

Procedimientos Metaheurísticos en Optimización Combinatoria

Rafael Martí

*Departament d'Estadística i Investigació Operativa
Facultat de Matemàtiques
Universitat de València
E-mail: rafael.marti@uv.es*

1. INTRODUCCIÓN

Los métodos descritos en este artículo reciben el nombre de algoritmos heurísticos, metaheurísticos o sencillamente heurísticos. Este término deriva de la palabra griega *heuriskein* que significa encontrar o descubrir y se usa en el ámbito de la optimización para describir una clase de algoritmos de resolución de problemas.

En el lenguaje coloquial, optimizar significa poco más que mejorar; sin embargo, en el contexto científico la optimización es el proceso de tratar de encontrar la mejor solución posible para un determinado problema. En un *problema de optimización* existen diferentes soluciones y un criterio para discriminar entre ellas. De forma más precisa, estos problemas se pueden expresar como encontrar el valor de unas *variables de decisión* para los que una determinada *función objetivo* alcanza su valor máximo o mínimo. El valor de las variables en ocasiones está sujeto a unas *restricciones*.

Podemos encontrar una gran cantidad de problemas de optimización, tanto en la industria como en la ciencia. Desde los clásicos problemas de diseño de redes de telecomunicación u organización de la producción hasta los más actuales en ingeniería y re-ingeniería de software, existe una infinidad de problemas teóricos y prácticos que involucran a la optimización.

Algunas clases de problemas de optimización son relativamente fáciles de resolver. Este es el caso, por ejemplo, de los *problemas lineales*, en los que tanto la función objetivo como las restricciones son expresiones lineales. Estos problemas pueden ser resueltos con el conocido método Simplex; sin embargo, muchos otros tipos de problemas de optimización son muy difíciles de resolver. De hecho, la mayor parte de los que podemos encontrar en la práctica entran dentro de esta categoría.

La idea intuitiva de problema "difícil de resolver" queda reflejada en el término científico NP-hard utilizado en el contexto de la complejidad algorítmica. En términos coloquiales podemos decir que un problema de optimización difícil es aquel para el que no podemos garantizar el encontrar la mejor solución posible en un tiempo razonable. La existencia de una gran cantidad y variedad de problemas difíciles, que aparecen en la práctica y que necesitan ser resueltos de forma eficiente, impulsó el desarrollo de procedimientos eficientes para encontrar buenas soluciones aunque no fueran óptimas. Estos métodos, en los que la rapidez del proceso es tan importante como la calidad de la solución obtenida, se denominan heurísticos o aproximados. En Diaz y otros (1996) se recogen hasta ocho definiciones diferentes de algoritmo heurístico, entre las que destacamos la siguiente:

Un método heurístico es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución.

En contraposición a los *métodos exactos* que proporcionan una solución óptima del problema, los *métodos heurísticos* se limitan a proporcionar una buena solución no necesariamente óptima. Lógicamente, el tiempo invertido por un método exacto para encontrar la solución óptima de un problema difícil, si es que existe tal método, es de un orden de magnitud muy superior al del heurístico (pudiendo llegar a ser tan grande en muchos casos, que sea inaplicable).

En este texto consideraremos los llamados problemas de *Optimización Combinatoria*. En estos problemas el objetivo es encontrar el máximo (o el mínimo) de una determinada función sobre un conjunto finito de soluciones que denotaremos por S . No se exige ninguna condición o propiedad sobre la función objetivo o la definición del conjunto S . Es importante notar que dada la finitud de S , las variables han de ser discretas, restringiendo su dominio a una serie finita de valores. Habitualmente, el número de elementos de S es muy elevado, haciendo impracticable la evaluación de todas sus soluciones para determinar el óptimo.

En los últimos años ha habido un crecimiento espectacular en el desarrollo de procedimientos heurísticos para resolver problemas de optimización. Este hecho queda claramente reflejado en el gran número de artículos publicados en revistas especializadas. En 1995 se edita el primer número de la revista *Journal of Heuristics* dedicada íntegramente a la difusión de los procedimientos heurísticos.

Aunque hemos mencionado el caso de la resolución de un problema difícil, existen otras razones para utilizar métodos heurísticos, entre las que podemos destacar:

- El problema es de una naturaleza tal que no se conoce ningún método exacto para su resolución.
- Aunque existe un método exacto para resolver el problema, su uso es computacionalmente muy costoso.
- El método heurístico es más flexible que un método exacto, permitiendo, por ejemplo, la incorporación de condiciones de difícil modelización.
- El método heurístico se utiliza como parte de un procedimiento global que garantiza el óptimo de un problema. Existen dos posibilidades:
 - El método heurístico proporciona una buena solución inicial de partida.
 - El método heurístico participa en un paso intermedio del procedimiento, como por ejemplo las reglas de selección de la variable a entrar en la base en el método Simplex.

Al abordar el estudio de los algoritmos heurísticos podemos comprobar que dependen en gran medida del problema concreto para el que se han diseñado. En otros métodos de resolución de propósito general, como pueden ser los algoritmos exactos de Ramificación y Acotación, existe un procedimiento conciso y preestablecido, independiente en gran medida del problema abordado. En los métodos heurísticos esto no es así. Las técnicas e ideas aplicadas a la resolución de un problema son específicas de éste y aunque, en general, pueden ser trasladadas a otros problemas, han de particularizarse en cada caso. Así pues, es necesario referirse a un problema concreto para estudiar con detalle los procedimientos heurísticos.

En los capítulos segundo y tercero se describen los métodos heurísticos que podríamos denominar “clásicos”. Hemos seleccionado el Problema del Viajante para describir estos métodos por cumplir una serie de propiedades que lo hacen especialmente indicado. Dicho problema puede enunciarse del siguiente modo:

Un viajante de comercio ha de visitar n ciudades, comenzando y finalizando en su propia ciudad. Conociendo el coste de ir de cada ciudad a otra, determinar el recorrido de coste mínimo.

En la siguiente sección introducimos algunos otros problemas combinatorios que también nos ayudaran a explicar e ilustrar algunas de las técnicas así como a plantear ejercicios para el estudiante. Podemos citar las siguientes razones por las que el problema del viajante ha recibido una especial atención.

- Resulta muy intuitivo y con un enunciado muy fácil de comprender.
- Es extremadamente difícil de resolver por lo que resulta un desafío constante para los investigadores del área.
- Es uno de los que más interés ha suscitado en Optimización Combinatoria y sobre el que se ha publicado abundante material.

- Sus soluciones admiten una doble interpretación: mediante grafos y mediante permutaciones, dos herramientas de representación muy habituales en problemas combinatorios, por lo que las ideas y estrategias empleadas son, en gran medida, generalizables a otros problemas.
- La gran mayoría de las técnicas que han ido apareciendo en el área de la Optimización Combinatoria han sido probadas en él, puesto que su resolución es de gran complejidad.

Existen muchos métodos heurísticos de naturaleza muy diferente, por lo que es complicado dar una clasificación completa. Además, muchos de ellos han sido diseñados para un problema específico sin posibilidad de generalización o aplicación a otros problemas similares. El siguiente esquema trata de dar unas categorías amplias, no excluyentes, en donde ubicar a los heurísticos más conocidos:

Métodos de Descomposición El problema original se descompone en subproblemas más sencillos de resolver, teniendo en cuenta, aunque sea de manera general, que ambos pertenecen al mismo problema.

Métodos Inductivos La idea de estos métodos es generalizar de versiones pequeñas o más sencillas al caso completo. Propiedades o técnicas identificadas en estos casos más fáciles de analizar pueden ser aplicadas al problema completo.

Métodos de Reducción Consiste en identificar propiedades que se cumplen mayoritariamente por las buenas soluciones e introducirlas como restricciones del problema. El objeto es restringir el espacio de soluciones simplificando el problema. El riesgo obvio es dejar fuera las soluciones óptimas del problema original.

Métodos Constructivos Consisten en construir literalmente paso a paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración. Estos métodos han sido muy utilizados en problemas clásicos como el del viajante.

Métodos de Búsqueda Local A diferencia de los métodos anteriores, los procedimientos de búsqueda o mejora local comienzan con una solución del problema y la mejoran progresivamente. El procedimiento realiza en cada paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna solución accesible que la mejore.

Si bien todos estos métodos han contribuido a ampliar nuestro conocimiento para la resolución de problemas reales, los métodos constructivos y los de búsqueda local constituyen la base de los procedimientos metaheurísticos. Por ello, estudiaremos en el capítulo segundo los métodos constructivos en el problema del viajante y en el capítulo tercero los métodos de búsqueda local en este mismo problema. El lector podrá encontrar alusiones a lo

largo del texto a cualquiera de los métodos de descomposición, inductivos o de reducción, pero no dedicaremos una sección específica a su estudio. Alternativamente, prestaremos especial atención a los métodos resultantes de combinar la construcción con la búsqueda local y sus diferentes variantes en el capítulo tercero, puesto que puede considerarse un punto de inicio en el desarrollo de método metaheurísticos.

En los últimos años han aparecido una serie de métodos bajo el nombre de Metaheurísticos con el propósito de obtener mejores resultados que los alcanzados por los heurísticos tradicionales. El término metaheurístico fue introducido por Fred Glover en 1986. En este artículo utilizaremos la acepción de heurísticos para referirnos a los métodos clásicos en contraposición a la de metaheurísticos que reservamos para los más recientes y complejos. En algunos textos podemos encontrar la expresión “heurísticos modernos” refiriéndose a los meta-heurísticos (Reeves, 1995). Los profesores Osman y Kelly (1995) introducen la siguiente definición:

Los procedimientos Metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.

Los procedimientos Meta-Heurísticos se sitúan conceptualmente “por encima” de los heurísticos en el sentido que guían el diseño de éstos. Así, al enfrentarnos a un problema de optimización, podemos escoger cualquiera de estos métodos para diseñar un algoritmo específico que lo resuelva aproximadamente.

En estos momentos existe un gran desarrollo y crecimiento de estos métodos. En este artículo vamos a limitarnos a aquellos procedimientos relativamente consolidados y que han probado su eficacia sobre una colección significativa de problemas. Específicamente consideraremos en el capítulo quinto la Búsqueda Tabú, en el sexto el Templado Simulado y en el séptimo los diferentes Métodos Evolutivos, incluyendo los Algoritmos Genéticos y la Búsqueda Dispersa. Los métodos GRASP junto con los Multi-Arranque han sido incluidos en el capítulo cuarto de Métodos Combinados que sirve de “puente” entre los métodos heurísticos y los metaheurísticos.

Es importante notar que para la correcta comprensión y asimilación de los métodos descritos, resulta indispensable su puesta en práctica, para lo cual el lector deberá programar en un ordenador los algoritmos descritos y resolver algún problema de optimización combinatoria. Recomendamos utilizar algún lenguaje de programación de relativo bajo nivel como el C que permita controlar los detalles de implementación. La siguiente sección incluye una colección de problemas de entre los que el lector puede escoger

alguno e ir trabajando con él, aplicando los métodos descritos a lo largo de todo el texto.

Al resolver un problema de forma heurística debemos de medir la calidad de los resultados puesto que, como ya hemos mencionado, la optimalidad no está garantizada. En la sección tercera de este capítulo se recogen los principales métodos para medir la calidad y eficiencia de un algoritmo y poder determinar su valía frente a otros.

1.1. Problemas Estructurados

El objeto de esta sección no es únicamente dar una colección de ejemplos reales, sino el de establecer modelos que han sido muy estudiados. Así, al enfrentarse el lector a un problema dado, tratará de reconocer las estructuras especiales que aparecen en estos modelos y de esta forma se podrá aprovechar la extensa literatura y experiencia computacional al respecto. Además, no debemos olvidar la limitada, pero significativa, importancia práctica de estos modelos.

Problema de la Mochila

Se tienen n objetos donde cada objeto j tiene un peso w_j y un valor v_j . El problema consiste en seleccionar los objetos a incluir en una mochila sin exceder el peso máximo W , de modo que el valor total de los mismos sea máximo.

Para formular el problema se define una variable x_i , por cada objeto i , de modo que vale 1 si el objeto es seleccionado y 0 en caso contrario.

$$\begin{aligned} &MAX \quad v_1x_1 + v_2x_2 + \dots + v_nx_n \\ &s.a. : \\ &w_1x_1 + w_2x_2 + \dots + w_nx_n \leq W \\ &x \geq 0, \text{ entero} \end{aligned}$$

Este problema tiene numerosas aplicaciones tales como:

- La denominada Cutting Stock, en donde hay que cortar una plancha de acero en diferentes piezas.
- Determinar los artículos que puede almacenar un depósito para maximizar su valor total.
- Maximizar el beneficio en asignación de inversiones cuando sólo hay una restricción.

Problema del Cubrimiento de Conjuntos

Este problema, también conocido como *Set Covering*, se puede enunciar del siguiente modo: Sea un conjunto de objetos $S = \{1, 2, \dots, m\}$ y una clase

H de subconjuntos de S , $H = \{H_1, H_2, \dots, H_n\}$ donde cada H_i tiene un coste c_i asociado. El problema consiste en cubrir con coste mínimo todos los elementos de S con subconjuntos H_i .

Para formular el problema se define una variable x_i que vale 1 si H_i está en la solución y 0 en otro caso. También se introduce una matriz $A = \{a_{ij}\}$ en donde el a_{ij} vale 1 si el elemento j de S está en H_i y 0 en otro caso.

$$\begin{aligned} & \text{MIN } c_1x_1 + c_2x_2 + \dots + c_nx_n \\ & \text{s.a. :} \\ & a_{1j}x_1 + a_{2j}x_2 + \dots + a_{nj}x_n \geq 1 \quad j = 1, \dots, m \\ & x_i = 1, 0 \quad i = 1, \dots, n \end{aligned}$$

Este problema tiene diferentes aplicaciones, entre las que podemos destacar la localización de servicios, tales como hospitales, bomberos, etc. y, la asignación de tripulaciones a vuelos.

El problema del Set Covering es relativamente fácil de resolver con métodos de Ramificación y Acotación ya que la solución óptima del problema lineal coincide, en ocasiones, con la del entero o está bastante cerca de él. La dificultad del problema viene del número enorme de variables que suelen aparecer en problemas reales.

Problema del Empaquetado de Conjuntos

A este problema también se le conoce como *Set Packing*. Como en el problema anterior se tienen los conjuntos S y H , pero ahora cada H_i tiene un valor asociado. El objetivo es empaquetar tantos elementos de S como sea posible de forma que el beneficio obtenido sea máximo y no haya solapamientos (ningún elemento de S puede aparecer más de una vez).

En cierto modo, la relajación lineal del Set Covering y la del Set Packing son problemas duales. Sin embargo esto no sirve para establecer una relación entre los problemas enteros originales.

Uno de los ejemplos/aplicaciones es el problema del **Acoplamiento Máximo** o *Matching*. Un acoplamiento es un subconjunto de las aristas de un grafo de manera que cualquier vértice no sea incidente con más de una de esas aristas. El problema del acoplamiento máximo consiste en encontrar un acoplamiento de máximo cardinal.

Problema de la Partición de Conjuntos

Este problema también es conocido como *Set Partitioning* y, al igual que en los dos anteriores, se tienen los conjuntos S y H . Así como en el Set Covering cada elemento de S tiene que aparecer al menos en uno de H , en este problema cada elemento de S tiene que aparecer exactamente en uno

de H , por lo tanto la solución representa una partición del conjunto S . La función objetivo puede ser maximizar o minimizar, según la aplicación.

Aplicaciones:

- Asignación de tripulaciones en una versión más restringida que la anteriormente mencionada.
- Creación de distritos electorales: Asignación de electores a un colegio electoral.

Los tres problemas vistos pueden ser muy útiles para mostrar la transformación y relaciones entre problemas. Así podemos ver que el Set Packing y el Set Partitioning son equivalentes. Para pasar del primero al segundo basta con añadir variables de holgura. La transformación inversa se realiza mediante variables artificiales. Estos dos problemas son más "fáciles" de resolver de forma exacta que el Set Covering ya que en ellos las restricciones lineales están más ajustadas respecto al conjunto de soluciones enteras posibles, por lo que los óptimos de las relajaciones lineales están más cerca de las soluciones enteras.

Problema del Viajante

Este problema, también conocido como *Traveling Salesman Problem (TSP)*, ha sido uno de los más estudiados en Investigación Operativa, por lo que merece una atención especial. Cuando la teoría de la Complejidad Algorítmica se desarrolló, el TSP fue uno de los primeros problemas en estudiarse, probando Karp en 1972 que pertenece a la clase de los problemas difíciles (NP-hard).

Desde los métodos de Ramificación y Acotación hasta los basados en la Combinatoria Poliédrica, pasando por los procedimientos Metaheurísticos, todos han sido inicialmente probados en el TSP, convirtiéndose éste en una prueba obligada para "validar cualquier técnica de resolución de problemas enteros o combinatorios. La librería TSPLIB (Reinelt, 1991) de dominio público contiene un conjunto de ejemplos del TSP con la mejor solución obtenida hasta la fecha y, en algunos casos, con la solución óptima. A efectos de medir empíricamente la bondad de los algoritmos que se describen en los capítulos segundo y tercero, consideraremos un conjunto de 30 ejemplos de la TSPLIB basados en problemas reales con óptimos conocidos.

El Problema del Viajante puede enunciarse del siguiente modo:

Un viajante de comercio ha de visitar n ciudades, comenzando y finalizando en su propia ciudad. Conociendo el coste de ir de cada ciudad a otra, determinar el recorrido de coste mínimo.

Para enunciar el problema formalmente introducimos la siguiente terminología: Sea un grafo $G = (V, A, C)$ donde V es el conjunto de vértices, A

es el de aristas y $C = (c_{ij})$ es la matriz de costes. Esto es, c_{ij} es el coste o distancia de la arista (i, j) .

- Un **camino** (o cadena) es una sucesión de aristas (e_1, e_2, \dots, e_k) en donde el vértice final de cada arista coincide con el inicial de la siguiente. También puede representarse por la sucesión de vértices utilizados.

