))
98    disp(["optimized function is " ff])
99    format short g
100   disp(["popsize = ", num2str(popsize), " mutrate = ", num2str(
          mutrate), " # par = ", num2str(npar)])
101   disp([" best cost=", num2str(cost(1))])
102   disp(["best solution"])
103   disp([num2str(pop(1,:))])
104
105   % figure(1)
106   % iters=1:maxit;
107   % plot(iters,minc,iters,meanc);
108   % xlabel('generation');ylabel('cost');
109
110
111   % Solucion y funcion objetivo:
112   sol = pop(1,:);
113   obj = cost(1);
114
115   end
```

- Sistema de Hormigas

```
1
2   function [sol,obj,dd]=hormigas(Ncity,maxit,a,b,rr,Q)
3
4   % Input:
5   % Ncity: numero de ciudades del recorrido
6   % maxit: numero maximo de iteraciones
7   % a=0: se selecciona la ciudad mas cercana
8   % b=0: el algoritmo solo funciona ocn feromonas y no con
        distancia de las ciudades
9   % rr: tasa de evaporacion del rastro de feromona en el camino
10  % Q: selecciona el tamaño del camino optimo
11
```

```
12
13  % Output
14
15  % sol:   solucion al problema TSP
16  % obj: distancia del camino optimo, funcion objetivo
17  % dd: dbest y dmin funciones en cada iteracion
18
19  global dcity   % dcity: matriz de distancias entre ciudades
20
21  dbest=9999999;
22  e=5; % parametro que pondera la importancia del rastro de
        feromonas del mejor camino
23
24
25  Nants=Ncity;   % Numero de hormigas
26
27  vis=1./dcity; % heuristica (igual a la inversa de la distancia)
28  phmone=.1*ones(Ncity,Ncity); % inicia las feromonas entre las
        ciudades
29
30  % inicia el camino
31  for ic=1:Nants
32      tour(ic,:)=randperm(Ncity);
33  end
34  tour(:,Ncity+1)=tour(:,1); % cierra el camino
35
36  for it=1:maxit
37  % encuentra el recorrido de las ciudades de cada hormiga
38  % st es la ciudad actual
39  % nxt contiene el resto de ciudades todavia no visitadas
40      for ia=1:Nants
41          for iq=2:Ncity-1
42              st=tour(ia,iq-1); % Ciudad actual
43              nxt=tour(ia,iq:Ncity); % Ciudades que van despues de
                    la actual
44              prob=((phmone(st,nxt).^a).*(vis(st,nxt).^b))./sum((
                   phmone(st,nxt).^a).*(vis(st,nxt).^b)); %
                   probabilidad de transicion
45              rcity=rand;
46              for iz=1:length(prob)
47                  if rcity<sum(prob(1:iz))
48                      newcity=iq-1+iz; % siguiente ciudad a
                            visitar
49                      break
50                  end
51              end
52              temp = tour(ia,newcity); % pone la nueva ciudad
53              % selecciona la siguiente del camino
54              tour(ia,newcity) = tour(ia,iq);
55              tour(ia,iq) = temp;
56          end
57      end
58      % calcula la longitud de cada recorrido y distribucion de
            las feromonas
```

