Desarrollo y aplicación del algoritmo PSO al problema del ATSP

Trabajo fin de grado

Morán Bermúdez, Noemi
Grado en Ingeniería de Organización Industrial
Escuela de Ingenierías Industriales
Universidad de Valladolid
Julio de 2015

Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA DE ORGANIZACIÓN INDUSTRIAL

Desarrollo y aplicación del algoritmo PSO al problema del TSP

Autor:

Morán Bermúdez, Noemi

Tutor:

Pérez Vázquez, María Elena
Dpto. de Organización de Empresas

Valladolid, Julio de 2015.

RESUMEN

En la vida cotidiana nos enfrentamos a varios problemas. Uno de los más comunes se da cuando debemos realizar varias actividades o ir a varios lugares y tenemos que decidir el orden para realizarlo, de manera que nos suponga el menor coste posible ya sea en tiempo, energía, distancia, dinero… Esta situación se puede representar mediante el problema de Optimización del Agente Viajero (TSP).

En este documento se va a desarrollar e implementar al TSP (práctica y teóricamente mediante la programación del Algoritmo) un método aproximado de optimización, basado en el comportamiento de los enjambres de abejas en la naturaleza, para buscar una buena solución al problema. Este método es conocido como Optimización por Enjambre de Partículas (PSO).

Palabras clave: Optimización por Enjambre de Partículas, Problema del Agente Viajero, Métodos Metaheurísticos, problemas NP-Duros, operador de permutación.

ABSTRACT

In our quotidian life we deal with problems. One of the most common arrives when we must do several activities and go to several places, and we have to decide which order is the best one, when it comes to use as less time, energy, distance, money, etc., as possible. This situation could be represented by the Travelling Salesman Optimization problem (TSP).

In this document I'll develop and implement to TSP (as theoretically as practically with the algorithm programming) an approximated optimization system, based in the behavior of swarms of bees in the nature, to look for a good problem solution. This system is known as Particle System Optimization (PSO).

Keywords: Particle Swarm Optimization, Traveling Salesman Problem, Metaheuristic Methods, NP-Hard problems, swap operator.

ÍNDICE

INTRODUCCIÓN Y OJETIVOS

El algoritmo de optimización por enjambre de partículas (PSO), desarrollado originalmente por Kennedy y Eberhart, es un método aproximado de optimización, de tipo metaheurístico basado en la población, que simula el comportamiento natural de los enjambres de abejas a la hora de buscar el polen, cuya característica principal es que poseen memoria y que el enjambre esta interrelacionado.

Las abejas vuelan en el espacio en busca de flores y se van comunicando unas con otras para saber cuál de ellas ha encontrado mayor densidad floral y dónde. Después, cada abeja sigue su búsqueda, modificando su vuelo en función del mejor lugar encontrado por el enjambre y del mejor lugar encontrado por ella misma. De este modo el enjambre cada vez encuentra lugares con mayor densidad de flores.

En el PSO, cada partícula es una solución potencial al problema, y posee una posición actual y una velocidad. La población se inicializa de manera aleatoria y después, de forma iterativa va evolucionando a mejores soluciones del problema mediante la actualización de la velocidad y la posición de cada partícula. En primer lugar se actualiza la velocidad teniendo en cuenta la mejor posición global encontrada por la población y la mejor posición encontrada hasta el momento por la propia partícula, y posteriormente se actualiza la posición aplicándole la velocidad hallada.

La solución del algoritmo será la mejor posición encontrada por el conjunto del enjambre, que al tratarse de una metaheurística, aunque no se puede asegurar que sea el resultado óptimo, sí que será próximo a él.

Este método según está descrito, sirve para buscar soluciones en espacios de búsqueda continuos, sin embargo, el problema al que nosotros nos enfrentamos es un problema discreto llamado Problema del Agente Viajero (TSP). Este problema es del tipo NP-Hard y es uno de los que más interés suscita en la comunidad científica debido a la simplicidad con la que se enuncia y a la dificultad que supone su resolución.

El TSP se puede describir de la siguiente forma: Un viajero debe visitar n ciudades pasando por todas ellas, sin repetir ninguna, y volviendo a la ciudad de origen, de manera que se genera un ciclo Hamiltoniano. El objetivo del problema es encontrar el ciclo óptimo que debe realizar el viajante, es decir, encontrar la trayectoria que menor distancia total suponga de entre todas las posibles.

La dificultad de su resolución está en que los posibles ciclos que se pueden llevar a cabo aumentan de forma exponencial con el tamaño del problema, por lo que probar cada una de las soluciones posibles no es una opción.

Este trabajo surge de la necesidad de adaptar la metaheurística PSO a un espacio discreto, para así poderla implementar al problema TSP, y obtener así una buena solución.

Para lograr ese objetivo, este trabajo se estructura de la siguiente manera:

En el primer capítulo se explicará en detalle en qué consiste el TSP para saber con exactitud las características del problema al que nos enfrentamos.

El segundo y tercer capítulo están enfocados al desarrollo del PSO; en el segundo se introducirán los diferentes métodos de optimización de problemas, mientras que el tercero está centrado en el método PSO que nos concierne en este trabajo.

El cuarto capítulo es la parte práctica del trabajo. Está compuesto por tres partes diferenciadas: en la primera se explica cómo vamos a aplicar el PSO al TSP, es decir, cómo vamos a adaptar el PSO obteniendo una versión discreta del mismo. Una vez que sabemos cómo queremos realizar la implementación, se lleva a la práctica mediante la creación de un algoritmo en lenguaje C++ que será explicado paso a paso en el segundo apartado. Para finalizar el capítulo, el algoritmo será aplicado a una serie de ejemplos para poder hacer el estudio experimental que compruebe la fiabilidad del algoritmo.

1. PROBLEMA DEL AGENTE VIAJERO

Dado que lo que buscamos resolver en este documento es una de las versiones del Problema del Agente Viajero (TSP), en concreto la variante asimétrica, también conocida como ATSP, y éste pertenece a los problemas NP-Hard de optimización combinatoria, en este capítulo comenzamos introduciendo las características de los problemas de decisión y la clasificación de los mismos, para así describir en qué consiste un problema NP-Completo (al que pertenece la versión de decisión del TSP) y posteriormente la clase NP-Hard, para con ello poder contextualizar el TSP. Finalmente describiremos el TSP y la versión asimétrica del mismo, que es el problema que realmente nos concierne en este trabajo.