- Un camino es **simple** o elemental si no utiliza el mismo vértice más de una vez.

- Un **ciclo** es un camino (e_1, e_2, \dots, e_k) en el que el vértice final de e_k coincide con el inicial de e_1 .

- Un ciclo es **simple** si lo es el camino que lo define.

- Un **subtour** es un ciclo simple que no pasa por todos los vértices del grafo.

- Un **tour** o ciclo hamiltoniano es un ciclo simple que pasa por todos los vértices del grafo.

El Problema del Viajante consiste en determinar un tour de coste mínimo. La Figura 1 muestra un grafo de 8 vértices en el que aparece destacado un ciclo hamiltoniano.

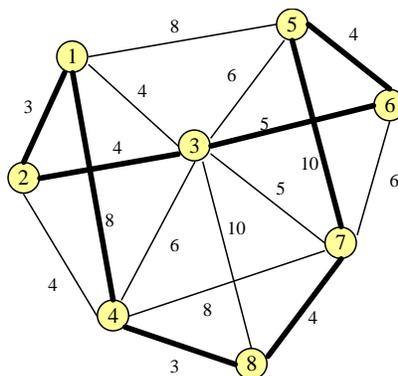


Figura 1. Ciclo Hamiltoniano

Consideraremos, sin pérdida de generalidad, que el grafo es completo; es decir, que para cada par de vértices existe una arista que los une. Notar que, de no ser así, siempre podemos añadir una arista ficticia entre dos vértices con el coste del camino más corto que los une. Así por ejemplo, en el grafo de la Figura 1 podemos añadir una arista entre los vértices 1 y 6 con coste 9 correspondiente al camino 1-3-6.

Entre las aplicaciones más importantes del TSP podemos destacar:

- Fabricación de circuitos integrados
- Rutas de vehículos
- Recogida (robotizada) de material en almacenes
- Instalación de componentes en ordenadores
- Aparece como subproblema en otras aplicaciones

Este problema puede ser formulado mediante un modelo de programación lineal entera con variables binarias. Para ello basta considerar las variables x_{ij} que valen 1 si el viajante va de la ciudad i a la j y 0 en otro caso, y llamar c_{ij} al coste de ir de la ciudad i a la j :

$$\begin{aligned}
 & \text{MIN } \sum_{i < j} c_{ij} x_{ij} \\
 & \text{s.a. :} \\
 & \sum_{i < j} x_{ij} + \sum_{j < i} x_{ji} = 2, \quad i = 1, 2, \dots, n \\
 & \sum_{(i,j) \in \partial(S)} x_{ij} \geq 2, \quad \forall S \subseteq \{1, 2, \dots, n\} \quad 3 \leq |S| \leq [n/2] \\
 & x_{ij} = 0, 1 \quad \forall i < j
 \end{aligned}$$

Donde $\partial(S)$ representa el conjunto de aristas incidentes con exactamente un vértice de S .

Las restricciones que aparecen en segundo lugar (vinculadas a casi la mitad de los subconjuntos de vértices de S) reciben el nombre de restricciones de *eliminación de subtours* y garantizan que la solución sea un tour. El problema es que aparecen en una cantidad del orden de $2^{\frac{n}{2}}$, lo cual hace inmanejable tal formulación. Se han encontrado restricciones alternativas para evitar la formación de subtours que suponen la incorporación de una cantidad polinómica de restricciones (Miller, Tucker y Zemlin, 1960). Aún así, la resolución óptima del problema ha resultado poco eficiente, salvo para ejemplos relativamente pequeños, dado el elevado tiempo de computación requerido por cualquier método exacto.

Problema de la Asignación Cuadrática

Introduciremos el problema mediante el siguiente ejemplo:

”Se tienen n módulos electrónicos y n posiciones en donde situarlos sobre una placa. Sea t_{ik} el número de cables que conectan los módulos i y k , y sea d_{jl} la distancia entre las posiciones j y l de la placa. El problema consiste en determinar la ubicación de los módulos minimizando la longitud total del cable utilizado”

Al igual que en los otros modelos de asignación vistos, se introducen variables x_{ij} que valen 1 si el módulo i se asigna a la posición j y 0 en otro

caso.

$$\begin{aligned}
 & MIN \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n \sum_{l=1}^n t_{ik} d_{jl} x_{ij} x_{kl} \\
 & s.a. : \\
 & \sum_{j=1}^n x_{ij} = 1, \quad i = 1, 2, \dots, n \\
 & \sum_{i=1}^n x_{ij} = 1, \quad j = 1, 2, \dots, n \\
 & x_{ij} = 0, 1
 \end{aligned}$$

El problema se llama cuadrático por la función objetivo ya que el coste viene dado por parejas de variables que aparecen como producto. Así pues la función objetivo es no lineal, aunque se puede transformar en un problema lineal entero introduciendo variables que representen a los productos. Notar que esta transformación obligaría a reformular las restricciones.

Este problema tiene numerosas aplicaciones ya que podemos encontrar en ámbitos muy diversos situaciones como la descrita. Así, por ejemplo, el problema de ubicar determinados servicios (como laboratorios, rayos X, .etc.) en un hospital en donde se conoce el flujo previsto de personal entre tales servicios. Análogamente el guardar determinados productos en un almacén.

El objeto de introducir este problema es doble: por una parte mostrar un problema no lineal, con un gran número de aplicaciones prácticas, que puede transformarse en un PLE y, por otra, presentar uno de los problemas más difíciles (sino el que más) dentro de los ya de por sí difíciles problemas enteros.

Problema de Asignación Generalizada

Se tiene un conjunto $J = \{1, 2, \dots, n\}$ de índices de los trabajos a realizar y otro conjunto $I = \{1, 2, \dots, m\}$ de personas para realizarlos. El coste (o valor) de asignar la persona i al trabajo j viene dado por c_{ij} . Además se tiene una disponibilidad b_i de recursos de la persona i (como por ejemplo horas de trabajo) y una cantidad a_{ij} de recursos de la persona i necesarias para realizar el trabajo j . Con todo esto, el problema consiste en asignar las personas a los trabajos con el mínimo coste (o el máximo valor).

Al igual que en los otros modelos de asignación vistos, se introducen variables x_{ij} que valen 1 si la persona i se asigna al trabajo j y 0 en otro caso.

$$\begin{aligned}
 & MIN \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\
 & s.a. : \\
 & \sum_{i=1}^n x_{ij} = 1, \quad j = 1, 2, \dots, n \\
 & \sum_{j=1}^n a_{ij} x_{ij} \leq b_i, \quad i = 1, 2, \dots, n \\
 & x_{ij} = 0, 1
 \end{aligned}$$

En este modelo de asignación se puede asignar una persona a más de un trabajo, respetando obviamente las limitaciones en los recursos.

Algunas de las aplicaciones más relevantes son:

- Asignación de clientes a camiones (de reparto o recogida) de mercancías.
- Asignación de tareas a programadores.
- Asignación de trabajos a una red de ordenadores.

Problema de la Ordenación Lineal

Este problema consiste en determinar una permutación p de las filas y columnas de una matriz cuadrada dada, de manera que la suma de los elementos por encima de la diagonal sea máxima. Notar que la permutación p proporciona el orden tanto de las filas como de las columnas.

En términos económicos este problema es equivalente al de triangulación de matrices input-output, que puede describirse del siguiente modo: La economía de una región se divide en m sectores, se construye una matriz $m \times m$ donde la entrada a_{ij} denota la cantidad de mercancías (en valor monetario) que el sector i sirve al j en un año dado. El problema de triangulación consiste en permutar las filas y columnas de la matriz simultáneamente de manera que la suma de elementos por encima de la diagonal sea lo mayor posible. Una solución óptima presenta una ordenación de sectores de modo que los proveedores (sectores que producen para otros) van en primer lugar seguidos de los consumidores.

Este problema también puede enunciarse en términos de grafos, lo cual ayuda a formularlo del siguiente modo:

$$\begin{aligned}
 &MAX \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\
 &s.a. : \\
 &x_{ij} + x_{ji} = 1, \quad \forall i, j \in V, i \neq j \\
 &x_{ij} + x_{jk} + x_{ki} \leq 2, \quad 1 \leq i < j < k \leq n \\
 &x_{ij} = 0, 1
 \end{aligned}$$

donde $x_{ij} = 1$ representa que el sector (vértice) i precede al j en la ordenación dada por la solución.

1.2. Medidas de Calidad de un Algoritmo

Un buen algoritmo heurístico debe de tener las siguientes propiedades:

1. **Eficiente.** Un esfuerzo computacional realista para obtener la solución.

2. **Bueno.** La solución debe de estar, en promedio, cerca del óptimo.
3. **Robusto.** La probabilidad de obtener una mala solución (lejos del óptimo) debe ser baja.

Para medir la calidad de un heurístico existen diversos procedimientos, entre los que se encuentran los siguientes:

Comparación con la solución óptima

Aunque normalmente se recurre al algoritmo aproximado por no existir un método exacto para obtener el óptimo, o por ser éste computacionalmente muy costoso, en ocasiones puede que dispongamos de un procedimiento que proporcione el óptimo para un conjunto limitado de ejemplos (usualmente de tamaño reducido). Este conjunto de ejemplos puede servir para medir la calidad del método heurístico.

Normalmente se mide, para cada uno de los ejemplos, la desviación porcentual de la solución heurística frente a la óptima, calculando posteriormente el promedio de dichas desviaciones. Si llamamos c_h al coste de la solución del algoritmo heurístico y c_{opt} al coste de la solución óptima de un ejemplo dado, en un problema de minimización la desviación porcentual viene dada por la expresión: $\frac{c_h - c_{opt}}{c_{opt}} \cdot 100$.

Comparación con una cota

En ocasiones el óptimo del problema no está disponible ni siquiera para un conjunto limitado de ejemplos. Un método alternativo de evaluación consiste en comparar el valor de la solución que proporciona el heurístico con una cota del problema (inferior si es un problema de minimización y superior si es de maximización). Obviamente la bondad de esta medida dependerá de la bondad de la cota (cercanía de ésta al óptimo), por lo que, de alguna manera, tendremos que tener información de lo buena que es dicha cota. En caso contrario la comparación propuesta no tiene demasiado interés.

Comparación con un método exacto truncado

Un método enumerativo como el de Ramificación y Acotación explora una gran cantidad de soluciones, aunque sea únicamente una fracción del total, por lo que los problemas de grandes dimensiones pueden resultar computacionalmente inabordables con estos métodos. Sin embargo, podemos establecer un límite de iteraciones (o de tiempo) máximo de ejecución para el algoritmo exacto. También podemos saturar un nodo en un problema de maximización cuando su cota inferior sea menor o igual que la cota superior global más un cierto valor a (análogamente para el caso de minimizar). De esta forma se garantiza que el valor de la mejor solución proporcionada por el procedimiento no dista más de a del valor óptimo

del problema. En cualquier caso, la mejor solución encontrada con estos procedimientos truncados proporciona una cota con la que contrastar el heurístico.

Comparación con otros heurísticos

Este es uno de los métodos más empleados en problemas difíciles (NP-hard) sobre los que se ha trabajado durante tiempo y para los que se conocen algunos buenos heurísticos. Al igual que ocurre con la comparación con las cotas, la conclusión de dicha comparación está en función de la bondad del heurístico escogido.

Análisis del peor caso

Uno de los métodos que durante un tiempo tuvo bastante aceptación es analizar el comportamiento en el peor caso del algoritmo heurístico; esto es, considerar los ejemplos que sean más desfavorables para el algoritmo y acotar analíticamente la máxima desviación respecto del óptimo del problema. Lo mejor de este método es que acota el resultado del algoritmo para cualquier ejemplo; sin embargo, por esto mismo, los resultados no suelen ser representativos del comportamiento medio del algoritmo. Además, el análisis puede ser muy complicado para los heurísticos más sofisticados.

Aquellos algoritmos que, para cualquier ejemplo, producen soluciones cuyo coste no se aleja de un porcentaje ε del coste de la solución óptima, se llaman *Algoritmos ε -Aproximados*. Esto es; en un problema de minimización se tiene que cumplir para un $\varepsilon > 0$ que: $c_h \leq (1 + \varepsilon)c_{opt}$.

2. MÉTODOS CONSTRUCTIVOS EN EL PROBLEMA DEL VIAJANTE

Los métodos constructivos son procedimientos iterativos que, en cada paso, añaden un elemento hasta completar una solución. Usualmente son métodos deterministas y están basados en seleccionar, en cada iteración, el elemento con mejor evaluación. Estos métodos son muy dependientes del problema que resuelven, por lo que utilizaremos el problema del viajante para describirlos. En este capítulo se describen cuatro de los métodos más conocidos para el TSP tal y como aparecen en la revisión realizada por Jünger, Reinelt y Rinaldi (1995).

2.1. Heurísticos del Vecino más Próximo

Uno de los heurísticos más sencillos para el TSP es el llamado “del vecino más cercano”, que trata de construir un ciclo Hamiltoniano de bajo coste basándose en el vértice cercano a uno dado. Este algoritmo es debido a Rosenkrantz, Stearns y Lewis (1977) y su código, en una versión standard, es el siguiente:

Algoritmo del Vecino más Próximo*Inicialización*

Seleccionar un vértice j al azar.
 Hacer $t = j$ y $W = V \setminus \{j\}$.

Mientras ($W \neq \emptyset$)

Tomar $j \in W / c_{tj} = \min\{c_{ti} / i \in W\}$
 Conectar t a j
 Hacer $W = W \setminus \{j\}$ y $t = j$.

Este procedimiento realiza un número de operaciones de orden $O(n^2)$. Si seguimos la evolución del algoritmo al construir la solución de un ejemplo dado, veremos que comienza muy bien, seleccionando aristas de bajo coste. Sin embargo, al final del proceso probablemente quedarán vértices cuya conexión obligará a introducir aristas de coste elevado. Esto es lo que se conoce como *miopía* del procedimiento, ya que, en una iteración escoge la mejor opción disponible sin "ver" que esto puede obligar a realizar malas elecciones en iteraciones posteriores.

El algoritmo tal y como aparece puede ser programado en unas pocas líneas de código. Sin embargo una implementación directa será muy lenta al ejecutarse sobre ejemplos de gran tamaño (10000 vértices). Así pues, incluso para un heurístico tan sencillo como éste, es importante pensar en la eficiencia y velocidad de su código.

Para reducir la miopía del algoritmo y aumentar su velocidad se introduce el concepto de **subgrafo candidato**, junto con algunas modificaciones en la exploración. Un subgrafo candidato es un subgrafo del grafo completo con los n vértices y únicamente las aristas consideradas "atractiva" para aparecer en un ciclo Hamiltoniano de bajo coste. Una posibilidad es tomar, por ejemplo, el *subgrafo de los k vecinos más cercanos*; esto es, el subgrafo con los n vértices y para cada uno de ellos las aristas que lo unen con los k vértices más cercanos. Este subgrafo también será usado en otros procedimientos.

El algoritmo puede "mejorarse" en los siguientes aspectos:

- Para seleccionar el vértice j que se va a unir a t (y por lo tanto al tour parcial en construcción), en lugar de examinar todos los vértices, se examinan únicamente los adyacentes a t en el subgrafo candidato. Si todos ellos están ya en el tour parcial, entonces sí que se examinan todos los posibles.
- Cuando un vértice queda conectado (con grado 2) al tour en construcción, se eliminan del subgrafo candidato las aristas incidentes con él.
- Se especifica un número $s < k$ de modo que cuando un vértice que no está en el tour está conectado únicamente a s o menos aristas del subgrafo candidato se considera que se está quedando aislado. Por ello se inserta

inmediatamente en el tour. Como punto de inserción se toma el mejor de entre los k vértices más cercanos presentes en el tour.

Considerando el estudio empírico sobre las 30 ejemplos utilizados, la versión inicial del algoritmo presenta un porcentaje de desviación en promedio respecto del óptimo de 24.2%, mientras que la mejorada con $k = 10$ y $s = 4$ de 18.6%. La primera tiene un tiempo de ejecución medio de 15.3 segundos mientras que la segunda lo tiene de 0.3 segundos.

2.2. Heurísticos de Inserción

Otra aproximación intuitiva a la resolución del TSP consiste en comenzar construyendo ciclos que visiten únicamente unos cuantos vértices, para posteriormente extenderlos insertando los vértices restantes. En cada paso se inserta un nuevo vértice en el ciclo hasta obtener un ciclo Hamiltoniano. Este procedimiento es debido a los mismos autores que el anterior y su esquema es el siguiente:

Algoritmo de Inserción

Inicialización

Seleccionar un ciclo inicial (subtour) con k vértices.

Hacer $W = V \setminus \{\text{vértices seleccionados}\}$.

Mientras ($W \neq \emptyset$)

Tomar $j \in W$ de acuerdo con algún criterio preestablecido

Insertar j donde menos incremente la longitud del ciclo

Hacer $W = W \setminus \{j\}$.

Existen varias posibilidades para implementar el esquema anterior de acuerdo con el criterio de selección del vértice j de W a insertar en el ciclo. Se define la distancia de un vértice v al ciclo como el mínimo de las distancias de v a todos los vértices del ciclo:

$$d_{min}(v) = \min\{c_{iv} / i \in V \setminus W\}$$

Los criterios más utilizados son:

Inserción más cercana: Seleccionar el vértice j más cercano al ciclo.

$$d_{min}(j) = \min\{d_{min}(v) / v \in W\}$$

Inserción más lejana: Seleccionar el vértice j más lejano al ciclo.

$$d_{min}(j) = \max\{d_{min}(v) / v \in W\}$$

Inserción más barata: Seleccionar el vértice j que será insertado con el menor incremento del coste.

Inserción aleatoria: Seleccionar el vértice j al azar.

La Figura 2 muestra la diferencia entre estos criterios en un caso dado. El ciclo actual está formado por 4 vértices y hay que determinar el próximo a insertar. La inserción más cercana escogerá el vértice i , la más lejana el s y la más barata el k .