```
59          phtemp=zeros(Ncity,Ncity);
60          for ic=1:Nants
61              dist(ic,1)=0;
62              for id=1:Ncity
63                 dist(ic,1)=dist(ic)+dcity(tour(ic,id),tour(ic,id+1));
64                 phtemp(tour(ic,id),tour(ic,id+1))=phtemp(tour(ic,id),
                       tour(ic,id+1))+Q/dist(ic,1);
65              end
66          end
67                          % actualizamos la mejor solucion
68          [dmin,ind]=min(dist);
69          if dmin<dbest
70              dbest=dmin;
71              sol=tour(ind,:);
72          end
73          % calculo de la feromona del mejor camino recorrido por las
                hormigas
74          ph1=zeros(Ncity,Ncity);
75          for id=1:Ncity
76              ph1(tour(ind,id),tour(ind,id+1))=Q/dbest;
77          end
78          % actualizacion de la feromona
79          phmone=(1-rr)*phmone+phtemp+e*ph1;
80          dd(it,:)=[dbest dmin];
81          [it dmin dbest]
82      end
83
84      obj=dbest; % solucion optima obtenida por el algoritmo
85
86      end
```

Bibliografía

[1] Aarts, E. y Lenstra, J. K., (2003): *"Local search in combinatorial optimization"*. Ed. Princeton University Press

[2] Bhatti, M.A., (2000): *"Practical optimization methods: with mathematica applications"*. Ed. Springer-Verlag.

[3] Bellman, R.E., (1957): *"Dynamic Programming"*. Princeton University Press.

[4] Bonabeau, E., Dorigo, M., y Theraulaz, G., (1999): *"Swarm intelligence from natural to artificial systems"*. Ed. New York: Oxford University Press.

[5] Dantzig, G., (1998): *"Linear Programming and Extensions"*. Princeton University Press.

[6] De Castro, L. N., (2006): *"Fundamentals of natural computing"*. Ed. Chapman and Hall/CRC.

[7] Denardo, E. V., (1982): *"Dynamic programming. Models and applications"*. Ed. Prentice-Hall.

[8] Di Caro, G., y Dorigo, M. , (1998): *"AntNet: Distributed stigmergetic control for communications networks"*. Journal of Artificial Intelligence Research, 9, 317-365.

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

[10] Goss, S., Aron, S., Deneubourg, J. L., y Pasteels, J. M., (1989): *"Self-organized shortcuts in the argentine ant"*. Naturwissenschaften, 76, 579-581

[11] Gurevich, Y., (1999): *"The sequential ASM thesis"*. Bulletin of European Association for Theoretical Computer Science. Number 67, February 1999, pp. 93-124

[12] Hansen, P., (1986): *"The steepest ascent mildest descent heuristic for combinatorial programming"*. Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy.

[13] Haupt, R. L. y Haupt, S. E., (2004): *"Practical genetic algorithms"*. Ed. Wiley (Second Edition).

[14] Hillier, F. y Lieberman, G., (2005): *"Introduction to operations research"*. Ed. McGraw-Hill

[15] Holland, J. H., (1975): *"Adaptation in natural and artificial systems"*. Ed. University of Michigan Press, Ann Arbor. Republished by the MIT press, 1992.

[16] Kennedy, J., Eberhart, R. C., y Shi, Y., (2001): *"Swarm intelligence"*. Ed. San Francisco: Morgan Kaufmann Publishers.

[17] Michalewicz, Z., y Fogel, D. B., (1998): *"How to solve it: Modern Heuristics"*. Ed. Springer.

[18] Minsky, M., (1967): *"Computation: Finite and Infinite Machines"*. Prentice-Hall, Englewood Cliffs, NJ.

[19] Moreno-Vega, J.M. y Moreno-Pérez, J.A., (1999): *"Heurísticas en Optimización"*. Ed. Consejería de Educación, Cultura y Deportes. Gobierno de Canarias.

[20] Ríos Insua, S., (1996): *"Investigación operativa. Programación lineal y aplicaciones"*. Ed. Centro de Estudios Ramón Areces.

[21] Schoonderwoerd, R., (1996): *"Ant-based load balancing in telecommunications networks"*. Adaptive Behavior, 2, 169-207.

[22] Salazar González, J. S., (2001): *"Programación matemática"*. Ed. Díaz de Santos.

[23] Savage, J. E., (1987): *"The complexity of computing"*. Robert E. Krieger, Malabar, Florida.

[24] Taha, H., (2004): *"Investigación de operaciones"*. Ed. Pearson.

[25] Yang, X.-S., (2010): *"Nature-inspired metaheuristic algorithms"*. Ed. Luniver Press (second Edition).