Antes de comenzar exponemos una breve definición del TSP para hacernos una idea de lo que pretendemos resolver:

El Problema del Agente Viajero consiste en hallar el mejor ciclo Hamiltoniano (el de mínima distancia) que podría realizar un viajante que tiene que ir a n ciudades, sabiendo las distancias entre ellas. Para que el ciclo sea Hamiltoniano tiene que cumplir que: debe partir de una ciudad concreta, recorrer todas las demás ciudades sin repetir ninguna, y regresar a la de partida.

Lo que el ATSP añade al TSP original es que la distancia de la ciudad i a la j no tiene por qué ser la misma que la distancia de la ciudad j a la i.

1.1 Problemas de Decisión

Descrito de la manera más simple posible, un problema en términos computacionales es un agregado de frases finitas, que tiene asociado otro conjunto finito de frases como respuesta.

Un problema de optimización combinatoria es tanto un problema de minimización como uno de maximización que tiene asociadas un conjunto de instancias. El término problema se aplica a la cuestión general que debe ser contestada (habitualmente formado por varios parámetros y variables sin valores específicos), mientras que la palabra instancia se refiere a un problema al que ya se le han asignado valores específicos para todos los parámetros, por ejemplo, el problema TSP es un problema general que busca un ciclo Hamiltoniano de coste mínimo, mientras que una instancia del TSP es un problema específico en el que ya se sabe el número de ciudades que posee, sus conexiones y los costes de ir de unas a otras.

Además todo problema de optimización, que lo que hace es buscar la mejor solución posible del problema (recorrido mínimo en el caso del TSP), tiene varias versiones [1], como por ejemplo: la versión de evaluación, la de localización, la de decisión…

El caso concreto de los problemas de decisión es aquel donde las únicas respuestas posibles son "si" o "no", por ello la versión de decisión de un problema de optimización tiene el siguiente aspecto: ¿Existe una solución posible del problema menor o mayor (según sea de minimización o maximización) que un número dado?, y más concretamente para el TSP sería: ¿Dado un número k, existe un circuito Hamiltoniano del problema de longitud menor o igual que k?

De la clasificación de los problemas de decisión, según la complejidad computacional que implica su resolución, se encargan las Ciencias de la Computación. Para ello se consideran todos los posibles algoritmos que pueden resolver el problema y se estudia los recursos que requieren (el tiempo y la memoria que supone el problema en su ejecución).

1.1.1 Clasificación de los Problemas de Decisión

Los problemas de decisión se pueden clasificar en base a dos criterios, ambos estudiados por áreas de la Teoría de la computación:

- Según la Teoría de la computabilidad, que estudia los lenguajes y los cataloga en función de si son resolubles con diferentes máquinas y modelos de computación o no lo son, los problemas de decisión pueden ser clasificados como [2]:

    - Decidibles, resolubles o computables: Siempre que el problema tenga solución es capaz resolverlo, al menos, mediante un procedimiento mecánico o algoritmo (entendiendo por algoritmo una sucesión ordenada y finita de pasos relacionados mediante los cuales resolvemos el problema). Además cuando se le presenta un problema que no tiene solución, es capaz de reconocerlo.

    - Parcialmente decidible, reconocibles o semicomputables: Existe al menos un algoritmo capaz de resolver el problema, siempre que éste tenga solución. Pero no es capaz de distinguir cuándo no existe solución, de modo que en ese caso iteraría de forma infinita.

    - No Decidible: No existe ningún algoritmo que resuelva el problema, independientemente de que posean o no solución. Puesto que no se

resuelven, se puede decir que su complejidad es infinita, por lo que como veremos a continuación, este tipo de problemas no son estudiados por la Teoría de la Complejidad Computacional.

- El segundo criterio es el que da la Teoría de la Complejidad Computacional, área que mediante herramientas y mecanismos describe y analiza la complejidad de los algoritmos y problemas. Éste área sólo estudia los problemas decidibles, considerando el tiempo que implican y la memoria que ocupan, debido a que carece de sentido estudiar la complejidad de los problemas no decidibles cuando ya sabemos que es infinita.
  Podemos ver en la clasificación que aunque un problema sea computable, puede que de forma práctica no sea posible resolverlo debido al exceso de recursos que requiere. Los principales problemas según esta consideración son [3]:

  - Clase P.
    Se trata de algoritmos de complejidad polinómica (P). Se pueden resolver en un tiempo polinómico con una máquina de Turing determinista (dadas unas variables de entrada y el estado en que se encuentra la computadora, si hay varias acciones posibles, la máquina sólo puede realizar una de ellas, en la no determinista la máquina se replica y continúa cada una por una rama distinta) y secuencial (realiza cada acción después de que la anterior haya finalizado).
    El tiempo polinómico es el tiempo ideal, lo cual significa que el tiempo de ejecución del algoritmo es menor que cierto valor calculado (con una fórmula polinómica, que tiene en cuenta los datos de entrada, por lo que el tiempo polinómico es proporcional a los mismos), por ello un algoritmo eficiente es aquel cuya complejidad es polinómica.

  - Clase NP.
    Son problemas de decisión que se resuelven en un tiempo polinómico (P) con una máquina de Turing no determinista (N). Podemos decir que la clase P es un tipo dentro de la clase NP.
    De este grupo forman parte muchos problemas de optimización y de búsqueda cuyo objetivo es averiguar si existe una determinada solución, o si existe una mejor que la conocida hasta el momento.

  - Clase NP-Completo.
    Para que un problema sea NP-Completo se tienen que cumplir dos características: que sea un NP (los completos son los problemas más difíciles de la clase NP), y que todos los problemas NP se puedan

reducir (relacionar dos problemas, de forma que si el primero se puede resolver con un algoritmo, se garantiza que hay un algoritmo que resuelve el segundo) en cada uno de los problemas NP-Completos, es decir, que todo problema NP mediante la transformación polinómica puede convertirse en NP-Completo.
Todos los problemas que cumplen la condición de la transformación polinómica, aunque no cumplan la pertenencia a NP, son problemas NP-Hard.