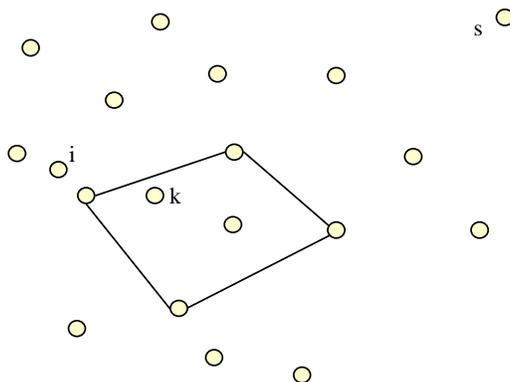


Figura 2. Selección del vértice a insertar

Todos los métodos presentan un tiempo de ejecución de $O(n^2)$ excepto la inserción más cercana que es de $O(n^2 \log n)$. Respecto al estudio empírico sobre los 30 grafos de la TSPLIB los porcentajes de desviación del óptimo son de 20%, 9.9%, 16.8% y 11.1% para el más cercano, más lejano, más barato y aleatorio, respectivamente, aunque el tiempo de ejecución del más barato es mucho mayor. Respecto al ciclo inicial se ha tomado $k = 3$ vértices y se comprueba experimentalmente que el procedimiento no depende mucho del ciclo inicial (aproximadamente un 6% de variación).

2.3. Heurísticos Basados en Árboles Generadores

Los heurísticos considerados anteriormente construyen un ciclo Hamiltoniano basándose únicamente en los costes de las aristas. Los heurísticos de este apartado se basan en el árbol generador de coste mínimo, lo que aporta una información adicional sobre la estructura del grafo. Comenzaremos por ver algunos conceptos de teoría de grafos.

- Un grafo es **conexo** si todo par de vértices está unido por un camino.
- Un **árbol** es un grafo conexo que no contiene ciclos. El número de aristas de un árbol es igual al número de vértices menos uno.
- Un **árbol generador** de un grafo $G = (V, A, C)$ es un árbol sobre todos los vértices y tiene, por tanto, $|V| - 1$ aristas de G .
- Un **árbol generador de mínimo peso** (o de coste mínimo) es aquel que de entre todos los árboles generadores de un grafo dado, presenta la menor suma de los costes de sus aristas.

- Un **acoplamiento** de un grafo $G = (V, A, C)$ es un subconjunto M del conjunto A de aristas cumpliendo que cada vértice del grafo es a lo sumo incidente con una arista de M .
- Un acoplamiento sobre un grafo $G = (V, A, C)$ es **perfecto** si es de cardinalidad máxima e igual a $\lfloor |V|/2 \rfloor$.

Las figuras siguientes ilustran los conceptos vistos sobre un grafo completo de 8 vértices. En la Figura 3 tenemos un árbol generador. Notar que contiene a todos los vértices y no hay ningún ciclo. La Figura 4 muestra un acoplamiento perfecto en el que podemos ver cómo cada vértice es incidente con una, y sólo una, de las aristas. Al ser un grafo de 8 vértices el número máximo de aristas en un acoplamiento es de 4, por lo que el de la figura es perfecto.

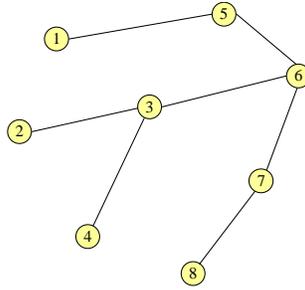


Figura 3. Árbol generador.

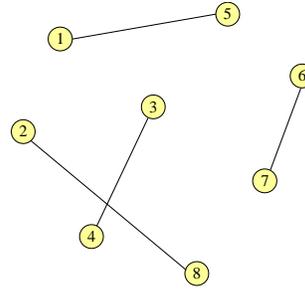


Figura 4. Acoplamiento Perfecto

El algoritmo debido a Prim (1957) obtiene un árbol generador de mínimo peso de un grafo G completo. El algoritmo comienza por definir el conjunto T de aristas del árbol (inicialmente vacío) y, el conjunto U de vértices del árbol (inicialmente formado por uno elegido al azar). En cada paso se calcula la arista de menor coste que une U con $V \setminus U$, añadiéndola a T y pasando su vértice adyacente de $V \setminus U$ a U . El procedimiento finaliza cuando U es igual a V ; en cuyo caso el conjunto T proporciona la solución.

Dado un ciclo $v_{i0}, v_{i1}, \dots, v_{ik}$ que pasa por todos los vértices de G (no necesariamente simple), el siguiente procedimiento obtiene un ciclo Hamiltoniano comenzando en v_{i0} y terminando en v_{ik} ($v_{i0} = v_{ik}$). En el caso de grafos con costes cumpliendo la desigualdad triangular (como es el caso de los grafos con distancias euclídeas), este procedimiento obtiene un ciclo de longitud menor o igual que la del ciclo de partida.

Algoritmo de Obtención de Tour

Inicialización

Seleccionar un ciclo inicial (subtour) con k vértices.

Hacer $T = \{v_{i0}\}$, $v = v_{i0}$ y $s = 1$.

Mientras ($|T| < |V|$)

Si v_{is} no en T hacer $T = T \cup \{v_{is}\}$, conectar v a v_{is} y hacer $v = v_{is}$

*Hacer $s = s + 1$
 Conectar v a v_{i0} y formar el ciclo Hamiltoniano.*

A partir de los elementos descritos se puede diseñar un algoritmo para obtener un ciclo Hamiltoniano. Basta con construir un árbol generador de mínimo peso (Figura 5), considerar el ciclo en el que todas las aristas del árbol son recorridas dos veces, cada vez en un sentido (Figura 6), y aplicar el *algoritmo de obtención de tour* a dicho ciclo (Figura 7). El ejemplo de las figuras mencionadas ilustra dicho procedimiento sobre un grafo completo con 10 vértices.

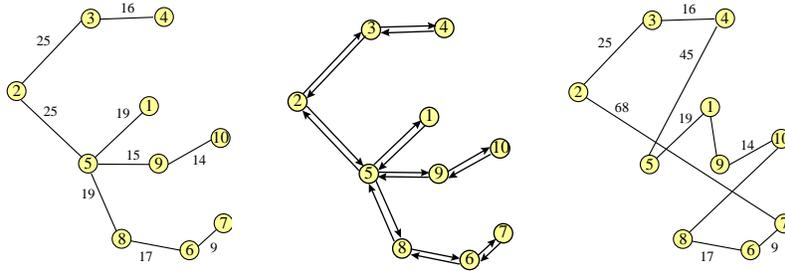


Figura 5. Árbol generador **Figura 6.** Duplicación **Figuras 7.** Tour

La Figura 5 muestra un árbol generador de mínimo peso e igual a 156. En la Figura 6 se han duplicado las aristas y se señala mediante flechas la dirección del ciclo resultante. Su coste obviamente será de $156 \times 2 = 312$. Aplicando el procedimiento de obtención de tour al ciclo de la Figura 6, se obtiene el ciclo Hamiltoniano de la Figura 7 con un coste de 258.

El proceso de duplicación de las aristas (recorrerlas todas en ambos sentidos) para obtener un tour aumenta en gran medida el coste de la solución. Podemos ver que es posible obtener un ciclo que pase por todos los vértices de G a partir de un árbol generador sin necesidad de duplicar todas las aristas. De hecho, basta con añadir aristas al árbol de modo que todos los vértices tengan grado par. El siguiente procedimiento, debido a Christofides (1976), calcula un acoplamiento perfecto de mínimo peso sobre los vértices de grado impar del árbol generador. Añadiendo al árbol las aristas del acoplamiento se obtiene un grafo con todos los vértices pares, y por lo tanto, un ciclo del grafo original, a partir del cual ya hemos visto cómo obtener un tour.

Algoritmo de Christofides

1. Calcular un Árbol Generador de Mínimo Peso
 2. Obtener el conjunto de vértices de grado impar en el Árbol.
 3. Obtener un Acoplamiento Perfecto de mínimo peso sobre dichos vértices.
 4. Añadir las aristas del Acoplamiento al Árbol.
 5. Aplicar el procedimiento de Obtención de Tour.
-

El cálculo del acoplamiento perfecto de coste mínimo sobre un grafo de k vértices se realiza en un tiempo $O(k^3)$ con el algoritmo de Edmonds (1965). Dado que un árbol generador de mínimo peso tiene como máximo $n - 1$ hojas (vértices de grado 1 en el árbol), el procedimiento de Christofides tendrá un tiempo de orden $O(n^3)$.

Propiedad: El algoritmo de Christofides sobre ejemplos cuya matriz de distancias cumple la desigualdad triangular produce una solución cuyo valor es como mucho 1.5 veces el valor óptimo:

$$c_h \leq \frac{3}{2}c_{OPT}$$

Es decir, es un algoritmo $1/2$ - aproximado sobre esta clase de ejemplos.

Prueba:

Sea $c(AGMP)$ el coste del árbol generador de mínimo peso y $c(A)$ el coste del acoplamiento perfecto calculado en el algoritmo de Christofides. Al añadir las aristas del acoplamiento al árbol se obtiene un ciclo (con posibles repeticiones) cuyo coste es la suma de ambos costes. Dado que la matriz de distancias cumple la desigualdad triangular, al aplicar el algoritmo de obtención de tour el coste puede reducirse eventualmente. Por ello el coste de la solución obtenida, c_H , cumple:

$$c_H \leq c(AGMP) + c(A) \tag{1}$$

Un árbol generador de mínimo peso, por construcción, tiene un coste menor que cualquier ciclo Hamiltoniano y, por lo tanto, que el ciclo Hamiltoniano de coste mínimo (tour óptimo). Para probarlo basta con considerar un ciclo Hamiltoniano y quitarle una arista, con lo que se obtiene un árbol generador. El árbol generador de mínimo peso tiene, obviamente, un coste menor que dicho árbol generador y, por lo tanto, que el ciclo Hamiltoniano. Luego:

$$c(AGMP) \leq c_{OPT} \tag{2}$$

El acoplamiento del paso 3 del algoritmo de Christofides tiene un coste menor o igual que la mitad de la longitud de un tour óptimo en un grafo con costes cumpliendo la desigualdad triangular. Para probarlo consideremos un tour óptimo y llamemos S al conjunto de vértices de grado impar en el árbol. El algoritmo calcula un acoplamiento perfecto de coste mínimo sobre los vértices de S . La Figura 8 muestra un ciclo Hamiltoniano óptimo y los vértices de S en oscuro. Además aparecen dos acoplamientos perfectos sobre S , A_1 (trazo continuo) y A_2 (trazo discontinuo), en donde cada vértice está acoplado al más próximo en el tour óptimo.

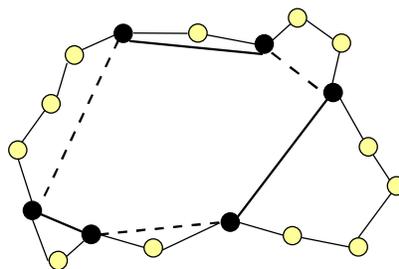


Figura 8. Dos acoplamientos sobre S

Como se cumple la desigualdad triangular, se tiene que $c(A_1) + c(A_2) \leq c_{OPT}$. Es evidente que por ser el de coste mínimo $c(A) \leq c(A_1)$ y $c(A) \leq c(A_2)$, de donde:

$$2c(A) \leq c_{OPT} \tag{3}$$

De (1), (2) y (3) se concluye el resultado. ■

Las figuras siguientes ilustran el método de Christofides sobre el mismo ejemplo de las figuras 5, 6 y 7. En la Figura 9 aparecen oscurecidos los vértices de grado impar en el árbol generador, y en trazo discontinuo las aristas del acoplamiento perfecto de coste mínimo sobre tales vértices. Al añadirlas se obtiene un tour de coste 199. La Figura 10 muestra el ciclo Hamiltoniano que se obtiene al aplicarle el procedimiento de obtención del ciclo al tour de la figura anterior. La solución tiene un coste de 203, mientras que la obtenida con el procedimiento de duplicar aristas (Figura 7) tenía un coste de 258.

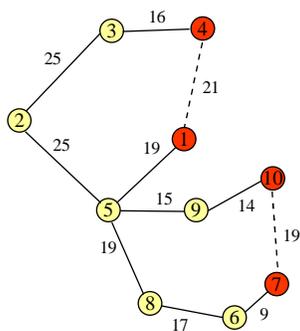


Figura 9. Acoplamiento Perfecto

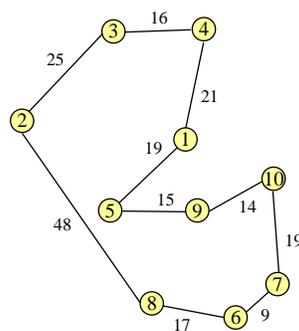


Figura 10. Ciclo Hamiltoniano

El procedimiento del árbol generador y posterior duplicación de aristas obtiene, sobre el conjunto de ejemplos considerados de la TSPLIB, una desviación del óptimo de 38%, mientras que el heurístico de Christofides de 19.5%.

2.4. Heurísticos Basados en Ahorros

Los métodos de esta sección son debidos a Clarke y Wright (1964) y fueron propuestos inicialmente para problemas de rutas de vehículos. Veamos una adaptación de estos procedimientos al problema del viajante.

El algoritmo siguiente se basa en combinar sucesivamente subtours hasta obtener un ciclo Hamiltoniano. Los subtours considerados tienen un vértice común llamado *base*. El procedimiento de unión de subtours se basa en eliminar las aristas que conectan dos vértices de diferentes subtours con el vértice base, uniendo posteriormente los vértices entre sí. Llamamos *ahorro* a la diferencia del coste entre las aristas eliminadas y la añadida.

Algoritmo de Ahorros

Inicialización

Tomar un vértice $z \in V$ como base.

Establecer los $n-1$ subtours $[(z, v), (v, z)], \forall v \in V \setminus \{z\}$.

Mientras (Queden dos o más subtours)

Para cada par de subtours calcular el ahorro de unirlos al eliminar en cada uno una de las aristas que lo une con z y conectar los dos vértices asociados. Unir los dos subtours que produzcan un ahorro mayor

En las figuras 11 y 12 se ilustra una iteración del procedimiento. Podemos ver cómo se combinan dos subtours eliminando las aristas de los vértices i y j al vértice base z , e insertando la arista (i, j) .

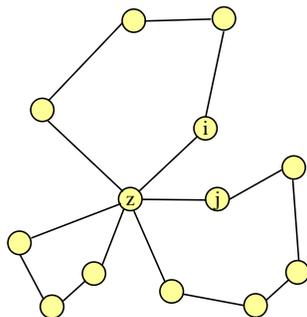


Figura 11. Subtours iniciales

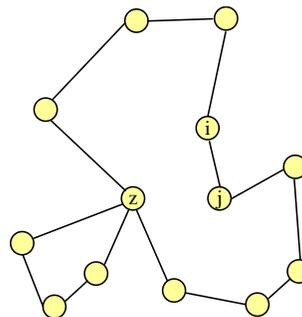


Figura 12. Subtours finales

En la implementación del algoritmo se tiene que mantener una lista con las combinaciones posibles. El punto clave de la implementación es la actualización de esta lista. Sin embargo, al unir dos subtours únicamente se ven afectados aquellos en los que su "mejor conexión" pertenece a alguno de los dos subtours recién unidos. Luego basta con actualizar éstos en cada iteración sin necesidad de actualizarlos todos cada vez que se realiza una unión.

Al igual que en otros heurísticos, podemos utilizar el subgrafo candidato (en el que están todos los vértices y sólo las aristas consideradas "atractivas")

para acelerar los cálculos. Así, al actualizar la lista de las mejores conexiones únicamente se consideran aristas del subgrafo candidato.

El método presenta un tiempo de ejecución de $O(n^3)$. Respecto al estudio empírico sobre los 30 ejemplos de la TSPLIB los porcentajes de desviación respecto del óptimo son de 9.8% para el método original y 9.6% para el mejorado con el uso del subgrafo candidato. Además, el tiempo de ejecución es mucho menor para este último.

La siguiente tabla recoge los resultados del estudio comparativo sobre los cuatro algoritmos descritos, con los 30 ejemplos de la TSPLIB considerados:

Heurístico	Desviación del Óptimo	T. Ejecución (pr2392)
Vecino más Próximo	18.6%	0.3
Inserción más Lejana	9.9%	35.4
Christofides	19.5%	0.7
Ahorros	9.6%	5.07

Todos los métodos están basados en cálculos relativamente sencillos y han sido implementados eficientemente, por lo que los tiempos de computación son muy parecidos entre sí e inferiores a 1 segundo en promedio. Por ello, para distinguir entre todos, en la tabla se muestra el tiempo en segundos sobre el ejemplo de mayor tamaño considerado (pr2392) de casi 2400 vértices.

A la vista de los resultados podemos concluir que tanto el método de los ahorros como el de inserción basado en el elemento más lejano son los que mejores resultados obtienen, aunque presentan un tiempo de computación mayor que los otros dos.

3. BÚSQUEDA LOCAL EN EL PROBLEMA DEL VIAJANTE

En general, las soluciones obtenidas con los métodos constructivos suelen ser de una calidad moderada. En este apartado vamos a estudiar diversos algoritmos basados en la búsqueda local para mejorarlas. Al igual que ocurría con los métodos descritos en la sección anterior, estos algoritmos son muy dependientes del problema que resuelven, por lo que al igual que allí, utilizaremos el TSP para describirlos. Específicamente, consideraremos tres de los métodos más utilizados, tal y como aparecen descritos en Jünger, Reinelt y Rinaldi (1995). Comenzaremos por definir y explicar algunos de los conceptos genéricos de estos métodos.

Los procedimientos de búsqueda local, también llamados de mejora, se basan en explorar el entorno o vecindad de una solución. Utilizan una operación básica llamada movimiento que, aplicada sobre los diferentes

elementos de una solución, proporciona las soluciones de su entorno. Formalmente:

Definición: Sea X el conjunto de soluciones del problema combinatorio. Cada solución x tiene un conjunto de soluciones asociadas $N(x) \subseteq X$, que denominaremos entorno de x .

Definición: Dada una solución x , cada solución de su entorno, $x' \in N(x)$, puede obtenerse directamente a partir de x mediante una operación llamada movimiento.

Un procedimiento de búsqueda local parte de una solución inicial x_0 , calcula su entorno $N(x_0)$ y escoge una nueva solución x_1 en él. Dicho de otro modo, realiza el movimiento m_1 que aplicado a x_0 da como resultado x_1 . Este proceso se aplica reiteradamente, describiendo una trayectoria en el espacio de soluciones.

Un procedimiento de búsqueda local queda determinado al especificar un entorno y el criterio de selección de una solución dentro del entorno. La definición de entorno/movimiento, depende en gran medida de la estructura del problema a resolver, así como de la función objetivo. También se pueden definir diferentes criterios para seleccionar una nueva solución del entorno. Uno de los criterios más simples consiste en tomar la solución con mejor evaluación de la función objetivo, siempre que la nueva solución sea mejor que la actual. Este criterio, conocido como greedy, permite ir mejorando la solución actual mientras se pueda. El algoritmo se detiene cuando la solución no puede ser mejorada. A la solución encontrada se le denomina *óptimo local* respecto al entorno definido.

El óptimo local alcanzado no puede mejorarse mediante el movimiento definido. Sin embargo, el método empleado no permite garantizar, de ningún modo, que sea el óptimo global del problema. Más aún, dada la "miopía" de la búsqueda local, es de esperar que en problemas de cierta dificultad, en general no lo sea.

La Figura 13 muestra el espacio de soluciones de un problema de maximización de dos dimensiones donde la altura del gráfico mide el valor de la función objetivo. Se considera un procedimiento de búsqueda local greedy iniciado a partir de una solución x_0 con valor 5 y que realiza 8 movimientos de mejora hasta alcanzar la solución x_8 con valor 13. La figura muestra cómo x_8 es un óptimo local y cualquier movimiento que se le aplique proporcionará una solución con peor valor. Podemos ver cómo el óptimo global del problema, con un valor de 15, no puede ser alcanzado desde x_8 , a menos que permitamos realizar movimientos que empeoren el valor de las soluciones y sepamos dirigir correctamente la búsqueda.

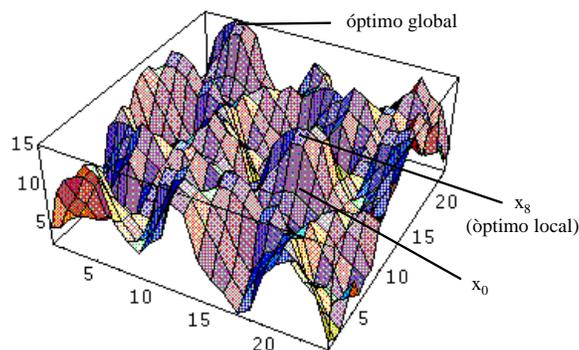


Figura 13. Óptimo local y global

Esta limitación de la estrategia greedy es el punto de partida de los procedimientos meta-heurísticos basados en búsqueda local: evitar el quedar atrapados en un óptimo local lejano del global. Para lo cual, como hemos visto, se hace preciso el utilizar movimientos que empeoren la función objetivo. Sin embargo esto plantea dos problemas. El primero es que al permitir movimientos de mejora y de no mejora, el procedimiento se puede ciclar, revisitando soluciones ya vistas, por lo que habría que introducir un mecanismo que lo impida. El segundo es que hay que establecer un criterio de parada ya que un procedimiento de dichas características podría iterar indefinidamente. Los procedimientos meta-heurísticos incorporan mecanismos sofisticados para solucionar eficientemente ambas cuestiones así como para tratar, en la medida de lo posible, de dirigir la búsqueda de forma inteligente.

Pese a la miopía de los métodos de búsqueda local simples, suelen ser muy rápidos y proporcionan soluciones que, en promedio, están relativamente cerca del óptimo global del problema. Además, dichos métodos suelen ser el punto de partida en el diseño de algoritmos meta-heurísticos más complejos. En este apartado vamos a estudiar algunos métodos heurísticos de búsqueda local para el problema del viajante.

3.1. Procedimientos de 2 intercambio

Este procedimiento está basado en la siguiente observación para grafos con distancias euclídeas (o en general con costes cumpliendo la desigualdad triangular). Si un ciclo Hamiltoniano se cruza a si mismo, puede ser fácilmente acortado, basta con eliminar las dos aristas que se cruzan y reconectar los dos caminos resultantes mediante aristas que no se corten. El ciclo final es más corto que el inicial.

Un *movimiento 2-opt* consiste en eliminar dos aristas y reconectar los dos caminos resultantes de una manera diferente para obtener un nuevo ciclo.

Las figuras 14 y 15 ilustran este movimiento en el que las aristas (i, j) y (l, k) son reemplazadas por (l, j) y (i, k) . Notar que sólo hay una manera de reconectar los dos caminos formando un único tour.

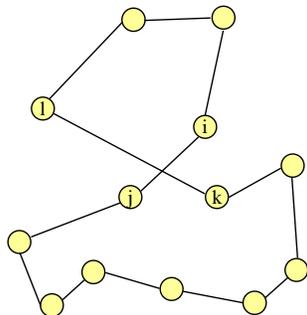


Figura 14. Solución original

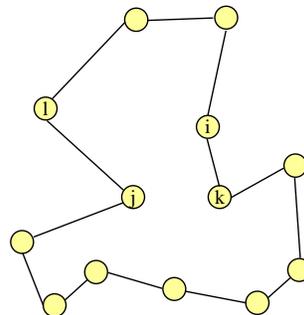


Figura 15. Solución mejorada

El siguiente código recoge el algoritmo heurístico de mejora 2-óptimo. Consiste en examinar todos los vértices, realizando, en cada paso, el mejor movimiento 2-opt asociado a cada vértice.

Algoritmo 2-óptimo

Inicialización

Considerar un ciclo Hamiltoniano inicial
move=1

Mientras (move=1)

move=0. Etiquetar todos los vértices como no explorados.

Mientras(Queden vértices por explorar)

Seleccionar un vértice i no explorado.

Examinar todos los movimientos 2-opt que incluyan la arista de i a su sucesor en el ciclo. Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer

move = 1.

En otro caso etiquetar i como explorado.

La variable *move* vale 0 si no se ha realizado ningún movimiento al examinar todos los vértices, y 1 en otro caso. El algoritmo finaliza cuando *move* = 0, con lo que queda garantizado que no existe ningún movimiento 2-opt que pueda mejorar la solución.

El orden en el que el algoritmo examina los nodos incide de manera notable en su funcionamiento. En una implementación sencilla podemos considerar el orden natural $1, 2, \dots, n$. Sin embargo, es fácil comprobar que cuando se realiza un movimiento hay muchas posibilidades de encontrar movimientos de mejora asociados a los vértices que han intervenido en el movimiento recién realizado. Por ello, una implementación más eficiente consiste en considerar una lista de vértices candidatos a examinar. El orden inicial es el de los vértices en el ciclo comenzando por uno arbitrario y en cada iteración se examina el primero de la lista. Cada vez que se examina

un vértice i , éste se coloca al final de la lista y, los vértices involucrados en el movimiento (vértices j , k y l de la Figura 15) se insertan en primer lugar.

Dado que el proceso de examinar todos los movimientos 2-opt asociados a cada vértice es muy costoso computacionalmente, se pueden introducir las siguientes mejoras para acelerar el algoritmo:

- Exigir que al menos una de las dos aristas añadidas en cada movimiento, para formar la nueva solución, pertenezca al subgrafo candidato.
- Observando el funcionamiento del algoritmo se puede ver que en las primeras iteraciones la función objetivo decrece substancialmente, mientras que en las últimas apenas se modifica. De hecho la última únicamente verifica que es un óptimo local al no realizar ningún movimiento. Por ello, si interrumpimos el algoritmo antes de su finalización, ahorraremos bastante tiempo y no perderemos mucha calidad.

Es evidente que ambas mejoras reducen el tiempo de computación a expensas de perder la garantía de que la solución final es un óptimo local. Así pues, dependiendo del tamaño del ejemplo a resolver, así como de lo crítico que sea el tiempo de ejecución, se deben implementar o no.

El comprobar si existe, o no, un movimiento 2-opt de mejora utiliza un tiempo de orden $O(n^2)$, ya que hay que examinar todos los pares de aristas en el ciclo. Podemos encontrar clases de problemas para los que el tiempo de ejecución del algoritmo no está acotado polinómicamente. Respecto al estudio empírico sobre los ejemplos de la TSPLIB considerados, partiendo de la solución del algoritmo del vecino más próximo, el promedio de desviación del óptimo es del 8.3%.

3.2. Procedimientos de k - intercambio

Para introducir mayor flexibilidad al modificar un ciclo Hamiltoniano, podemos considerar el dividirlo en k partes, en lugar de dos, y combinar los caminos resultantes de la mejor manera posible. Llamamos movimiento k -opt a tal modificación.

Es evidente que al aumentar k aumentará el tamaño del entorno y el número de posibilidades a examinar en el movimiento, tanto por las posibles combinaciones para eliminar las aristas del ciclo, como por la reconstrucción posterior. El número de combinaciones para eliminar k aristas en un ciclo viene dado por el número $\binom{n}{k}$.

Examinar todos los movimientos k -opt de una solución lleva un tiempo del orden de $O(n^k)$ por lo que, para valores altos de k , sólo es aplicable a ejemplos de tamaño pequeño.

En este apartado vamos a estudiar el caso de $k = 3$ y además impondremos ciertas restricciones para reducir el entorno y poder realizar los cálculos en un tiempo razonable.

En un movimiento 3-opt, una vez eliminadas las tres aristas hay ocho maneras de conectar los tres caminos resultantes para formar un ciclo. Las figuras siguientes ilustran algunos de los ocho casos. La Figura 16 muestra el ciclo inicial en el que se encuentran las aristas (a, b) , (c, d) y (e, f) por las que se dividirá éste. La Figura 17 utiliza la propia arista (e, f) para reconstruir el ciclo, por lo que, este caso equivale a realizar un movimiento 2-opt sobre las aristas (a, b) y (c, d) . Análogamente podemos considerar los otros dos casos en los que se mantiene una de las tres aristas en el ciclo y se realiza un movimiento 2-opt sobre las restantes. Las figuras 18 y 19 muestran un movimiento 3-opt “puro” en el que desaparecen del ciclo las tres aristas seleccionadas.

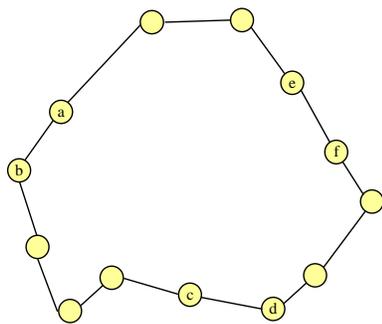


Figura 16. Ciclo inicial

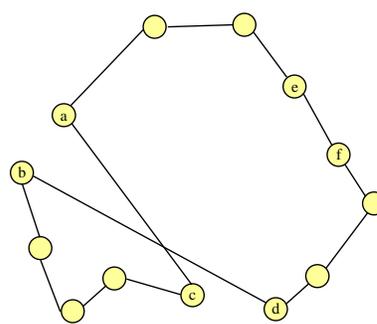


Figura 17. Movimiento 2-opt

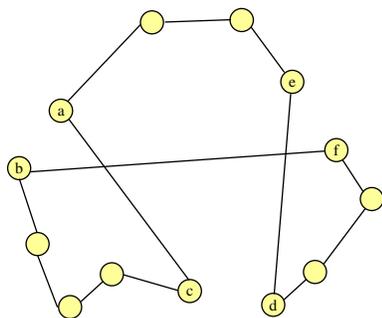


Figura 18. Movimiento 3-opt

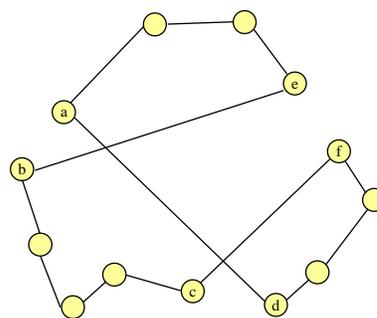


Figura 19. Movimiento 3-opt

A diferencia de los movimientos 2-opt, el reconstruir el ciclo una vez eliminadas las tres aristas es muy costoso. Notar que la dirección en el ciclo puede cambiar en todos los caminos menos en el más largo por lo que hay que realizar varias actualizaciones. Además, el mero hecho de examinar todas las posibilidades representa un esfuerzo computacional enorme (más de 1 hora en los problemas considerados). Por ello, se consideran únicamente algunos de los movimientos 3-opt. En concreto se define para cada vértice i un conjunto de vértices $N(i)$ de modo que al examinar los movimientos

3-opt asociados a una arista (i, j) , únicamente se consideran aquellos en los que las otras dos aristas tengan al menos uno de los vértices en $N(i)$. Una posibilidad para definir $N(i)$ consiste en considerar los vértices adyacentes a i en el subgrafo candidato.

Algoritmo 3-óptimo restringido

Inicialización

Considerar un ciclo Hamiltoniano inicial

Para cada vértice i definir un conjunto de vértices $N(i)$

move = 1

Mientras (move = 1)

move=0

Etiquetar todos los vértices como no explorados.

Mientras(Queden vértices por explorar)

Seleccionar un vértice i no explorado.

Examinar todos los movimientos 3-opt que eliminen 3 aristas teniendo cada una, al menos un vértice en $N(i)$.

Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer move = 1. En otro caso etiquetar i como explorado.

El promedio de las desviaciones al óptimo sobre el conjunto test considerado (TSPLIB) es de 3.8% con esta versión restringida, partiendo de la solución del vecino más próximo, y de 3.9% partiendo de una solución al azar.

3.3. Algoritmo de Lin y Kernighan

Como vimos en la introducción a los métodos de mejora (Figura 13), el problema de los algoritmos de búsqueda local es que suelen quedarse atrapados en un óptimo local. Vimos que para alcanzar una solución mejor a partir de un óptimo local habría que comenzar por realizar movimientos que empeoren el valor de la solución, lo que conduciría a un esquema de búsqueda mucho más complejo, al utilizar el algoritmo tanto movimientos de mejora como de no mejora.

El algoritmo de Lin y Kernighan parte de este hecho y propone un movimiento compuesto, en donde cada una de las partes consta de un movimiento que no mejora necesariamente pero el movimiento compuesto sí es de mejora. De esta forma es como si se realizaran varios movimientos simples consecutivos en donde algunos empeoran y otros mejoran el valor de la solución, pero no se pierde el control sobre el proceso de búsqueda ya que el movimiento completo sí que mejora. Además, combina diferentes movimientos simples, lo cual es una estrategia que ha producido muy buenos resultados en los algoritmos de búsqueda local. En concreto la estrategia denominada “cadenas de eyección” se basa en encadenar movimientos y ha dado muy buenos resultados en el contexto de la Búsqueda Tabú.

Se pueden considerar muchas variantes para este algoritmo. En este apartado consideraremos una versión sencilla basada en realizar dos movimientos 2-opt seguidos de un movimiento de inserción. Ilustraremos el procedimiento mediante el ejemplo desarrollado en las figuras siguientes sobre un grafo de 12 vértices. Consideramos el ciclo Hamiltoniano inicial dado por el orden natural de los vértices y lo representamos tal y como aparece en la Figura 20.

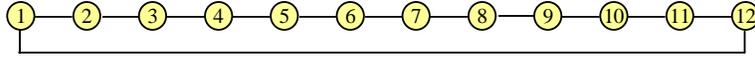


Figura 20

Paso 1: Realiza un movimiento 2-opt reemplazando las aristas $(12,1)$ y $(5,6)$ por $(12,5)$ y $(1,6)$. El resultado se muestra en la Figura 21.

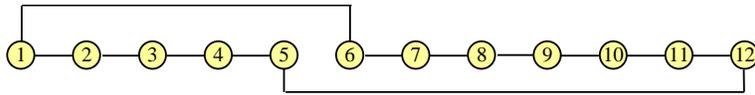


Figura 21

Paso 2: Realiza un movimiento 2-opt reemplazando las aristas $(6,1)$ y $(3,4)$ por $(6,3)$ y $(1,4)$. Ver Figura 22.

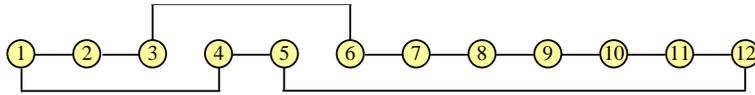


Figura 22

Paso 3: Realiza un movimiento de inserción, insertando el vértice 9 entre el 1 y el 4 (Figura 23).

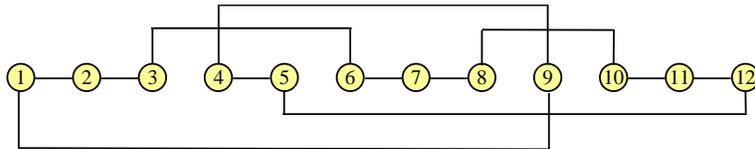


Figura 23

El algoritmo funciona de igual modo que el 2-óptimo o el 3-óptimo: parte de un ciclo Hamiltoniano inicial y realiza movimientos de mejora hasta alcanzar un óptimo local.