- Clase NP-Hard, NP-Complejo o NP-Duro [4].
  La clase NP-Hard no es exclusiva de los problemas de decisión ya que no tienen que cumplir la condición de ser NP, de hecho, todos los NP-Completos pertenecen a los NP-Hard, pero no se cumple la afirmación en sentido contrario.
  Los NP-Hard son aquellos problemas que cumplen que todos los problemas de NP se pueden transformar polinómicamente en NP-Hard.
  Hasta el momento no se ha encontrado ningún algoritmo eficiente (acotado polinómicamente) que proporcione la solución óptima, ni para este tipo de problemas, ni para los anteriores. Por ello, la forma de actuar ante estos problemas es realizar una búsqueda exhaustiva (enumeración de todas las posibles soluciones y elección de la mejor) si las entradas son "pequeñas", o en caso contrario, buscar una solución cercana al óptimo construyendo soluciones mediante algoritmos polinómicos de aproximación.
  Se trata de problemas, como mínimo tan difíciles como los NP, ya que si encontrásemos un algoritmo que resolviese en tiempo polinómico uno de los problemas NP-Hard (se cree que no existe ningún algoritmo polinómico que los resuelva, pero nunca ha sido demostrado), se podría construir un algoritmo polinómico para cualquier problema NP, mediante la reducción del problema NP en NP-Hard y la posterior ejecución del algoritmo que resuelve el NP-Hard. Si existiese ese algoritmo, significaría que P=NP y esto sería un descubrimiento muy importante para la teoría de la computación, tanto, como que hay un premio del Instituto Clay que otorga un millón de dólares a quién logre dicha demostración.
  Teniendo en cuenta las características de la clase NP-Hard, lo que sí que podemos asegurar a día de hoy es que los NP-Completos son la intersección de los NP con los NP-Hard.

En la figura 1 se muestra mediante el diagrama de Euler la relación que existe entre los tipos de problemas anteriormente explicados. Debido a la cantidad de incógnitas que existen sobre el tema a día de hoy, vemos las

dos versiones principales: a la izquierda el caso en el que no se ha demostrado la existencia de un algoritmo polinómico que resuelva un problema NP-Hard y a la derecha el caso contrario, por lo que P es igual que NP y que NP Completo, siendo además todos ellos NP-Hard.

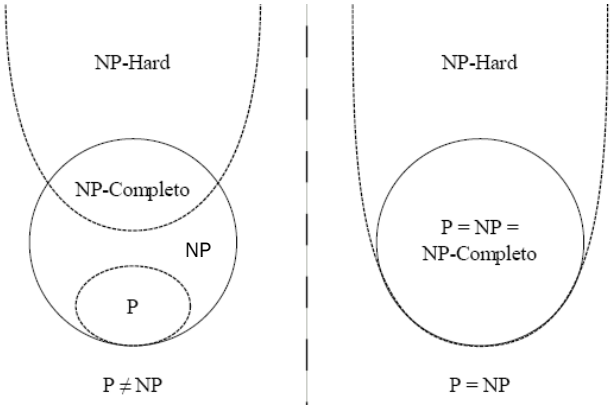


Figura 1. Problemas según la complejidad computacional. [5].

1.2 Problema del Agente Viajero

El Problema del Agente Viajante o Vendedor Viajero, también conocido como TSP por sus siglas del inglés "Traveling Salesman Problem", fue definido en del siglo XIX por el irlandés W.R. Hamilton y el británico Thomas Kirkman, ambos matemáticos. Y fue durante los años 1930s cuando matemáticos de Viena y Harvard estudiaron por primera vez la forma general del TSP, siendo muy importante para su desarrollo Karl Menger.

El origen del nombre que se otorga hoy en día a este problema es un poco misterioso (Applegate et al., 2011), puesto que se han encontrado documentos en los que los autores utilizan esa nomenclatura, pero en ninguno de ellos se dan indicios de que el autor del documento sea el creador del nombre "Traveling Salesman Problem", sino que se da a entender que el nombre ya estaba establecido.

El primer documento en que tenemos constancia del término data de 1949 y es un artículo de Julia Robinson, pero según el contenido del artículo ella no parece ser la persona que introdujo el nombre. Varios autores creen que el término fue creado durante los años 30 o 40 en la universidad de Princeton.

Este problema fue poco a poco haciéndose más popular entre los científicos europeos y americanos y su estudio fue evolucionando de modo que cada vez se podían resolver problemas de mayor dimensión. En la tabla 1 se muestra a grandes

rasgos ese progreso en el tiempo, indicando el año en que se resuelve el TSP de la dimensión indicada en la columna "Número de ciudades" y los autores responsables de dicho estudio.

Año	Autores	Número de ciudades
1954	Dantzig, Fulkerson, and Johnson	49
1971	Held and Karp	64
1975	Camerini, Fratta, and Maffioli	67
1977	Grötschel	120
1980	Crowder and Padberg	318
1987	Padberg and Rinaldi	532
1987	Grötschel and Holland	666
1987	Padberg and Rinaldi	2.392
1994	Applegate, Bixby, Chvátal, and Cook	7.397
1998	Applegate, Bixby, Chvátal, and Cook	13.509
2001	Applegate, Bixby, Chvátal, and Cook	15.112
2004	Applegate, Bixby, Chvátal, Cook, and Helsgaun	24.978
2005	Cook, Espinoza and Goycoolea	33.810

Tabla 1. Evolución de la dimensión del TSP. [6].

El TSP se puede formular de la siguiente manera (Lin, 1965): Un viajante o vendedor debe visitar todas y cada una de las n ciudades dadas una sola vez, comenzando por una de ellas y volviendo a la misma ciudad de origen al finalizar el recorrido, de manera que se genera un ciclo. Dado que hay que hay que cumplir que se va a cada ciudad exactamente una vez, el ciclo resultante es un ciclo Hamiltoniano.

Lo que se busca en el TSP es elegir la ruta o tour (orden en que las ciudades son visitadas) que optimiza el resultado, es decir, la que implique menos coste (utilizamos coste de forma general y puede estar referido a la distancia, el tiempo, el coste de traslado… eso depende del objetivo del problema. El más común en este tipo de problemas es la distancia recorrida) de entre todas las posibles.

Una solución para una instancia del TSP puede ser representada como una permutación de los índices de las ciudades (una ordenación de las ciudades). Dicha permutación es cíclica por lo que la posición absoluta de una ciudad dentro del tour no es relevante, lo que verdaderamente importa es la secuencia, es decir, el orden relativo entre unas ciudades y otras. Lo que esto significa es que no es importante que una ciudad se visite en primer o en segundo lugar, lo que afecta es qué ciudades se han visitado antes y después de ella, dicho con otras palabras, la ciudad de partida no afecta al resultado del problema.