Algoritmo de Lin y Kernighan*Inicialización**Considerar un ciclo Hamiltoniano inicial**move = 1**Mientras (move = 1)**move=0**Etiquetar todos los vértices como no explorados.**Mientras (Queden vértices por explorar)**Seleccionar un vértice i no explorado.**Examinar todos los movimientos (2-opt, 2-opt, inserción) que incluyan la arista de i a su sucesor en el ciclo.**Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer $move = 1$. En otro caso etiquetar i como explorado.*

Dado el gran número de combinaciones posibles para escoger los movimientos, es evidente que una implementación eficiente del algoritmo tendrá que restringir el conjunto de movimientos a examinar en cada paso. De entre las numerosas variantes estudiadas para reducir los tiempos de computación del algoritmo y aumentar la eficiencia del proceso, destacamos las dos siguientes:

- Utilizar el subgrafo candidato en el que únicamente figuran las aristas relativas a los 6 vecinos más cercanos para cada vértice. Se admiten movimientos compuestos de hasta 15 movimientos simples todos del tipo 2-opt o inserción. Para el primer movimiento simple únicamente se examinan 3 candidatos.
- El subgrafo candidato está formado por las aristas relativas a los 8 vecinos más cercanos. Se admiten hasta 15 movimientos simples del tipo 2-opt o inserción por cada movimiento completo. En los 3 primeros movimientos simples únicamente se examinan 2 aristas.

Respecto al estudio computacional sobre los ejemplos de la TSPLIB considerados, la desviación del óptimo partiendo de la solución del heurístico del vecino más próximo es de 1.9% para la primera variante y de 1.5% para la segunda.

La siguiente tabla recoge el promedio de las desviaciones del óptimo y tiempos de ejecución de los 4 algoritmos de mejora considerados sobre los 30 ejemplos de la TSPLIB. Todos ellos toman como solución inicial la obtenida con el método del vecino más próximo.

Heurístico	Desviación del Óptimo	T. Ejecución (pr2392)
2-óptimo	8.3 %	0.25
3-óptimo	3.8 %	85.1
Lin y Kernighan 1	1.9 %	27.7
Lin y Kernighan 2	1.5 %	74.3

Respecto a los tiempos de ejecución, podemos ver cómo el pasar de una exploración 2-opt a 3-opt aumenta considerablemente el tiempo, incluso en la versión restringida planteada. También es de señalar cómo aumenta el tiempo, en casi un factor de 3, de una versión a otra del algoritmo de Lin y Kernighan, mientras que el porcentaje de desviación respecto del óptimo únicamente gana un 0.4 %.

A la vista de los resultados, parece más interesante utilizar movimientos compuestos, que permitan controlar movimientos simples de no mejora, que utilizar movimientos k -óptimos con valores altos de k que, por su complejidad, consumen mucho tiempo y, sin embargo, no llegan a tan buenos resultados.

4. MÉTODOS COMBINADOS

En los apartados anteriores hemos visto los métodos constructivos que obtienen una solución del problema y los métodos de mejora que, a partir de una solución inicial, tratan de obtener nuevas soluciones con mejor valor. Es evidente que ambos métodos pueden combinarse, tomando los segundos como solución inicial la obtenida con los primeros. En este apartado estudiaremos algunas variantes y mejoras sobre tal esquema.

Como hemos visto, una de las limitaciones más importantes de los métodos heurísticos es la denominada miopía provocada por seleccionar, en cada paso, la mejor opción. Resulta muy ilustrativo que en los métodos de inserción, se obtengan mejores resultados al elegir al azar el vértice a insertar, que al tomar el elemento más cercano (11.1 % frente a 20 % de promedio de desviación del óptimo). Sin embargo, es evidente que el tomar una opción al azar como norma puede conducirnos a cualquier resultado, por lo que parece más adecuado recurrir a algún procedimiento sistemático que compendie la evaluación con el azar. Veamos dos de los más utilizados.

4.1. Procedimientos Aleatorizados

Una modificación en el algoritmo de construcción consiste en sustituir una elección greedy por una elección al azar de entre un conjunto de buenos candidatos. Así, en cada paso del procedimiento, se evalúan todos los elementos que pueden ser añadidos y se selecciona un subconjunto con los

mejores. La elección se realiza al azar sobre ese subconjunto de buenos candidatos.

Existen varias maneras de establecer el subconjunto de mejores candidatos. En un problema de maximización, en donde cuanto mayor es la evaluación de una opción más “atractiva” resulta, podemos destacar:

- Establecer un número fijo k para el subconjunto de mejores candidatos e incluir los k mejores.
- Establecer un valor umbral e incluir en el conjunto todos los elementos cuya evaluación esté por encima de dicho valor. Incluir siempre el mejor de todos.
- Establecer un porcentaje respecto del mejor, e incluir en el subconjunto todos aquellos elementos cuya evaluación difiere, de la del mejor en porcentaje, en una cantidad menor o igual que la establecida

En todos los casos la elección final se realiza al azar de entre los preseleccionados.

Una estrategia alternativa a la anterior consiste en considerar las evaluaciones como pesos y utilizar un método probabilístico para seleccionar una opción. Así, si v_1, v_2, \dots, v_k son los posibles elementos a añadir en un paso del algoritmo, se calculan sus evaluaciones e_1, e_2, \dots, e_k , y se les asigna un intervalo del siguiente modo:

Elemento	Evaluación	Intervalo
v_1	e_1	$[0, e_1[$
v_2	e_2	$[e_1, e_1 + e_2[$
...
v_k	e_k	$\left[\sum_{i=1}^{k-1} e_i, \sum_{i=1}^k e_i \right]$

Se genera un número a al azar entre 0 y $\sum_{i=1}^k e_i = 1$, y se selecciona el elemento correspondiente al intervalo que contiene al valor a .

Al algoritmo constructivo modificado, tanto con la opción primera como con la segunda, lo llamaremos algoritmo constructivo aleatorizado. Este algoritmo puede que produzca una solución de peor calidad que la del algoritmo original. Sin embargo, dado que el proceso no es completamente determinista, cada vez que lo realicemos sobre un mismo ejemplo obtendremos resultados diferentes. Esto permite definir un proceso iterativo consistente en ejecutar un número prefijado de veces (*MAXITER*) el algoritmo y quedarnos con la mejor de las soluciones obtenidas. Obviamente, cada una de dichas soluciones puede mejorarse con un algoritmo de búsqueda local. El siguiente procedimiento incorpora el algoritmo de mejora a dicho esquema.

Algoritmo combinado aleatorizado*Inicialización*

*Obtener una solución con el algoritmo constructivo aleatorizado.
Sea c^* el coste de dicha solución.
Hacer $i = 0$*

Mientras ($i < MAX_ITER$)

*Obtener una solución $x(i)$ con el algoritmo constructivo aleatorizado.
Aplicar el algoritmo de búsqueda local a $x(i)$.
Sea $x^*(i)$ la solución obtenida y $S^*(i)$ su valor.
Si ($S^*(i)$ mejora a c^*)
Hacer $c^* = S^*(i)$ y guardar la solución actual
 $i = i + 1$*

En cada iteración el algoritmo construye una solución (fase 1) y después trata de mejorarla (fase 2). Así, en la iteración i , el algoritmo construye la solución $x(i)$ con valor $S(i)$ y posteriormente la mejora obteniendo $x^*(i)$ con valor $S^*(i)$. Notar que $x^*(i)$ puede ser igual a $x(i)$ si el algoritmo de la segunda fase no encuentra ningún movimiento que mejore la solución.

Después de un determinado número de iteraciones es posible estimar el porcentaje de mejora obtenido por el algoritmo de la fase 2 y utilizar esta información para aumentar la eficiencia del procedimiento. En concreto, al construir una solución se examina su valor y se puede considerar que la fase 2 la mejoraría en un porcentaje similar al observado en promedio. Si el valor resultante queda alejado del valor de la mejor solución encontrada hasta el momento, podemos descartar la solución actual y no realizar la fase 2 del algoritmo, con el consiguiente ahorro computacional.

Dadas las numerosas variantes posibles sobre el esquema propuesto, no las incluiremos en la comparativa realizada sobre los 30 ejemplos de la TSPLIB. Únicamente citar que en general los resultados son de mejor calidad que los obtenidos por el heurístico de Lin y Kernighan (alrededor de un 0.5%) aunque a expensas de emplear tiempos de computación bastante mayores (del orden de algunos minutos).

Dado el interés por resolver problemas enteros en general, y en particular el TSP, se han propuesto numerosas mejoras y nuevas estrategias sobre el esquema anterior. Una de las más utilizadas es la denominada técnica de Multi-Arranque que abordamos en la próxima sección.

4.2. Métodos Multi-Arranque

Los métodos Multi-Arranque (también llamados Multi-Start o Re-Start) generalizan el esquema anterior. Tienen dos fases: la primera en la que se genera una solución y la segunda en la que la solución es típicamente, pero no necesariamente, mejorada. Cada iteración global produce una solución, usualmente un óptimo local, y la mejor de todas es la salida del algoritmo.

Algoritmo Multi-Arranque

*Mientras (Condición de parada)***Fase de Generación***Construir una solución.***Fase de Búsqueda***Aplicar un método de búsqueda para mejorar la solución construida***Actualización***Si la solución obtenida mejora a la mejor almacenada, actualizarla.*

Dada su sencillez de aplicación, estos métodos han sido muy utilizados para resolver gran cantidad de problemas. En el contexto de la programación no lineal sin restricciones, podemos encontrar numerosos trabajos tanto teóricos como aplicados. Rinnooy Kan y Timmer (1989) estudian la generación de soluciones aleatorias (métodos Monte Carlo) y condiciones de convergencia. Las primeras aplicaciones en el ámbito de la optimización combinatoria consistían en métodos sencillos de construcción, completa o parcialmente aleatorios, y su posterior mejora con un método de búsqueda local. Sin embargo el mismo esquema permite sofisticar el procedimiento basándolo en unas construcciones y/o mejoras más complejas. Numerosas referencias y aplicaciones se pueden encontrar en Martí(2000).

Uno de los artículos que contiene las ideas en que se basa el método Tabu Search (Glover, 1977) también incluye aplicaciones de estas ideas para los métodos de multi-start. Básicamente se trata de almacenar la información relativa a soluciones ya generadas y utilizarla para la construcción de nuevas soluciones. Las estructuras de memoria reciente y frecuente se introducen en este contexto. Diferentes aplicaciones se pueden encontrar en Rochat y Taillard (1995) y Lokketangen y Glover (1996).

Utilizando como procedimiento de mejora un algoritmo genético, Ulder y otros (1990) proponen un método para obtener buenas soluciones al problema del agente viajero. Los autores muestran cómo el uso de las técnicas de re-starting aumenta la eficiencia del algoritmo comparándolo con otras versiones de heurísticos genéticos sin re-starting.

Un problema abierto actualmente para diseñar un buen procedimiento de búsqueda basada en multi- arranque es si es preferible implementar un procedimiento de mejora sencillo que permita realizar un gran número de iteraciones globales o, alternativamente, aplicar una rutina más compleja que mejore significativamente unas pocas soluciones generadas. Un procedimiento sencillo depende fuertemente de la solución inicial pero un método más elaborado consume mucho más tiempo de computación y, por tanto, puede ser aplicado pocas veces, reduciendo el muestreo del espacio de soluciones.

Una de las variantes más populares de estos métodos se denomina GRASP (Feo y Resende, 1995) y está obteniendo resultados excelentes en la resolución de numerosos problemas combinatorios. La próxima sección describe en detalle estos métodos.

4.3. GRASP (Greedy Randomized Adaptive Search Procedures)

Los métodos GRASP fueron desarrollados al final de la década de los 80 con el objetivo inicial de resolver problemas de cubrimientos de conjuntos (Feo y Resende, 1989). El término GRASP fue introducido por Feo y Resende (1995) como una nueva técnica metaheurística de propósito general.

GRASP es un procedimiento de multi-arranque en donde cada paso consiste en una fase de construcción y una de mejora. En la fase de construcción se aplica un procedimiento heurístico constructivo para obtener una buena solución inicial. Esta solución se mejora en la segunda fase mediante un algoritmo de búsqueda local. La mejor de todas las soluciones examinadas se guarda como resultado final.

La palabra GRASP proviene de las siglas de Greedy Randomized Adaptive Search Procedures que en castellano sería aproximadamente: Procedimientos de Búsqueda basados en funciones "Voraces.^Aleatorizadas que se Adaptan. Veamos los elementos de este procedimiento.

En la **fase de construcción** se construye iterativamente una solución posible, considerando un elemento en cada paso. En cada iteración la elección del próximo elemento para ser añadido a la solución parcial viene determinada por una función **greedy**. Esta función mide el beneficio de añadir cada uno de los elementos según la función objetivo y elegir la mejor. Notar que esta medida es miope en el sentido que no tiene en cuenta qué ocurrirá en iteraciones sucesivas al realizar una elección, sino únicamente en esta iteración.

Se dice que el heurístico greedy **se adapta** porque en cada iteración se actualizan los beneficios obtenidos al añadir el elemento seleccionado a la solución parcial. Es decir, la evaluación que se tenga de añadir un determinado elemento a la solución en la iteración j , no coincidirá necesariamente con la que se tenga en la iteración $j + 1$.

El heurístico es **aleatorizado** porque no selecciona el mejor candidato según la función greedy adaptada sino que, con el objeto de diversificar y no repetir soluciones en dos construcciones diferentes, se construye un lista con los mejores candidatos de entre los que se toma uno al azar.

Al igual que ocurre en muchos métodos, las soluciones generadas por la fase de construcción de GRASP no suelen ser óptimos locales. Dado que la fase inicial no garantiza la optimalidad local respecto a la estructura de entorno en la que se esté trabajando (notar que hay selecciones aleatorias),

se aplica un procedimiento de búsqueda local como postprocesamiento para mejorar la solución obtenida.

En la **fase de mejora** se suele emplear un procedimiento de intercambio simple con el objeto de no emplear mucho tiempo en esta mejora. Notar que GRASP se basa en realizar múltiples iteraciones y quedarse con la mejor, por lo que no es especialmente beneficioso para el método el detenerse demasiado en mejorar una solución dada.

El siguiente esquema muestra el funcionamiento global del algoritmo:

Algoritmo GRASP

Mientras (Condición de parada)

Fase Constructiva

Seleccionar una lista de elementos candidatos.

Considerar una Lista Restringida de los mejores Candidatos.

Seleccionar un elemento aleatoriamente de la Lista Restringida.

Fase de Mejora

Realizar un proceso de búsqueda local a partir de la solución construida hasta que no se pueda mejorar más.

Actualización

Si la solución obtenida mejora a la mejor almacenada, actualizarla.

El realizar muchas iteraciones GRASP es una forma de realizar un muestreo del espacio de soluciones. Basándonos en las observaciones empíricas, se ve que la distribución de la muestra generalmente tiene un valor en promedio que es inferior al obtenido por un procedimiento determinista, sin embargo, la mejor de las soluciones encontradas generalmente supera a la del procedimiento determinista con una alta probabilidad.

Las implementaciones GRASP generalmente son robustas en el sentido de que es difícil el encontrar ejemplos patológicos en donde el método funcione arbitrariamente mal.

Algunas de las sugerencias de los autores para mejorar el procedimiento son:

- Se puede incluir una fase previa a la de construcción: una fase determinista con el objetivo de ahorrar esfuerzo a la fase siguiente.
- Si se conoce que ciertas subestructuras forman parte de una solución óptima, éstas pueden ser el punto de partida de la fase constructiva.

Tal y como señalan Feo y Resende una de las características más relevantes de GRASP es su sencillez y facilidad de implementación. Basta con fijar el tamaño de la lista de candidatos y el número de iteraciones para determinar completamente el procedimiento. De esta forma se pueden concentrar los esfuerzos en diseñar estructuras de datos para optimizar la eficiencia del código y proporcionar una gran rapidez al algoritmo, dado que éste es uno de los objetivos principales del método.

El enorme éxito de este método se puede constatar en la gran cantidad de aplicaciones que han aparecido en los últimos años. Festa y Resende (2001) comentan cerca de 200 trabajos en los que se aplica o desarrolla GRASP.

5. BÚSQUEDA TABÚ

Los orígenes de la Búsqueda Tabú (Tabu Search, TS) pueden situarse en diversos trabajos publicados a finales de los 70 (Glover, 1977). Oficialmente, el nombre y la metodología fueron introducidos posteriormente por Fred Glover (1989). Numerosas aplicaciones han aparecido en la literatura, así como artículos y libros para difundir el conocimiento teórico del procedimiento (Glover and Laguna, 1997).

TS es una técnica para resolver problemas combinatorios de gran dificultad que está basada en principios generales de Inteligencia Artificial (IA). En esencia es un metaheurístico que puede ser utilizado para guiar cualquier procedimiento de búsqueda local en la búsqueda agresiva del óptimo del problema. Por agresiva nos referimos a la estrategia de evitar que la búsqueda quede “atrapada” en un óptimo local que no sea global. A tal efecto, TS toma de la IA el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente. El principio de TS podría resumirse como:

*Es mejor una mala decisión basada en información que una buena decisión al azar, ya que, en un sistema que emplea **memoria**, una mala elección basada en una estrategia proporcionará claves útiles para continuar la búsqueda. Una buena elección fruto del azar no proporcionará ninguna información para posteriores acciones.*

TS comienza de la misma forma que cualquier procedimiento de búsqueda local, procediendo iterativamente de una solución x a otra y en el entorno de la primera: $N(x)$. Sin embargo, en lugar de considerar todo el entorno de una solución, TS define el **entorno reducido** $N^*(x)$ como aquellas soluciones disponibles del entorno de x . Así, se considera que a partir de x , sólo las soluciones del entorno reducido son alcanzables.

$$N^*(x) \subseteq N(x)$$

Existen muchas maneras de definir el entorno reducido de una solución. La más sencilla consiste en etiquetar como tabú las soluciones previamente visitadas en un pasado cercano. Esta forma se conoce como memoria a corto plazo (short term memory) y está basada en guardar en una lista tabú T las soluciones visitadas recientemente (Recency). Así en una iteración deter-

minada, el entorno reducido de una solución se obtendría como el entorno usual eliminando las soluciones etiquetadas como tabú.

$$N^*(x) = N(x) \setminus T$$

El objetivo principal de etiquetar las soluciones visitadas como tabú es el de evitar que la búsqueda se cicle. Por ello se considera que tras un cierto número de iteraciones la búsqueda está en una región distinta y puede liberarse del status tabú (pertenencia a T) a las soluciones antiguas. De esta forma se reduce el esfuerzo computacional de calcular el entorno reducido en cada iteración. En los orígenes de TS se sugerían listas de tamaño pequeño, actualmente se considera que las listas pueden ajustarse dinámicamente según la estrategia que se esté utilizando.

Se define un **nivel de aspiración** como aquellas condiciones que, de satisfacerse, permitirían alcanzar una solución aunque tenga status tabú. Una implementación sencilla consiste en permitir alcanzar una solución siempre que mejore a la mejor almacenada, aunque esté etiquetada tabú. De esta forma se introduce cierta flexibilidad en la búsqueda y se mantiene su carácter agresivo.

Es importante considerar que los métodos basados en búsqueda local requieren de la exploración de un gran número de soluciones en poco tiempo, por ello es crítico el reducir al mínimo el esfuerzo computacional de las operaciones que se realizan a menudo. En ese sentido, la memoria a corto plazo de TS está basada en atributos en lugar de ser explícita; esto es, en lugar de almacenar las soluciones completas (como ocurre en los procedimientos enumerativos de búsqueda exhaustiva) se almacenan únicamente algunas características de éstas.

La memoria mediante **atributos** produce un efecto más sutil y efectivo en la búsqueda, ya que un atributo o grupo de atributos identifica a un conjunto de soluciones, del mismo modo que los hiperplanos o esquemas utilizados en los algoritmos Genéticos (ver sección 7.1). Así, un atributo que fue etiquetado como tabú por pertenecer a una solución visitada hace n iteraciones, puede impedir en la iteración actual, el alcanzar una solución por contenerlo, aunque ésta sea diferente de la que provocó el que el atributo fuese etiquetado. Esto permite, a largo plazo, el que se identifiquen y mantengan aquellos atributos que inducen una cierta estructura beneficiosa en las soluciones visitadas.

Con los elementos descritos puede diseñarse un algoritmo básico de TS para un problema de optimización dado. Sin embargo, TS ofrece muchos más elementos para construir algoritmos realmente potentes y eficaces. A menudo, dichos elementos han sido ignorados en muchas aplicaciones y actualmente la introducción de éstos en la comunidad científica constituye un reto para los investigadores del área.

Un algoritmo TS está basado en la interacción entre la memoria a corto plazo y la memoria a largo plazo. Ambos tipos de memoria llevan asociadas sus propias estrategias y atributos, y actúan en ámbitos diferentes. Como ya hemos mencionado la memoria a corto plazo suele almacenar atributos de soluciones recientemente visitadas, y su objetivo es explorar a fondo una región dada del espacio de soluciones. En ocasiones se utilizan estrategias de listas de candidatos para restringir el número de soluciones examinadas en una iteración dada o para mantener un carácter agresivo en la búsqueda.

La **memoria a largo plazo** almacena las frecuencias u ocurrencias de atributos en las soluciones visitadas tratando de identificar o diferenciar regiones. La memoria a largo plazo tiene dos estrategias asociadas: Intensificar y Diversificar la búsqueda. La intensificación consiste en regresar a regiones ya exploradas para estudiarlas más a fondo. Para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones encontradas. La Diversificación consiste en visitar nuevas áreas no exploradas del espacio de soluciones. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente.

La siguiente tabla muestra los elementos mencionados.

Memoria	Atributos	Estrategias	Ámbito
Corto Plazo	Reciente	Tabú - Aspiración Listas de Candidatos	Local
Largo Plazo	Frecuente	Intensif. - Diversif.	Global

Existen otros elementos más sofisticados dentro de TS que, aunque poco probados, han dado muy buenos resultados en algunos problemas. Entre ellos se pueden destacar:

Movimientos de Influencia: Son aquellos movimientos que producen un cambio importante en la estructura de las soluciones. Usualmente, en un procedimiento de búsqueda local, la búsqueda es dirigida mediante la evaluación de la función objetivo. Sin embargo, puede ser muy útil el encontrar o diseñar otros evaluadores que guíen a ésta en determinadas ocasiones. Los movimientos de influencia proporcionan una evaluación alternativa de la bondad de los movimientos al margen de la función objetivo. Su utilidad principal es la determinación de estructuras subyacentes en las soluciones. Esto permite que sean la base para procesos de Intensificación y Diversificación a largo plazo.

Oscilación Estratégica: La Oscilación Estratégica opera orientando los movimientos en relación a una cierta frontera en donde el método se detendría normalmente. Sin embargo, en vez de detenerse, las reglas para la elección de los movimientos se modifican para permitir que la región al otro lado de la frontera sea alcanzada. Posteriormente se fuerza al procedimiento

a regresar a la zona inicial. El proceso de aproximarse, traspasar y volver sobre una determinada frontera crea un patrón de oscilación que da nombre a esta técnica. Una implementación sencilla consiste en considerar la barrera de la factibilidad / infactibilidad de un problema dado. Implementaciones más complejas pueden crearse identificando determinadas estructuras de soluciones que no son visitadas por el algoritmo y considerando procesos de construcción / destrucción asociados a éstas. La oscilación estratégica proporciona un medio adicional para lograr una interacción muy efectiva entre intensificación y diversificación.

Elecciones Probabilísticas: Normalmente TS se basa en reglas sistemáticas en lugar de decisiones al azar. Sin embargo, en ocasiones se recomienda el aleatorizar algunos procesos para facilitar la elección de buenos candidatos o cuando no está clara la estrategia a seguir (quizá por tener criterios de selección enfrentados). La selección aleatoria puede ser uniforme o seguir una distribución de probabilidad construida empíricamente a partir de la evaluación asociada a cada movimiento.

Umbrales Tabú: El procedimiento conocido como Tabú Thresholding (TT) se propone para aunar ideas que provienen de la Oscilación Estratégica y de las Estrategias de Listas de Candidatos en un marco sencillo que facilite su implementación. El uso de la memoria es implícito en el sentido que no hay una lista tabú en donde anotar el status de los movimientos, pero la estrategia de elección de los mismos previene el ciclado. TT utiliza elecciones probabilísticas y umbrales en las listas de candidatos para implementar los principios de TS.

Re-encadenamiento de Trayectorias (Path Relinking): Este método se basa en volver a unir dos buenas soluciones mediante un nuevo camino. Así, si en el proceso de búsqueda hemos encontrado dos soluciones x e y con un buen valor de la función objetivo, podemos considerar el tomar x como solución inicial e y como solución final e iniciar un nuevo camino desde x hasta y . Para seleccionar los movimientos no consideraremos la función objetivo o el criterio que hayamos estado utilizando hasta el momento, sino que iremos incorporando a x los atributos de y hasta llegar a ésta. Por eso esperamos que alguna de las soluciones intermedias que se visitan en este proceso de “entorno constructivo” sea muy buena. En algunas implementaciones se ha considerado el explorar el entorno de las soluciones intermedias para dar más posibilidad al descubrimiento de buenas soluciones. Detalles sobre el método pueden encontrarse en Glover, Laguna y Martí (2000).

Laguna y Martí (1999) proponen el uso de Path Relinking (PR) en el contexto de GRASP, aunque aquí su significado es diferente ya que las soluciones no han estado unidas por ningún camino previo. Así, una vez generada una colección de soluciones mediante las fases de construcción y mejora de GRASP, se seleccionan parejas (o subconjuntos) de soluciones para unir las mediante PR. A partir de una solución se realiza una búsqueda

local para llegar a la otra (o a una combinación de las otras en el caso de subconjuntos).

Es importante destacar el hecho de que muchas de las aplicaciones basadas en TS no utilizan los últimos elementos descritos, por lo que son susceptibles de ser mejoradas. Al mismo tiempo, los éxitos de las numerosas implementaciones del procedimiento han llevado la investigación hacia formas de explotar con mayor intensidad sus ideas subyacentes. En este campo podemos destacar los últimos trabajos de *Modelos de entrenamiento y aprendizaje tabú*, *Maquinas tabú* y *Diseño tabú*.

6. TEMPLADO SIMULADO

Kirpatrick, Gelatt y Vecchi proponen en 1983 un procedimiento para obtener soluciones aproximadas a problemas de optimización, llamado Simulated Annealing (Templado simulado). Este procedimiento se basa en una analogía con el comportamiento de un sistema físico al someterlo a un baño de agua caliente. Simulated Annealing (SA) ha sido probado con éxito en numerosos problemas de optimización, mostrando gran “habilidad” para evitar quedar atrapado en óptimos locales. Debido a su sencillez de implementación así como a los buenos resultados que iban apareciendo, experimentó un gran auge en la década de los 80.

Kirpatrick y otros trabajando en el diseño de circuitos electrónicos consideraron aplicar el algoritmo de Metrópolis en alguno de los problemas de optimización combinatoria que aparecen en este tipo de diseños. El algoritmo de Metrópolis simula el cambio de energía en el proceso de enfriamiento de un sistema físico. Las leyes de la termodinámica establecen que, a una temperatura t , la probabilidad de un aumento de energía de magnitud ∂E viene dada por la expresión siguiente:

$$P(\partial E) = e^{\frac{-\partial E}{kt}},$$

donde k es la constante de Boltzmann.

La simulación de Metrópolis genera una perturbación y calcula el cambio resultante en la energía. Si ésta decrece, el sistema se mueve al nuevo estado, en caso contrario, se acepta el nuevo estado de acuerdo con la probabilidad dada en la ecuación anterior.

Kirpatrick y otros, pensaron que era posible establecer una analogía entre los parámetros que intervienen en la simulación termodinámica de Metrópolis y los que aparecen en los métodos de optimización local, tal y como muestra la tabla adjunta.

Termodinámica	Optimización
Configuración	Solución Posible
Configuración Fundamental	Solución óptima
Energía de la Configuración	Coste de la Solución

Para ello, establecen un paralelismo entre el proceso de las moléculas de una sustancia que van colocándose en los diferentes niveles energéticos buscando un equilibrio, y las soluciones visitadas por un procedimiento de búsqueda local. Así pues, SA es un procedimiento basado en búsqueda local en donde todo movimiento de mejora es aceptado y se permiten movimientos de no mejora de acuerdo con unas probabilidades. Dichas probabilidades están basadas en la analogía con el proceso físico de enfriamiento y se obtienen como función de la temperatura del sistema.

La estrategia de SA es comenzar con una temperatura inicial *alta*, lo cual proporciona una probabilidad también alta de aceptar un movimiento de no mejora. En cada iteración se va reduciendo la temperatura y por lo tanto las probabilidades son cada vez más pequeñas conforme avanza el procedimiento y nos acercamos a la solución óptima. De este modo, inicialmente se realiza una diversificación de la búsqueda sin controlar demasiado el coste de las soluciones visitadas. En iteraciones posteriores resulta cada vez más difícil el aceptar malos movimientos, y por lo tanto se produce un descenso en el coste.

De esta forma, SA tiene la habilidad de salir de óptimos locales al aceptar movimientos de no mejora en los estados intermedios. Al final del proceso éstos son tan poco probables que no se producen, con lo que, si no hay movimientos de mejora, el algoritmo finaliza. El diagrama siguiente muestra un esquema general del algoritmo para un problema de minimización:

Algoritmo Templado Simulado

Tomar una solución inicial x

Tomar una temperatura inicial T

Mientras (no congelado)

Realizar L veces

Tomar x' de $N(x)$.

$d = f(x') - f(x)$

Si ($d < 0$) hacer $x = x'$

Si ($d > 0$) hacer $x = x'$ con $p = d^{-e/T}$

Hacer $T = rT$

Para diseñar un algoritmo a partir del esquema general anterior hay que determinar los parámetros del sistema:

- **La Temperatura Inicial** se suele determinar realizando una serie de pruebas para alcanzar una determinada fracción de movimientos aceptados.

- **La Velocidad de Enfriamiento r .**
- La longitud L se toma habitualmente proporcional al tamaño esperado de $N(x)$.
- El criterio para finalizar la **Secuencia de Enfriamiento** (estado de congelación). Usualmente hacemos $cont = cont + 1$ cada vez que se completa una temperatura y el porcentaje de movimientos aceptados es menor de la cantidad $MinPercent$. Contrariamente hacemos $cont = 0$ cuando se mejora la mejor solución almacenada.

La clave de una implementación de SA es el manejo de la cola o secuencia de enfriamiento: Cual es la temperatura inicial y cómo disminuye la temperatura en cada iteración, son las dos preguntas a responder para diseñar un algoritmo. Podemos encontrar numerosos estudios en donde se comparan colas lentas (aquellas en las que la temperatura disminuye poco a poco) con colas rápidas que vienen a ser métodos de descenso simple con pequeñas perturbaciones.

Johnson y otros (1989) sugieren el realizar mejoras y especializaciones aunque no se basen en la analogía física. Además, aconsejan utilizar soluciones posibles y no posibles cuando la estructura del problema lo permita, penalizando en la función de evaluación las soluciones imposibles. El objetivo es diversificar y considerar más soluciones (Notar que con bajas temperaturas la solución tenderá a ser posible).

Las últimas implementaciones de SA apuntan a olvidarse de la analogía física y manejar la cola de enfriamiento como una estructura de memoria. Es decir, la probabilidad de aceptar o rechazar un movimiento de no mejora depende no de la iteración (tiempo transcurrido) sino de lo sucedido en la búsqueda. En este sentido, la probabilidad será función de algunas variables de estado del proceso. En la actualidad se están diseñando numerosos algoritmos híbridos en donde la búsqueda local se realiza con un procedimiento basado en SA, en ocasiones combinado con búsqueda Tabú.

7. MÉTODOS EVOLUTIVOS

Los métodos evolutivos están basados en poblaciones de soluciones. A diferencia de los métodos vistos hasta ahora, en cada iteración del algoritmo no se tiene una única solución sino un conjunto de éstas. Estos métodos se basan en generar, seleccionar, combinar y reemplazar un conjunto de soluciones. Dado que mantienen y manipulan un conjunto en lugar de una única solución a lo largo de todo el proceso de búsqueda, suelen presentar tiempos de computación sensiblemente más altos que los de otros metaheurísticos. Este hecho se ve agravado por la "convergencia" de la población que requiere de un gran número de iteraciones en el caso de los algoritmos genéticos, tal y como se describe a continuación.

7.1. Algoritmos Genéticos

Los Algoritmos Genéticos (GA) fueron introducidos por John Holland en 1970 inspirándose en el proceso observado en la evolución natural de los seres vivos.

Los Biólogos han estudiado en profundidad los mecanismos de la evolución, y aunque quedan parcelas por entender, muchos aspectos están bastante explicados. De manera muy general podemos decir que en la evolución de los seres vivos el problema al que cada individuo se enfrenta cada día es la supervivencia. Para ello cuenta con las habilidades innatas provistas en su material genético. A nivel de los genes, el problema es el de buscar aquellas adaptaciones beneficiosas en un medio hostil y cambiante. Debido en parte a la selección natural, cada especie gana una cierta cantidad de “conocimiento”, el cual es incorporado a la información de sus cromosomas.

Así pues, la evolución tiene lugar en los cromosomas, en donde está codificada la información del ser vivo. La información almacenada en el cromosoma varía de unas generaciones a otras. En el proceso de formación de un nuevo individuo, se combina la información cromosómica de los progenitores aunque la forma exacta en que se realiza es aún desconocida.

Aunque muchos aspectos están todavía por discernir, existen unos principios generales de la evolución biológica ampliamente aceptados por la comunidad científica. Algunos de éstos son:

- La evolución opera en los cromosomas en lugar de en los individuos a los que representan.
- La selección natural es el proceso por el que los cromosomas con “buenas estructuras” se reproducen más a menudo que los demás.
- En el proceso de reproducción tiene lugar la evolución mediante la combinación de los cromosomas de los progenitores. Llamamos Recombinación a este proceso en el que se forma el cromosoma del descendiente. También son de tener en cuenta las mutaciones que pueden alterar dichos códigos.
- La evolución biológica no tiene memoria en el sentido de que en la formación de los cromosomas únicamente se considera la información del período anterior.

Los algoritmos genéticos establecen una analogía entre el conjunto de soluciones de un problema y el conjunto de individuos de una población natural, codificando la información de cada solución en un string (vector binario) a modo de cromosoma. En palabras del propio Holland:

Se pueden encontrar soluciones aproximadas a problemas de gran complejidad computacional mediante un proceso de evolución simulada.

A tal efecto se introduce una función de evaluación de los cromosomas, que llamaremos calidad (“fitness”) y que está basada en la función ob-

jetivo del problema. Igualmente se introduce un mecanismo de selección de manera que los cromosomas con mejor evaluación sean escogidos para “reproducirse” más a menudo que los que la tienen peor.

Los algoritmos desarrollados por Holland (1992) inicialmente eran sencillos pero dieron buenos resultados en problemas considerados difíciles. Un primer esquema de un algoritmo genético se muestra en la Figura 24.

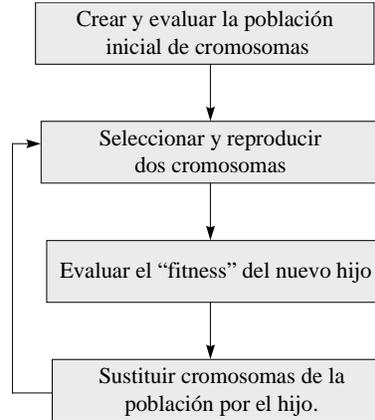


Figura 24

Los algoritmos Genéticos están basados en integrar e implementar eficientemente dos ideas fundamentales: Las representaciones simples como strings binarios de las soluciones del problema y la realización de transformaciones simples para modificar y mejorar estas representaciones.

Para llevar a la práctica el esquema anterior y concretarlo en un algoritmo, hay que especificar los siguientes elementos, que se comentarán a continuación:

- Una representación cromosómica.
- Una población inicial.
- Una medida de evaluación.
- Un criterio de selección / eliminación de cromosomas.
- Una o varias operaciones de recombinación.
- Una o varias operaciones de mutación.