Al tratarse de un problema tan sencillo de entender, a priori se suele pensar que la complejidad de su resolución también lo es, pero cuando se indaga en la cantidad de estudios que se han realizado durante toda la historia del problema y el hecho

de que hasta el momento nadie ha conseguido encontrar un método efectivo para su resolución, aun siendo uno de los problemas de optimización combinatoria más estudiados, se cae en la cuenta de la complejidad computacional que envuelve al problema.

Es tal su complejidad que el TSP forma parte de los problemas NP-Hard (fue Richard M. Karp quien en 1972 demostró que el Ciclo de Hamilton era NP-Completo, lo que implica que el TSP es NP-Hard, esto será demostrado más adelante), por lo que posee las características explicadas en el apartado anterior para esa clase.

Además se trata de un problema de optimización (lo que implica buscar valores para variables discretas de modo que se encuentre la solución óptima respecto a una función objetivo dada) y sufre el fenómeno de la explosión combinatoria, que quiere decir que cuando el número de variables del problema aumenta, el número de combinaciones posibles y los esfuerzos computacionales también, y lo hacen de forma exponencial, por ello se dice que el TSP es un problema de optimización combinatoria.

De manera teórica el algoritmo de "fuerza bruta", que consiste en probar todas las posibles rutas del problema, es óptimo y fácil de plantear, el problema llega al impleméntalo de forma práctica debido al tiempo que supone. Las posibles rutas que se pueden tomar, dadas n ciudades y pasando exactamente una vez por cada una de ellas, son n!, pero puesto que el problema especifica el punto de partida (aunque no lo especificase, al tratarse de un ciclo Hamiltoniano, el resultado del problema es independiente del punto de partida), las posibles rutas de un TSP son (n-1)!, por lo que queda demostrado que el problema aumenta de forma exponencial (más bien factorial). Para un problema de 24 ciudades nos encontramos ente la friolera de veinticinco mil trillones de posibilidades, y si el problema contase con unos pocos cientos de ciudades sería inabordable en la práctica porque no hay un ordenador tan potente como para poder resolverlo.

Esto provoca que sea inviable resolver el problema mediante un algoritmo que pruebe todas las posibilidades, por lo que en el siguiente capítulo veremos algoritmos que aunque no obtienen el óptimo del problema nos proporcionan una solución mucho más eficiente.

1.2.1 Definición formal del problema

Para la descripción formal del TSP vamos a introducir primero la nomenclatura que utilizamos para el problema y algunas nociones sobre la teoría de grafos (Martí, 2003), para finalmente exponer la formulación del problema:

- G es el grafo del problema y está formado por V, A y C, por lo que se representación es la siguiente: G = (V, A, C).

- V es el conjunto de vértices, también llamados nodos, que en nuestro caso son las ciudades que forman el problema V = {1, 2, …, n}.
  Al tamaño de V lo denominamos dimensión (n), que es equivalente al número de ciudades que forma el problema.

- A es el conjunto de aristas o arcos del problema, es decir, el camino o unión que hay para ir de un vértice a otro.
  Hablamos de un grafo completo si para cada par de vértices existe una arista que los une.

- C es la matriz de costes, donde c_{ij} es el coste que supone (generalmente en términos de distancia) ir de la ciudad i a la ciudad j, es decir, el coste asociado a la arista (i,j).
  Habitualmente C es de tamaño nxn, poniendo en las filas las ciudades de origen y en las columnas las de destino. Como en cada fila se pone la distancia de la ciudad de esa fila a todas las demás, y son n ciudades en total, hay n columnas y como cualquier ciudad puede ser tomada como origen también hay n filas.

Cuando nos enfrentamos a un problema del tipo TSP, los únicos datos de los que partimos para su resolución son los que forman el grafo del problema, es decir, los vértices, aristas y costes. En este documento asumimos que el grafo que se nos proporciona es el grafo completo, ya que al ser el caso más general el planteamiento que hagamos nos servirá para cualquier caso particular.

Que el grafo sea completo significa que para cada par de ciudades existe un camino de unión entre ellas. Si nos encontramos ante un problema que no une directamente dos vértices, creamos una arista ficticia de unión cuyo coste será equivalente al coste del camino más corto que une a ambas ciudades. Por ejemplo en la figura 2 podríamos añadir una arista ficticia (1-7) de coste 9 equivalente al camino (1-3-7).

Una sucesión de aristas (a_1, a_2,…, a_k) donde el vértice final de una arista es el mismo que el inicial de la arista sucesiva, es un camino. Además lo denominamos camino simple si cumple la condición de que no pasa por el mismo vértice más de una vez.

Un ciclo es un camino cuyo vértice final de a_k coincide con el vértice inicial de a_1 y se trata de un ciclo simple si el camino que lo compone también lo es.

Cuando hablamos de un ciclo simple que no utiliza todos los vértices del grafo del problema estamos hablando de un subtour. En el caso opuesto, cuando un ciclo es simple y pasa por todos y cada uno de los vértices que componen el grafo del problema nos encontramos ante un tour o ciclo Hamiltoniano por lo que se puede denotar como (a_1, a_2,..., a_n). Es éste último tipo el que se requiere construir en el TSP, incluyendo además la condición de que sea de coste mínimo.

En la figura 2 vemos un grafo de un problema, donde hay 8 vértices representados con círculos amarillos numerados, varias aristas que se reconocen porque son rectas de unión entre dos vértices y el coste de cada arista que está representado con un número al lado de su arista respectiva. El camino marcado en negrita es uno de los posibles ciclos Hamiltonianos del grafo.

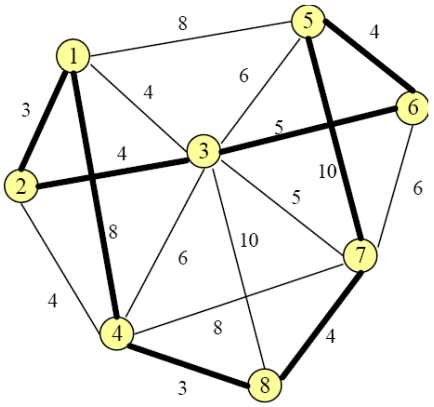


Figura 2. Ciclo Hamiltoniano. Martí, 2003.