En los trabajos originales las soluciones se representaban por strings binarios, es decir, listas de 1s y 0s. Este tipo de representaciones ha sido ampliamente utilizada incluso en problemas en donde no es muy natural. En 1985, De Jong introduce la siguiente cuestión: ¿Qué se debe hacer cuando los elementos del espacio de búsqueda se representan de modo natural por estructuras complejas como vectores, árboles o grafos?, ¿Se debe intentar linealizar en un string o trabajar directamente con estas estructuras?

En la actualidad podemos distinguir dos escuelas, la primera se limita a strings binarios, mientras que la segunda utiliza todo tipo de configuraciones. Hemos de notar que las operaciones genéticas dependen del tipo de representación, por lo que la elección de una condiciona a la otra.

La ventaja de las primeras es que permite definir fácilmente operaciones de recombinación, además los resultados sobre convergencia están probados para el caso de strings binarios. Sin embargo en algunos problemas puede ser poco natural y eficiente el utilizarlas. Por ejemplo en el problema del agente viajero sobre 5 ciudades y 20 aristas, el string 01000100001000100010 representa una solución sobre las aristas ordenadas. Sin embargo dicha representación no es muy natural y además, no todos los strings con cinco 1s representan soluciones lo cual complica substancialmente la definición de una operación de sobrecruzamiento. Es más natural la ruta de ciudades: (2,3,1,5,4), lo cual permite definir naturalmente diferentes operaciones estables.

La **población inicial** suele ser generada aleatoriamente. Sin embargo, últimamente se están utilizando métodos heurísticos para generar soluciones iniciales de buena calidad. En este caso, es importante garantizar la diversidad estructural de estas soluciones para tener una representación de la mayor parte de población posible o al menos evitar la convergencia prematura.

Respecto a la **evaluación** de los cromosomas, se suele utilizar la calidad como medida de la bondad según el valor de la función objetivo en el que se puede añadir un factor de penalización para controlar la infactibilidad. Este factor puede ser estático o ajustarse dinámicamente, lo cual produciría un efecto similar al de la Oscilación Estratégica en Tabu Search:

$$\begin{aligned} \text{calidad} &= \text{ValorObjetivoNormalizado} \\ &- \text{Penalización} \times \text{MedidaInfactibilidad} \end{aligned}$$

La **selección** de los padres viene dada habitualmente mediante probabilidades según su fitness. Uno de los procedimientos más utilizado es el denominado de la ruleta en donde cada individuo tiene una sección circular de una ruleta que es directamente proporcional a su calidad. Para realizar una selección se realizaría un tirada de bola en la ruleta, tomando el individuo asociado a la casilla donde cayó la bola.

Los Operadores de Cruzamiento más utilizados son:

- **De un punto:** Se elige aleatoriamente un punto de ruptura en los padres y se intercambian sus bits.
- **De dos puntos:** Se eligen dos puntos de ruptura al azar para intercambiar.

- **Uniforme:** En cada bit se elige al azar un padre para que contribuya con su bit al del hijo, mientras que el segundo hijo recibe el bit del otro padre.
- **PMX, SEX:** Son operadores más sofisticados fruto de mezclar y aleatorizar los anteriores.

La operación de **Mutación** más sencilla, y una de la más utilizadas consiste en reemplazar con cierta probabilidad el valor de un bit. Notar que el papel que juega la mutación es el de introducir un factor de diversificación ya que, en ocasiones, la convergencia del procedimiento a buenas soluciones puede ser prematura y quedarse atrapado en óptimos locales. Otra forma obvia de introducir nuevos elementos en una población es recombinar elementos tomados al azar sin considerar su fitness.

A continuación mostramos la implementación de un algoritmo genético propuesta por Michalewicz (1996), en donde se combinan los elementos genéticos con la búsqueda local.

Algoritmo Genético

-
1. **Generar soluciones** — Construir un conjunto de soluciones P con tamaño $PopSize$ mediante generación aleatoria.
 2. **Mejorar soluciones** — Aplicar un método de búsqueda local a cada solución del conjunto P .
- Mientras** (número de evaluaciones $<$ $MaxEval$)
3. **Evaluación** — Evaluar las soluciones en P y actualizar, si es necesario, la mejor solución almacenada.
 4. **Supervivencia** — Calcular la probabilidad de supervivencia basada en la calidad de las soluciones. Según dichas probabilidades seleccionar aleatoriamente $PopSize$ soluciones (con reemplazamiento) de P . Sea P el nuevo conjunto formado por las soluciones seleccionadas (algunas pueden aparecer repetidas).
 5. **Combinación** — Seleccionar una fracción p_c de soluciones de P para ser combinadas. La selección es aleatoria y equi-probable para todos los elementos de P . Los elementos seleccionados se emparejan al azar y, por cada pareja, se generan dos descendientes que reemplazaran a los padres en P .
 6. **Mutación** — Una fracción p_m de las soluciones de P se selecciona para aplicar el operador de mutación. La solución resultante reemplaza a la original en P .
-

Convergencia del Algoritmo

Dado que el algoritmo genético opera con una población en cada iteración, se espera que el método converja de modo que al final del proceso la población sea muy similar, y en el infinito se reduzca a un solo individuo.

Se ha desarrollado toda una teoría para estudiar la convergencia de estos algoritmos en el caso de strings binarios. Esta teoría se basa principalmente en considerar que un string es un representante de una clase de equivalencia

o esquema, reinterpretando la búsqueda en lugar de entre strings, entre esquemas. De este modo se concluye lo que se conoce como **paralelismo intrínseco**:

En una población de m strings se están procesando implícitamente $O(m^3)$ esquemas.

A partir de este resultado el teorema de esquemas prueba que la población converge a unos esquemas que cada vez son más parecidos, y en el límite a un único string.

En el caso de strings no binarios se introducen los conceptos de *forma* y *conjunto de similitud* que generalizan al de esquema. Se consideran una serie de condiciones sobre los operadores de manera que se garantice la convergencia. Básicamente se exige que al cruzar dos strings de la misma clase se obtenga otro dentro de ésta. Además hay que respetar ciertas condiciones sobre selección de los progenitores. Bajo toda esta serie de hipótesis se prueba la convergencia del algoritmo.

En la práctica no se suelen respetar las condiciones vistas ya que son difíciles de seguir y probar, encontrándonos con que, en ocasiones los algoritmos genéticos resuelven satisfactoriamente un problema de optimización dado y otras se quedan muy alejados del óptimo. Los estudiosos del tema han tratado de caracterizar lo que han denominado problemas *AG-fáciles* (aquellos en los que los AG proporcionan buenos resultados) y *AG-difíciles* con el objetivo de saber de antemano, al estudiar un nuevo problema, si los AG son una buena elección para su resolución.

Se han tratado de caracterizar estas clases mediante el concepto de *engaño* considerando que si el algoritmo converge al mejor esquema (aquel con mejor promedio del fitness de sus strings) y en éste se encuentra el óptimo, entonces es fácil que se resuelva satisfactoriamente. En caso de que el óptimo esté en un esquema con bajo promedio se denomina engaño y se pensaba que en estos casos es cuando el problema es AG-difícil. Sin embargo se ha visto que esta caracterización mediante el engaño no es siempre cierta y no constituye un criterio fiable.

Es importante citar que, a diferencia de otros metaheurísticos, los Algoritmos Genéticos han crecido de forma espectacular, hasta el punto de poder encontrar referencias sobre ellos en revista de informática de carácter general. Además muchos de los investigadores de este campo están trabajando en desarrollar los aspectos teóricos de la materia, e incorporando algunas otras técnicas de búsqueda local en el esquema genético. Los **Algoritmos Meméticos** son un caso particular de estos nuevos híbridos que aúnan los elementos de combinación con los de búsqueda local bajo el nombre de cooperación y competición.

Podemos encontrar esquemas de procedimientos genéticos ya implementados en los que incorporar nuestras funciones de evaluación y con poco

más, tener un algoritmo genético para nuestro problema. Existen numerosos accesibles en internet (algunos de libre distribución y otros comercializados por compañías de software) y son muy populares especialmente en el entornos informáticos. Estos esquemas reciben el nombre de *context-independent* y habitualmente sólo utilizan la evaluación de la solución como información del problema. En general proporcionan resultados de menor calidad que la de los algoritmos específicos (dependientes del contexto) diseñado para un problema dado, pero por otro lado, su aplicación es muy sencilla.

8. BÚSQUEDA DISPERSA

La Búsqueda Dispersa (BD, en inglés Scatter Search) es un método evolutivo que ha sido aplicado en la resolución de un gran número de problemas de optimización. Los conceptos y principios fundamentales del método, fueron propuestos a comienzo de la década de los setenta, basados en las estrategias para combinar reglas de decisión, especialmente en problemas de secuenciación, así como en la combinación de restricciones (como el conocido método de las restricciones subrogadas). La BD se basa en el principio de que la información sobre la calidad o el atractivo de un conjunto de reglas, restricciones o soluciones puede ser utilizado mediante la combinación de éstas. En concreto, dadas dos soluciones, se puede obtener una nueva mediante su combinación de modo mejore a las que la originaron.

Al igual que los algoritmos genéticos, el método que nos ocupa se basa en mantener un conjunto de soluciones y realizar combinaciones con éstas; pero a diferencia de éstos no está fundamentado en la aleatorización sobre un conjunto relativamente grande de soluciones sino en las elecciones sistemáticas y estratégicas sobre un conjunto pequeño de éstas. Como ilustración basta decir que los algoritmos genéticos suelen considerar una población de 100 soluciones mientras que en la búsqueda dispersa es habitual trabajar con un conjunto de tan sólo 10 soluciones.

La primera descripción del método fue publicada en 1977 por Fred Glover donde establece los principios de la BD. En este primer artículo se determina que la BD realiza una exploración sistemática sobre una serie de buenas soluciones llamadas conjunto de referencia. Los siguientes comentarios resumen los principales aspectos de este trabajo:

- El método se centra en combinar dos o más soluciones del conjunto de referencia. La combinación de más de dos soluciones tiene como objetivo el generar centroides.
- Generar soluciones en la línea que unen dos dadas se considera una forma reducida del método.

- Al combinar se deben de seleccionar pesos apropiados y no tomar valores al azar.
- Se deben de realizar combinaciones "convexas" y "no convexas" de las soluciones.
- La distribución de los puntos se considera importante y deben de tomarse dispersos.

En Glover (1994) se introduce la combinación ponderada (weighted combination) como el mecanismo principal para generar nuevas soluciones. En esta versión se enfatizan las búsquedas lineales entre dos soluciones y el uso de pesos para muestrear en dicha línea. Asimismo, se introduce el concepto de combinar soluciones de calidad con soluciones diversas. Además, el método incluye una componente de intensificación que consiste en tomar una muestra mayor de la línea que ha producido mejores soluciones.

En este trabajo se especifica que para trabajar con problemas con variables enteras, binarias o que forman una permutación, hay que diseñar métodos específicos de combinación (notar que no tiene sentido hablar de combinación lineal de dos permutaciones). Para ello se introducen los mecanismos de combinación basados en votos. En éstos se definen reglas mediante las que cada solución "vota" para que sus características aparezcan en la solución que se está construyendo. Estos métodos de votos han sido muy utilizados en las rutinas de combinación de los algoritmos de BD y parece que constituyen uno de las claves del éxito de estos métodos. A continuación mostramos un ejemplo sobre un problema de rutas de vehículos introducido en Corberán et al. (2000) para ilustrarlos.

Dada una serie de localizaciones en donde hay que recoger a unos estudiantes, el problema consiste en encontrar un conjunto de rutas de modo que cada una sea recorrida por un autobús. A los efectos que aquí nos ocupan podemos omitir el resto de detalles y considerar que tenemos dos soluciones del problema para ver cómo se combinan. Consideremos un ejemplo con 10 localizaciones y dos soluciones (A y B) con dos rutas cada una:

Solución A : $A_1 = 4, 2, 7, 1, A_2 = 5, 10, 6, 9, 3, 8$

Solución B : $B_1 = 2, 6, 8, 10, 9, B_2 = 3, 4, 7, 1, 5$

Para poder combinar A y B, tendremos que asociar las rutas de cada una. Si consideramos las dos asociaciones posibles (A_1 con B_1 y A_2 con B_2 ó, A_1 con B_2 y A_2 con B_1) y contamos el número de elementos comunes, podemos tomar la que mayor número de coincidencias presente. Así, asociamos A_2 con B_1 y A_1 con B_2 ya que la primera tiene cuatro elementos en común (6, 8, 9 y 10), y la segunda tiene tres (4, 7 y 1), mientras que la otra asignación presenta $1+2=3$ coincidencias. La nueva solución (N_1

y N_2) se construye en n pasos, en los que una localización se asigna a una ruta en cada paso. La siguiente tabla muestra el proceso.

Paso	Par	Voto 1	Voto 2	Asignación	Regla de selección
1	(A ₁ ,B ₂)	4	3	$N_1 = \{ 4 \}$	azar
2	(A ₂ ,B ₁)	5	2	$N_2 = \{ 2 \}$	azar
3	(A ₁ ,B ₂)	7	3	$N_1 = \{ 4, 3 \}$	3 antes 7
4	(A ₂ ,B ₁)	5	6	$N_2 = \{ 2, 5 \}$	5 antes 6
5	(A ₁ ,B ₂)	7	7	$N_1 = \{ 4, 3, 7 \}$	igual
6	(A ₂ ,B ₁)	10	6	$N_2 = \{ 2, 5, 6 \}$	azar
7	(A ₁ ,B ₂)	1	1	$N_1 = \{ 4, 3, 7, 1 \}$	igual
8	(A ₂ ,B ₁)	10	8	$N_2 = \{ 2, 5, 6, 10 \}$	10 antes 8
9		9	8	$N_2 = \{ 2, 5, 6, 10, 8 \}$	8 antes 9
10		9	9	$N_2 = \{ 2, 5, 6, 10, 8, 9 \}$	igual

En el paso 1, comenzamos construyendo la ruta N_1 con el par (A_1, B_2) . El primer elemento en la ruta A_1 es el 4, así que esta ruta vota para que el 4 sea el primer elemento de la ruta N_1 (Voto 1). Análogamente el voto asociado con B_2 es para la localización 3 (Voto 2). Como ambas localizaciones ocupan en sus rutas la misma posición (la primera) los dos votos valen lo mismo por lo que desempata al azar y gana el 4. En el paso 2 realizamos lo mismo para la otra pareja de rutas. En el paso 3, A_1 vota por la localización 7 (la 4 y la 2 ya han sido asignadas) y B_2 vota por la 3. Como la 7 está en la tercera posición de la ruta A_1 y la localización 3 está en la primera posición de la ruta B_2 , la regla específica dar preferencia a la localización en una posición anterior, por lo que 3 es seleccionada. Procediendo de igual forma se completa la nueva solución hasta obtener $N_1 = 4, 3, 7, 1$ y $N_2 = 2, 5, 6, 10, 8, 9$.

En 1977 Glover publica una versión más específica del método en donde se recogen y simplifican muchas de las ideas expuestas en trabajos anteriores. Esta publicación tuvo un gran impacto en lo que a la difusión del método se refiere y se ha quedado como la referencia *standard* de la búsqueda dispersa. Numerosos investigadores comenzaron a aplicar la BD a la resolución de problemas de optimización obteniendo resultados de gran calidad. La siguiente sección describe esta versión del método, actualizada según implementaciones y desarrollos posteriores.

Glover, Laguna y Martí (2000) estudian las implementaciones más recientes del método en la resolución de problemas de optimización combinatoria. Además, muestran las conexiones entre este método y el denominado Re-encadenamiento de Trayectorias ("Path relinking"). En el texto de Laguna y Martí (2002b) podemos encontrar una descripción exhaustiva del método desde sus orígenes hasta las estrategias más avanzadas. El libro incluye código fuente en C que implementa los métodos descritos.

Desde un punto de vista espacial, el proceso de generar combinaciones lineales de un conjunto de referencia de soluciones, puede ser visto como el generar caminos entre, y más allá, de estas soluciones. Esto lleva a una concepción más amplia del significado de combinar que es la introducida en el re-encadenamiento de trayectorias (RT).

RT se basa en el hecho de que entre dos soluciones se puede trazar un camino que las una, de modo que las soluciones en dicho camino contengan atributos de ellas. Las soluciones originales pueden haber sido generadas mediante un método basado en una búsqueda local y estar unidas por un camino, o haber sido generadas por otro método y no estar unidas de ningún modo; en cualquier caso, ahora generaremos un nuevo camino que las una. Las características de dicho camino vendrán especificadas respecto de los atributos que son añadidos o eliminados, o por los movimientos realizados para alcanzar una solución desde la otra. Esto constituye una extensión del concepto de combinación en tanto que se obtienen varias soluciones a partir de dos o más originales.

Consideremos en un problema de permutaciones dos soluciones $x=(1,3,4,2)$ e $y=(2,3,1,4)$. Si aplicamos un método de combinación podemos obtener una determinada solución $z=(1,3,2,4)$, mientras que si tratamos de obtener y partiendo de x , podemos obtener la secuencia $z_1=(2,1,3,4)$, $z_2=(2,3,1,4)$. En este sentido decimos que RT es una extensión de los métodos de combinación.

Dado que la búsqueda dispersa se basa en realizar combinaciones y aplicar métodos de búsqueda local, se puede considerar incluido en los llamados *algoritmos meméticos* (la clase de algoritmos evolutivos en donde la búsqueda local se aplica de forma selectiva). Actualmente, existen implementaciones comerciales del método (del tipo “context-independent” mencionado), como OptQuest Callable Library (Laguna y Martí, 2000), que están compitiendo en la industria con métodos de optimización establecidos en la resolución de problemas reales. A continuación se describen las partes esenciales de un algoritmo de búsqueda dispersa para pasar después a comentar algunas mejoras y aspectos avanzados del método.