Uno de los modos de formular el TSP es el que utilizamos a continuación [7], que utiliza variables binarias en un modelo programación lineal entera. La función objetivo del TSP consiste en minimizar el coste del recorrido que realiza el viajante, y, según dicho modelo, tiene el siguiente aspecto:

$$\text{Min} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (c_{ij} x_{ij})$$

Donde n es el número de ciudades del problema, c_{ij} es el coste de ir de la ciudad i a la j y x_{ij} es la variable binaria {0,1} del problema, que toma el valor de uno en caso de que el ciclo Hamiltoniano posea esa arista (i,j) y el valor cero en caso contrario (por lo que, como es lógico, si no se va por el camino que une la ciudad i con la j, su coste no se suma).

Además el problema debe cumplir las siguientes restricciones:

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} (x_{ij}) = 1 \quad \text{para} \quad j = 1, 2, \ldots, n$$

$$\sum_{\substack{j=1 \\ j\neq i}}^{n}(x_{ij}) = 1 \quad \text{para} \quad i = 1, 2, \ldots, n$$

$$u_i - u_j + n x_{ij} \leq n - 1 \quad \text{para} \quad 1 \leq i \neq j \leq n$$

La primera restricción asegura que no se pueda llegar a una misma ciudad desde varias y que sí que se llegue a ella, es decir que la ciudad de llegada tenga uno y sólo un origen.

La segunda restricción asegura que no se pueda salir desde una misma ciudad a varias y que además todas las ciudades tengan una salida, formando así parte del recorrido, lo que significa que cada ciudad tenga uno y sólo un destino.

La tercera restricción obliga a que sea sólo un camino el que cubra todas las ciudades, es decir, evita que dos o más caminos disjuntos se complementen cubriendo en conjunto las n ciudades. Para ello se asegura que todo camino tiene que pasar por la ciudad 1. La variable u_i es una variable de libre elección que indica el orden en que se realiza el camino, es decir, si u_i = t quiere decir que la ciudad i es visitada en el paso t, siendo el orden t = 1, 2, ..., n. Definimos el recorrido con origen y final en la ciudad 1 (lo cual no implica pérdida de generalidad puesto que se trata de un ciclo Hamiltoniano, por lo que el punto de partida no afecta a la solución). Se cumple que $u_i - u_j \leq n - 1$ ya que u_i no puede ser mayor que n (u_i = n es la última ciudad visitada) y u_j no puede ser menor que 1 (como muy pronto se visita la primera), además para el caso en que la variable x_{ij} = 1 tenemos que $u_i - u_j + n x_{ij} = (t) - (t + 1) + n = n - 1$.

El cumplimiento de las tres restricciones obliga a que el camino creado sea un ciclo y que además sea Hamiltoniano.

1.2.2 Complejidad computacional del TSP

Como hemos dicho en el apartado de clasificación de los problemas de decisión, el problema del agente viajero pertenece a la clase NP-Completo (por tanto también es NP-hard). Para demostrarlo trataremos a continuación las dos condiciones que debe cumplir todo problema para pertenecer a dicha clase [8]:

- El TSP es un problema NP.
  Para comprobar que un tour del problema es válido, lo primero que hay que hacer es comprobar que posee todos los vértices del problema, posteriormente sumar los costes de todas las aristas y comprobar que el coste total de la trayectoria es mínimo.

Todo este proceso se puede realizar en tiempo polinómico por lo que queda demostrado que el TSP pertenece a la clase de problemas NP.

- Todos los problemas NP se pueden trasformar en NP-Completo mediante transformación polinómica, es decir, el problema tiene que ser NP-hard.
  Para realizar la comprobación tomamos como referencia los ciclos Hamiltonianos, puesto que se sabe que son problemas NP-Completos (lo demostró en 1972 Richard M. Karp).
  Partimos de G = (V, A), siendo éste un ejemplo de ciclo Hamiltoniano. A partir de ahí creamos el grafo completo G' = (V, A'), donde A' son las aristas (i,j) para todo i, j pertenecientes a V (excepto i=j). Los costes de dichas aristas son 0 en caso de que la arista pertenezca a A (ciclo Hamiltoniano) y 1 en caso de que no pertenezca a A.
  Una vez creado el grafo completo suponemos que h es un ciclo Hamiltoniano que existe en G, por tanto el coste de cada arista de h en G' es 0, lo que hace que el coste total del recorrido h en G' sea 0. Esto significa que si el grafo G posee un ciclo Hamiltoniano, entonces G' tendrá un ciclo de coste 0.
  Inversamente suponemos que G' posee un ciclo h' de coste 0. Esto solo puede ocurrir en el caso de que todas las aristas de h' sean de coste 0 y para que puedan ser de coste 0 tienen que pertenecer a A.
  Ambos supuestos implican que G posee un ciclo Hamiltoniano, si y sólo si, G' posee un ciclo de coste 0, por lo que queda demostrado que el TSP es un problema NP-Completo.

1.2.3 Aplicaciones del TSP

El TSP tiene una amplia aplicación, sobre todo en los campos del transporte y la logística, tanto para optimizar rutas en sí mismas, como para servir de herramienta (subproblema) a problemas de optimización combinatoria mayores.

Pero lo cierto es que se puede emplear en cualquier situación que requiera seleccionar nodos en un orden determinado que permita minimizar el coste, por lo que podemos decir que el TSP va mucho más allá de los problemas de planificación de rutas, utilizándose también para otras aplicaciones como pueden ser: minimización de los desplazamientos en fabricación, recogida robotizada de material en almacenes, programación de una máquina de taladrado, impresión de circuitos electrónicos, organización de estaciones de trabajo…

A continuación explicaremos brevemente algunas de las aplicaciones del problema en cuestión (Gutin y Punnen, 2007):

- Problemas de programación de máquinas.
  Uno de los problemas más estudiados es el de organización y secuenciación de máquinas, donde los vértices son las n tareas que deben ser procesadas de forma secuencial en una máquina y la matriz de costes es el conjunto de c_{ij} siendo éste el coste de puesta en marcha (configuración, cambios de herramienta…) que supone cambiar de la tarea i a la j.
  Cuando todas las tareas son procesadas la maquina se resetea, volviendo a su estado inicial, lo cual supondrá el coste c_{n1}.
  Lo que se busca en esta aplicación es encontrar la secuencia de tareas que minimiza los costes totales de puesta en marcha de la máquina.

- Fabricación celular.
  Cuando se poseen familias piezas que requieren procesamientos similares, éstas se agrupan y se procesan juntas en una célula especializada para mejorar la eficiencia y reducir los costes. En la mayor parte de los casos en las células se utilizan robots para manejar el material y así reducir el tiempo de procesado. La secuenciación de las actividades a desarrollar por el robot puede ser considerada TSP.
  Un ejemplo de este problema es el siguiente:
  Secuenciar las actividades del robot en una célula robótica de dos máquinas (M1 y M2), existiendo n tipos de piezas y teniendo que ser todas ellas procesadas por la M1 y posteriormente por la M2. El proceso de una pieza es: el robot coge la pieza del stock inicial, la lleva a la M1, la procesa la M1, el robot la coge de la M1 y la lleva a la M2, la procesa la M2 y finalmente el robot la coge de la M2 y la lleva al stock final.
  Para este problema se consideran 2 ciclos de movimientos robóticos posibles, en el primero el robot coge una pieza de M1, la lleva a M2, espera a que se procese, la recoge y la deja en el stock final, luego va al inicial, coge otra pieza, la lleva a M1, espera a que se procese, la lleva a M2 y así sucesivamente, mientras que en el segundo ciclo considerado el robot coge una pieza de M1, la lleva a M2 y en lugar de esperar a que se procese, se va a por otra pieza al stock inicial y la lleva a la M1, vuelve a la M2 (espera si no ha acabado), coge la pieza y la lleva al stock final, va a la M1 (espera si es necesario), coge la pieza y la lleva a la M2 y así sucesivamente.
  Lo que se debe hacer es seleccionar en cada etapa (para este ejemplo, cuando se acaba de procesar la pieza en M1) qué ciclo robótico se elige para minimizar el coste.
  La elección del ciclo que más conviene para una etapa dada depende del tiempo que tarda el robot en trasladarse entre máquinas y stocks y del tiempo necesario para que las máquinas 1 y 2 realicen el proceso de la pieza.

- Problemas de asignación de frecuencia:
  Este problema puede ser por ejemplo, el de asignar una frecuencia a cada transmisor de una red de comunicaciones, de entre un conjunto dado de frecuencias posibles que satisfacen unas restricciones de interferencias.
  Las restricciones se pueden representar en el grafo G = (V, A), donde V son los n transmisores y el coste equivale a la tolerancia necesaria, por lo que se debe elegir una frecuencia (entre las posibles), tal que la frecuencia del transmisor i menos la frecuencia del transmisor j (en valor absoluto), tiene que ser mayor que la tolerancia c_{ij}.
  Lo que se busca es minimizar el rango de frecuencias utilizadas entre las posibles, cumpliendo siempre la restricción de la tolerancia.

- Estructuración de matrices:
  Partimos de una matriz X = $(x_{ij})_{mxn}$ y asumimos que f es una función de coste específica para cada problema.

  - Para cada fila i de la matriz tenemos $L_i\ (X) = \sum_{j=1}^{n} f(x_{ij},\ x_{i(j+1)})$ teniendo en cuenta que $x_{i(n+1)} = x_{i1}$. Por tanto $L\ (X) = \sum_{i=1}^{m} L_i\ (X)$.

  - Del mismo modo para cada columna j, $C_j\ (X) = \sum_{i=1}^{m} f(x_{ij},\ x_{(i+1)j})$, donde $x_{(m+1)j} = x_{1j}$. El sumatorio de $C_j\ (X)$ para todas las columnas es $C\ (X) = \sum_{j=1}^{n} C_j\ (X)$.

  Supongamos que queremos resolver un problema de estructuración de filas. A partir de la matriz X obtenemos S (X), que es la familia de matrices obtenida de realizar permutaciones en las filas de X (o al realizar permutaciones en las columnas en caso de que el problema a tratar sea de estructuración de columnas).
  Lo que se pretende en este problema es buscar la matriz Y perteneciente a S (X), de modo que C (Y) sea al mínimo C de todas las matrices que forman el conjunto S (X).
  Este problema se puede resolver mediante el TSP de grafo completo donde los vértices son las m filas V = {1,2,...,m} y el coste de las aristas es $c_{ik} = \sum_{j=1}^{n} f(x_{ij},\ x_{kj})$, para todo i, k pertenecientes a V.
  Si se tratase de un problema de estructuración de columnas de la matriz también se podría resolver como un TSP del mismo modo que la estructuración de filas pero sustituyendo C por L y construyendo el conjunto S (X) mediante permutaciones de columnas de X.

1.2.4 Versiones del TSP

Existen muchas versiones diferentes de este problema, que difieren en ciertas restricciones que añaden al problema original. Algunas de las más estudiadas son las que exponemos a continuación (Larré, 2012):

- TSP Métrico o delta-TSP.
  Este tipo es aquel en el que los costes (o distancias) del problema cumplen la desigualdad triangular, es decir que el coste de ir de una ciudad a otra pasando por una intermedia, no puede ser menor que el coste de ir directamente ($c_{uv} + c_{vw} \geq c_{uw}$ para todo vértice u, v y w pertenecientes al conjunto de vértices del problema V).
  Cuando las ciudades del problema están en un plano hay varios modos de calcular la distancia [7], dos de ellas son las que generan estos casos particulares del TSP Métrico:

  - TSP Euclidiano o TSP Planal: la distancia entre dos ciudades es la distancia euclidiana entre los puntos en el plano de dichas ciudades.

  - TSP Rectilíneo: Para calcular la distancia entre dos ciudades se halla la distancia de Manhattan, es decir, la suma de las diferencias entre las coordenadas “x” e “y” de las ciudades en el plano.
    Este es el caso, por ejemplo, del traslado de un robot de un punto a otro, cuyo modo de movimiento es realizar primero el desplazamiento en una coordenada, y luego en la otra.

- TSP Simétrico (STSP).
  Este es el caso en el que los costos o distancias del problema son simétricos, es decir, que el coste de ir de una ciudad a otra es igual en ambos sentidos: $c_{vw} = c_{wv}$ para todo vértice v, w pertenecientes a V.
  Esta característica hace que se trate de un grafo no dirigido, por lo que para esta versión del problema, es indiferente si el ciclo se recorre en un sentido o en el contrario, y por ello las rutas factibles para el problema se reducen a la mitad, quedando ((n-1)!)/2 ciclos posibles.

- TSP Asimétrico (ATSP).
  Es la versión contraria al STSP, es decir, dadas dos ciudades v y w, el coste de ir de v a w no tiene por qué ser el mismo que el que supone ir de w a v. Este problema es el que resolvemos en este documento por lo que lo detallaremos en el siguiente apartado.