El Algoritmo de Búsqueda Dispersa

El método se basa en combinar las soluciones que aparecen en el llamado conjunto de referencia. En este conjunto se tienen las soluciones buenas que se han ido encontrando. Es importante destacar que el significado de buena no se restringe a la calidad de la solución, sino que también se considera la diversidad que esta aporta al conjunto. La Búsqueda Dispersa consta básicamente de los siguientes elementos:

- **Un generador de soluciones diversas.** El método se basa en generar un conjunto P de soluciones diversas (alrededor de 100), del que extraer-

emos un subconjunto pequeño (alrededor de $b = 10$) con el que realizar las combinaciones y que denominamos R .

▪ **Un conjunto de referencia R .** Extraído del conjunto de soluciones diversas según el criterio de contener soluciones de calidad y diferentes entre sí (Calidad y Diversidad). Si el método no logra mejorar a la solución, se considera que el output es la propia solución considerada. Las soluciones en este conjunto están ordenadas de mejor a peor respecto de su calidad.

- **Creación.** Iniciamos el conjunto de referencia con las $b/2$ mejores soluciones de P . Las $b/2$ restantes se extraen de P por el criterio de máxima distancia con las ya incluidas en el conjunto de referencia. Para ello debemos de definir previamente una función de distancia en el problema.

- **Actualización.** Las soluciones fruto de las combinaciones pueden entrar en el conjunto de referencia y reemplazar a alguna de las ya incluidas si las mejoran. Así pues, el conjunto de referencia mantiene un tamaño b constante pero va mejorando a lo largo de la búsqueda. En implementaciones sencillas, la actualización de este conjunto se realiza únicamente por calidad, aunque podemos hacerlo también por diversidad.

▪ **Un método de combinación.** BD se basa en combinar todas las soluciones del conjunto de referencia. Para ello, se consideran subconjuntos de 2 o más elementos del conjunto de referencia y se combinan mediante una rutina diseñada a tal efecto. La solución o soluciones que se obtienen de esta combinación pueden ser inmediatamente introducidas en el conjunto de referencia (actualización dinámica) o almacenadas temporalmente en una lista hasta terminar de realizar todas las combinaciones y después ver qué soluciones entran en éste (actualización estática).

▪ **Un método de mejora.** Típicamente se trata de un método de búsqueda local para mejorar las soluciones, tanto del conjunto de referencia como las combinadas antes de estudiar su inclusión en el conjunto de referencia.

El siguiente esquema muestra cómo actúan los elementos descritos en un esquema básico del algoritmo.

Algoritmo de Búsqueda Dispersa

-
1. Comenzar con $P = \emptyset$. Utilizar el **método de generación** para construir una solución y el **método de mejora** para tratar de mejorarla; sea x la solución obtenida. Si $x \notin P$ entonces añadir x a P . (i.e., $P = P \cup x$), en otro caso, rechazar x . Repetir esta etapa hasta que P tenga un tamaño prefijado.
 2. Construir el **conjunto de referencia** $R = \{x^1, \dots, x^b\}$ con las $b/2$ mejores soluciones de P y las $b/2$ soluciones de P más diversas a las ya incluidas.
 3. Evaluar las soluciones en R y ordenarlas de mejor a peor respecto a la función objetivo.
 4. Hacer NuevaSolución = TRUE
- Mientras** (NuevaSolución)
5. NuevaSolución = FALSE
 6. Generar los subconjuntos de R en los que haya al menos una nueva solución.
Mientras (Queden subconjuntos sin examinar)
 7. Seleccionar un subconjunto y etiquetarlo como examinado.
 8. Aplicar el **método de combinación** a las soluciones del subconjunto.
 9. Aplicar el **método de mejora** a cada solución obtenida por combinación. Sea x la solución mejorada:
Si $f(x)$ mejora a $f(x^b)$ y x no está en R)
 10. Hacer $x^b = x$ y reordenar R
 11. Hacer NuevaSolución = TRUE
-

El algoritmo hace referencia a los subconjuntos de R ya que podemos combinar parejas, tríos o cualquier número de soluciones. Es usual el limitar las combinaciones a parejas, por lo que el punto 6 equivaldría a decir: "Generar todas las parejas de soluciones de R en las que al menos una de las dos sea nueva"; donde por nueva entenderemos que haya entrado al conjunto después de realizar la última combinación de todo R .

Notar que el algoritmo se detiene cuando al tratar de combinar vemos que no hay nuevos elementos en el conjunto de referencia (la variable NuevaSolución está en 0). Este algoritmo puede ser anidado en un esquema global que permita reconstruir el conjunto de referencia cuando éste ya ha sido explotado. Así, si el límite de tiempo (o evaluaciones) no se ha excedido, una estrategia habitual es regenerar el conjunto de referencia dejando la mitad superior ($b/2$ mejores) y eliminando la mitad inferior. Después, se genera un conjunto P como al comienzo del algoritmo, del que se extraen únicamente las $b/2$ soluciones más diversas con las ya existentes en R . Así obtenemos un nuevo conjunto de referencia en el que mantenemos las soluciones de calidad y renovamos las debidas a diversidad. Ahora se vuelve a combinar como anteriormente sobre este conjunto de referencia (pasos 5 a 11). De este modo se obtiene un esquema cíclico indefinido al que hay que añadirle una variable de control para detenerlo, típicamente esta variable está en función del tiempo o del número de iteraciones (evaluaciones de la función objetivo).

Desarrollo y Mejoras

La búsqueda dispersa es un método relativamente reciente que hemos de considerar en desarrollo. Durante los últimos años se han realizado nuevas contribuciones aplicando esta metodología en la resolución de conocidos problemas de optimización. Algunas de estas aplicaciones han abierto nuevos campos de estudio, ofreciendo alternativas a los diseños conocidos. Laguna y Martí (2002b) realizan una revisión de los aspectos clave del método desarrollados durante estos años. A continuación las resumimos de forma esquemática:

- Considerar el **uso de memoria** basada en la frecuencia para desarrollar métodos de diversificación eficientes. En lugar de generar las soluciones al azar, se propone construir un método que basado en el número de apariciones de los elementos significativos en la solución, evite la repetición de estructuras similares.
- Inicializar el conjunto de referencia a partir de un gran conjunto de soluciones creado con el generador antes mencionado.
- **Aplicar la rutina de mejora de forma selectiva.** Las pruebas indican que el aplicar el método de mejora a todas las soluciones generadas y combinadas, no garantiza el obtener mejores resultados finales. Establecer umbrales de calidad para no aplicar la mejora a soluciones que difícilmente van a proporcionar la mejor solución, es un gasto innecesario de tiempo de computación (Ugray et al., 2001). Por otro lado, al aplicar el método de mejora a todas las soluciones se acelera la convergencia de éste, lo cual puede ser deseable si disponemos de poco tiempo de computación, pero debemos de evitarlo si queremos ejecutar el método en un horizonte largo para obtener soluciones de gran calidad.
- Es necesario estudiar el porcentaje de tiempo que el método está generando soluciones y el tiempo que está combinando. En esencia esta es la cuestión que se plantea en todos los métodos heurísticos: el **equilibrio entre la intensificación y la diversificación.**
- Inicializar el conjunto de referencia con la mitad de soluciones por calidad y la otra mitad por diversidad. Se han realizado experimentos con distintos tamaños y parece ser que esta proporción es la que mejores resultados está dando.
- La calidad es más importante que la diversidad al actualizar el conjunto de referencia. Notar que aunque el método comienza con un conjunto de referencia con soluciones de calidad y diversidad, al realizar las combinaciones, sólo entran al conjunto las soluciones por el criterio de calidad (hasta llegar a la etapa de regenerarlo). En Laguna y Martí (2000) se han probado actualizaciones de este conjunto por diversidad, obteniendo resultados finales peores.

- Comparar las **actualizaciones** del conjunto de referencia **estática y dinámica**. Notar que al combinar las soluciones podemos aplicar dos estrategias, introducirlas en el conjunto, si procede, nada más generarlas, o anotarlas en una “pila” y cuando terminemos de realizar todas las combinaciones, proceder a la actualización. La primera estrategia es dinámica y más agresiva, en tanto que las soluciones buenas entran rápidamente en el conjunto de referencia, pero dado que este es de tamaño constante, esto implica que hay soluciones que salen sin llegar a haber sido utilizadas para realizar combinaciones. Es necesario comparar ambas estrategias para ver cual proporciona mejores resultados finales.

- La mayor parte de las soluciones de calidad proceden de combinaciones de dos soluciones. Asimismo, las buenas soluciones suelen proceder de combinar buenas soluciones. Campos et al. (1999) realizan numerosos experimentos realizando un seguimiento a lo largo de la búsqueda de las soluciones de calidad en un problema concreto.

- El uso de **múltiples métodos de combinación** ha de ser considerado. Martí y otros (2002) realizan un análisis de diferentes métodos de combinación, algunos con elementos aleatorios y otros deterministas, de modo que el algoritmo selecciona el método de combinación probabilísticamente, de acuerdo con los éxitos obtenidos por éste. De la misma forma que los métodos de búsqueda local basados en distintos entornos (Variable Neighborhood Search) están proporcionando muy buenos resultados, hemos de considerar el definir varios “entornos” de combinación para realizar una búsqueda más completa del espacio de soluciones.

Como conclusión podemos decir que la búsqueda dispersa es un método evolutivo que se encuentra en desarrollo. Sus orígenes se pueden situar en la década de los 70 y, aunque menos conocida que los algoritmos genéticos, se está aplicando en la resolución de numerosos problemas difíciles de optimización.

En la actualidad no existe un esquema único para aplicar la búsqueda dispersa. En este trabajo hemos tratado de introducir aquellos aspectos básicos del método que son ampliamente aceptados, así como revisar las últimas implementaciones realizadas y las cuestiones que son actualmente objeto de estudio.

AGRADECIMIENTOS

Este texto recoge los contenidos impartidos en el curso de doctorado “Procedimientos Heurísticos” del Departamento de Estadística e Investigación Operativa de la Universitat de València durante los cursos 96/97, 98/99 y 2000/01, así como en la Universitat Politècnica de Catalunya en el curso 95/96. La versión actual incluye las modificaciones sugeridas, de una u otra forma, por estudiantes y profesores. A todos ellos, mi más sincero agradecimiento.

REFERENCES

1. Campos V., Laguna M. y Martí R. (1999), Scatter Search for the Linear Ordering Problem, in *New Ideas in Optimisation*, D. Corne, M. Dorigo and F. Glover (Eds.), McGraw-Hill. 331–341.
2. Campos, V., F. Glover, M. Laguna and R. Martí (1999), An Experimental Evaluation of a Scatter Search for the Linear Ordering Problem, *Journal of Global Optimization*, 21(4), 397–414.
3. Corberán A., E. Fernández, M. Laguna and R. Martí (2000), Heuristic Solutions to the Problem of Routing School Buses with Multiple Objectives, *Journal of the Operational Research Society*, 53(4), 427–435.
4. Davis, L. (1996), *Handbook of Genetic Algorithms*, International Thomson Computer Press, Londres.
5. Díaz, A., Glover, F., Ghaziri, H.M., Gonzalez, J.L., Laguna, M, Moscato, P. y Tseng, F.T. (1996), *Optimización Heurística y Redes Neuronales*, Paraninfo, Madrid.
6. Feo, T. and Resende, M.G.C. (1989), A probabilistic heuristic for a computational difficult set covering problems, *Operations research letters*, 8, 67–71.
7. Feo, T. and Resende, M.G.C. (1995), Greedy Randomized Adaptive Search Procedures, *Journal of Global Optimization*, 2, 1–27.
8. Festa, P. and Resende, M.G.C. (2001), GRASP: An Annotated Bibliography, *AT&T Labs Research Tech. Report*.
9. Fisher, M.L. (1980), Worst-Case Analysis of Heuristic Algorithms, *Management Science*, 26, 1–17.
10. Glover, F. (1977), Heuristics for Integer Programming Using Surrogate Constraints, *Decision Sciences*, 8, 156–166.
11. Glover, F. (1986), Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research*, 13, 533–549.
12. Glover, F. (1989), Tabu Search: Part I, *ORSA Journal on Computing*, 1, 190–206.
13. Glover, F. (1990), Tabu Search: Part II, *ORSA Journal on Computing*, 3, 223–254.
14. Glover, F. (1999), A Template for Scatter Search and Path Relinking, in *Artificial Evolution, Lecture Notes in Computer Science*, J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer and D. Snyers (Eds.), Springer-Verlag, 13–54.
15. Glover, F., M. Laguna and R. Martí (1999), Scatter Search, in *Theory and Applications of Evolutionary Computation: Recent Trends*, A. Ghosh and S. Tsutsui (Eds.), Springer-Verlag, forthcoming.
16. Glover, F., M. Laguna and R. Martí (2000), Fundamentals of Scatter Search and Path Relinking, *Control and Cybernetics*, 29 (3), 653–684.
17. Glover, F., M. Laguna, E. Taillard and D. de Werra (1993), A user’s guide to tabu search, *Annals of Operations Research*, 4, 3–28.
18. Glover, F. and Laguna, M. (1997), *Tabu Search*, Ed. Kluwer, London.
19. Holland, J.H. (1992), Genetic Algorithms, *Scientific American*, 267, 66.
20. Johnson, D.S., Aragon, C.R., McGeoch, L.A. and Schevon, C. (1989), Optimization by Simulated Annealing: An experimental evaluation; Part I, Graph Partitioning, *Operations Research*, 37.
21. Johnson, D.S., Aragon, C.R., McGeoch, L.A. and Schevon, C. (1991), Optimization by Simulated Annealing: An experimental evaluation; Part II, Graph Coloring and Number Partitioning, *Operations Research*, 39.

22. Jünger, M., Reinelt, G. y Rinaldi, G. (1995), The Traveling Salesman Problem, in *Handbook in Operations Research and Management Science, Vol. 7*, Ball, M.O., Magnanti, T.L., Monma, C.L. y Nemhauser, G.L. (Eds.), North-Holland, Amsterdam, 225–330.
23. Kirkpatrick, S., Gelatt, C.D. and Vecchi, P.M. (1983), Optimization by simulated annealing, *Science*, 220, 671–680.
24. Laguna M. and Martí R. (1999), GRASP and Path relinking for two layer straight-line crossing minimization, *INFORMS Journal on Computing*, 11(1), 44–52.
25. Laguna M. and Martí R. (2000), Experimental Testing of Advanced Scatter Search Designs for Global Optimization of Multimodal Functions, *Technical report TR11-2000*, Departamento de Estadística e I.O., Universidad de Valencia.
26. Laguna M. and Martí R. (2002a), The OptQuest Callable Library, in *Optimization Software Class Libraries*, Voss and Woodruff (Eds.), Kluwer, 193–218.
27. Laguna M. and Martí R. (2002b), *Scatter Search*, Ed. Kluwer, London.
28. Laporte, G. (1992), The Travelling Salesman Problem: An Overview of Exact and Approximate Algorithms, *European Journal of Operational Research*, 59, 231–247.
29. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. y Shmoys, D.B. (eds.) (1985), *The Traveling Salesman Problem. A Guided Tour to Combinatorial Optimization*, Ed. John Wiley and Sons, Chichester.
30. Lin, S. y Kernighan, B.W. (1973), An Effective Heuristic Algorithm for the Traveling Salesman Problem, *Operations Research*, 21, 498–516.
31. Lokketangen, A. and Glover, F. (1996), Probabilistic move selection in tabu search for 0/1 mixed integer programming problems, in *Meta-Heuristics: Theory and Practice*, Kluwer, 467–488.
32. Martí, R. (2000), MultiStart Methods, in *State of the Art Handbook on MetaHeuristics*, F. Glover and G. Kochenberger (Eds.), Kluwer, forthcoming.
33. Martí, R., M. Laguna and V. Campos (2000), Scatter Search Vs. Genetic Algorithms: An experimental evaluation with permutation problems, in *Adaptive Memory and Evolution: Tabu Search and Scatter Search*, Cesar Rego y Bahram Alidaee (Eds.), Kluwer, forthcoming.
34. Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs*, Ed. Springer Verlag.
35. Miller, C. E., Tucker, A. W. and Zemlin, R. A. (1960), Integer programming formulations and the traveling salesman problem, *J. Assoc. Comput. Mach*, 7, 326–329.
36. Osman, I.H. and Kelly, J.P. (eds.) (1996), *Meta-Heuristics: Theory and Applications*, Ed. Kluwer Academic, Boston.
37. Padberg, M.W. y Hong, S. (1980), On the Symmetric Travelling Salesman Problem: A Computational Study, *Mathematical Programming Study*, 12, 78–107.
38. Reeves, C.R. (1995), *Modern Heuristic Techniques for Combinatorial Problems*, Ed. McGraw-Hill, UK.
39. Rinnooy Kan, A.H.G. and Timmer, G.T. (1989), Global Optimization, in *Handbooks in operations research and management science*, Rinnooy Kan and Todds (Eds.), North Holland, 1, 631–662.
40. Rochat, I. and E. Taillard (1995), Probabilistic diversification and intensification in local search for vehicle routing, *Journal of heuristics*, 1(1), pp 147–167.
41. Silver, E.A., Vidal, R.V. y De Werra, D. (1980), A Tutorial on Heuristic Methods, *European Journal of Operational Research*, 5, 153–162.

42. Ugray Z., L. Lasdon, J. Plummer, F. Glover, J. Kelly and R. Martí (2001), A Multi-start Scatter Search Heuristic for Smooth NLP and MINLP Problems, in *Adaptive Memory and Evolution: Tabu Search and Scatter Search*, Cesar Rego y Bahram Alidaee (Eds.), Kluwer, forthcoming.
43. Ulder, N.L.J., Aarts, E.H.L., Bandelt, H.J., Van Laarhoven, P.J.M. and Pesch, E. (1990), Genetic Local Search algorithms for the traveling salesman problem, in *Parallel problem solving from nature*, Schwefel and Manner (Eds.), Springer Verlag, 109-116.
44. Whitley D. (1993), A genetic Algorithm Tutorial, *Tech. Report CS-93-103*, Colorado State University.