1.2.5 Problema de Agente Viajero Asimétrico, ATSP

Para el Problema del Agente Viajero Asimétrico se puede dar el caso de que dadas dos ciudades, no existen caminos en ambas direcciones que las una directamente o incluso que la distancia o coste que hay de una ciudad a otra no sea la misma en un sentido que en el otro, (aunque también puede ser la misma). Con que esa desigualdad se de en un solo par de vértices ya se trata de un grafo dirigido, y por tanto el problema ya es asimétrico.

El ATSP es más genérico que el caso simétrico, de hecho, un método capaz de resolver el ATSP también puede resolver la versión simétrica ya que ésta es un caso concreto del asimétrico donde las posibles soluciones se reducen a la mitad. Para un problema ATSP de n ciudades, la cantidad de ciclos posibles es de (n-1)!.

Ante este tipo de problemas nos encontramos diariamente, cuando vamos a realizar un recorrido y nos topamos con un accidente de tráfico, calles transitables en un solo sentido, diferentes precios en los billetes de avión en función de si se entra en un país o se sale…

El ATSP puede ser resuelto mediante la conversión del mismo a un problema TSP simétrico, que resulta más fácil de manejar. Para ello una de las opciones es duplicar el tamaño de la matriz de modo que, teniendo una matriz de costes de tamaño n en el problema asimétrico, la convertimos en una matriz de costes simétrica de tamaño 2n como vemos en la figura 3.

Camino ponderado asimétrico

	A	B	C
A		1	2
B	6		3
C	5	4	

→

Camino ponderado simétrico

	A	B	C	A'	B'	C'
A				$-\infty$	6	5
B				1	$-\infty$	4
C				2	3	$-\infty$
A'	$-\infty$	1	2			
B'	6	$-\infty$	3			
C'	5	4	$-\infty$			

Figura 3. Conversión del ATSP al caso simétrico. [7].

El procedimiento para la conversión es duplicar cada ciudad del problema creando así un segundo nodo ficticio. A Los nodos duplicados se le asignan pesos muy bajos (usamos-∞, pero también puede ser cero, a no ser que exista alguna arista del

grafo original que valga 0, en cuyo caso tendría que ser un valor menor para permitir un salto libre entre un nodo y su ficticio) para que permita hallar rutas muy baratas que enlazan el nodo ficticio con el real.

En la actualidad existen muchos softwares libres que permiten resolver el TSP, pero en este proyecto no vamos a utilizar ninguno de ellos, sino que vamos a crear un programa propio, utilizando uno de los métodos de resolución de problemas de optimización, para encontrar una buena solución al ATSP.

El método que utilizaremos para crear nuestro programa junto con otros algoritmos adecuados para la resolución de problemas de optimización combinatoria serán explicaremos en el siguiente capítulo.

2. MÉTODOS DE RESOLUCIÓN DE PROBLEMAS

En primer lugar vamos a introducir los métodos principales utilizados para la resolución de un problema de optimización combinatoria, como es el caso de los NP-hard, y por tanto del Problema del Agente Viajero Asimétrico (ATSP), éstos son:

- Algoritmos exactos.
- Algoritmos de aproximación o métodos heurísticos.
- Métodos híper heurísticos.

El tercero de los tipos, denominado "híper heurísticos", se utiliza en algunas situaciones en las que un problema concreto no tiene una buena resolución ni mediante algoritmos exactos, ni con la heurística, pero sí se puede descomponer en varios casos especiales o "sub-problemas" para los cuales son posibles heurísticas mejores o incluso algoritmos exactos, es decir, lo que hace la Híper Heurística es crear nuevos algoritmos mediante la combinación de heurísticos ya conocidos, compensando las debilidades de unos con otros (Ross, 2005).

A continuación se explicara brevemente cada uno de los dos primeros tipos, para posteriormente enfocarnos en los algoritmos de aproximación, más concretamente en la metaheurística, de la cual forma parte el método que nos proporcionará la solución adecuada al ATSP en este artículo, el algoritmo de Optimización por Enjambre de Partículas (PSO). Finalmente se describirá en que consiste esa metaheurística que nos concierne, el PSO.

En la figura 4 que se muestra a continuación, se expone el esquema con la clasificación de los métodos de resolución que consideramos más importantes, desde el punto de vista de nuestro estudio, para resolver los problemas de optimización, marcando además a qué categorías corresponde el PSO.

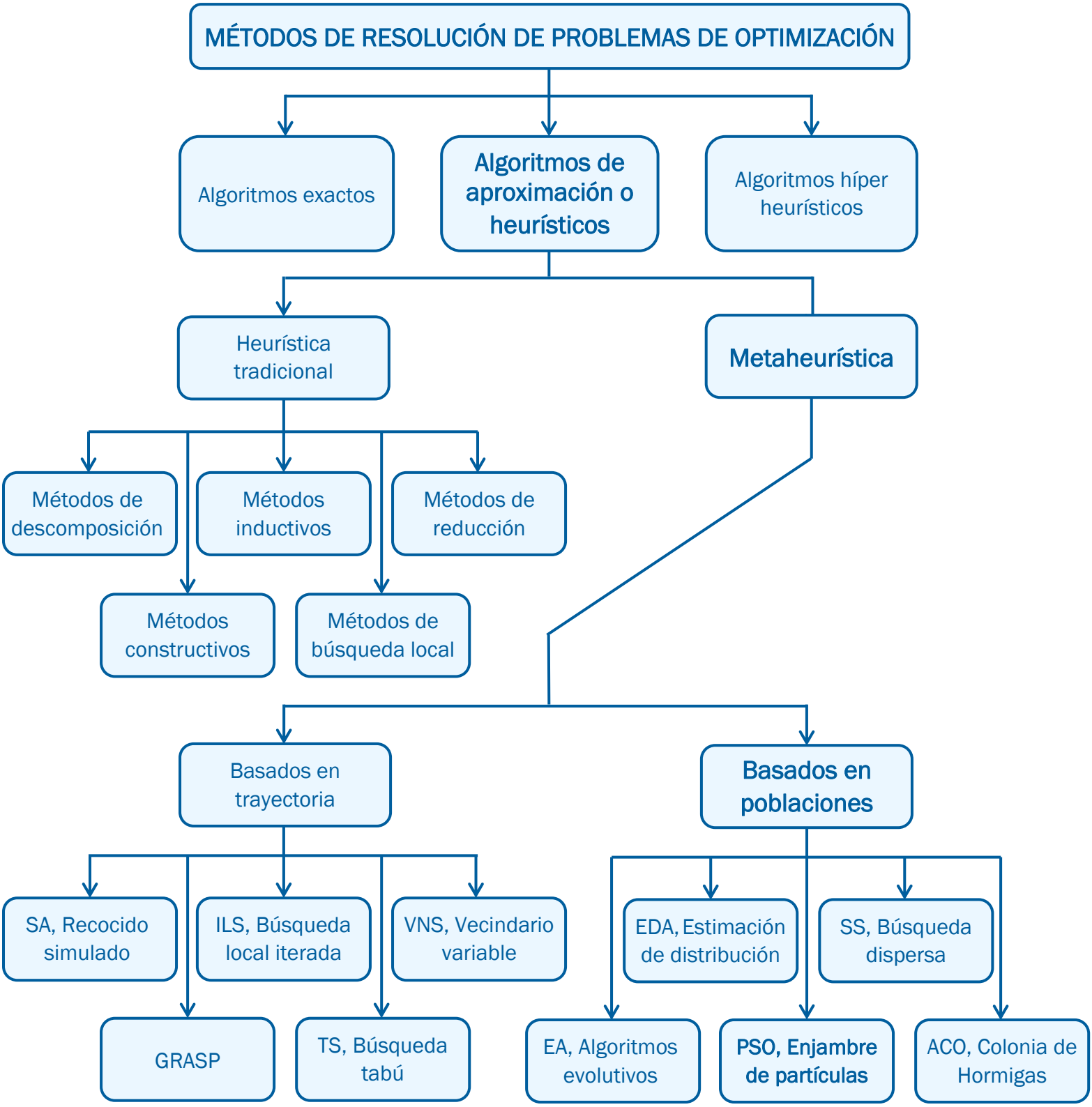


Figura 4. Métodos de resolución de problemas de optimización. Elaboración propia.

2.1 Algoritmos exactos

El algoritmo exacto se caracteriza por encontrar la solución óptima al problema propuesto. El hecho de que siempre proporcione como solución el óptimo global es una ventaja, mientras que el tiempo de ejecución de estos algoritmos es su principal desventaja ya que crece de forma exponencial con el tamaño del problema.

El tiempo invertido para resolver un problema NP-hard de optimización combinatoria mediante un método exacto, si es que existe el método, en la mayoría de los casos suele ser tan grande que resulta inaplicable. De hecho, este tipo de algoritmos es incluso inviable para un problema de 20 ciudades.

En este documento describiremos de forma muy breve este tipo de algoritmos puesto que la mayoría de los problemas ingenieriles no se pueden resolver mediante métodos exactos, y hay que recurrir a los aproximados que explicaremos más adelante.

Algunos de los algoritmos exactos más importantes se describen a continuación (Hoffman, 2000):

- Técnicas enumerativas.
  La solución más directa es intentar todas las posibles combinaciones del problema y hallar la función objetivo para cada una de las permutaciones, eligiendo la que logra el mejor resultado (menor o mayor valor de la función objetivo, dependiendo de si es un problema de minimización (caso del ATSP) o de maximización).
  El problema de esta solución, es que el tiempo de ejecución es de orden (n!), siendo n el número de ciudades que habría que visitar, por lo que implica mucho tiempo. Hay veces que se puede reducir, en cierta medida, el tiempo computacional eliminando posibilidades debido a la dominación, que es en lo que se basa el algoritmo de ramificación y poda o ramificación y acotación (Guerequeta y Vallecillo, 1998). Este algoritmo consiste en realizar una enumeración parcial del espacio de soluciones posibles, mediante la generación de un árbol de expansión. El árbol se organiza de modo que cada nodo de nivel k representa una parte de la solución, formado por k etapas ya realizadas, y sus hijos son las prolongaciones posibles al añadir una nueva etapa. El proceso de poda consiste en calcular una cota (función) para cada posible prolongación y si dicha cota es peor que la de la mejor solución hallada hasta ahora, no se explora más esa rama (la poda entera).
  El método se realiza en 3 etapas: Selección del nodo entre el conjunto de los nodos vivos (aquellos que no han sido ya podados), Ramificación (construcción de los posibles hijos del nodo seleccionado) y Poda.

El algoritmo finaliza cuando se agota el conjunto de nodos vivos, lo cual garantiza que el que queda es el mejor del problema, puesto que todos los que han sido podados tenían una cota que la de éste.

- Enfoques de relajación y descomposición.
  Otra de las técnicas de acotación muy conocidas (con mucho éxito hasta que apareció la de acotación y poda) es la relajación Lagrangiana donde se busca eliminar las restricciones malas, poniéndolas en la función objetivo como penalización (al ponerlas como penalización también se evita infringirlas para obtener un mejor resultado. El peso de ellas depende del multiplicador Lagrangiano que se le aplique). Moviendo esas restricciones a la función objetivo, el sub-problema resultante es más fácil de resolver. Es necesario resolver iterativamente el sub-problema hasta encontrar los valores óptimos para los multiplicadores.

- Planos de corte basados en combinaciones poliédricas.
  Se basa en la aplicación de la teoría poliédrica para resolver problemas numéricos. Teniendo en cuenta el teorema de Weyl, si podemos enumerar el conjunto de desigualdades lineares que definen la convexificación de los puntos posibles y los rayos extremos del problema, podremos resolver el problema de programación entera mediante programación linear.

Podemos decir que los métodos exactos son muy eficaces ya que logran la solución óptima, pero no son eficientes debido al excesivo tiempo que necesitan para ello. Es la segunda razón, la que provoca que los problemas de optimización combinatoria sean, en su inmensa mayoría, resueltos mediante algoritmos de aproximación.

2.2 Algoritmos de aproximación o heurísticos

El problema ATSP que tratamos, pertenece a la categoría NP-hard de optimización combinatoria (tienen como objetivo encontrar el máximo o mínimo de la función objetivo del problema de entre un conjunto de soluciones finito, teniendo en cuenta que sus variables de decisión son discretas). Los problemas NP-hard son considerados “difíciles de resolver” y en términos algorítmicos podemos decir que eso significa que no se puede garantizar encontrar el óptimo global (mejor solución de entre todas las posibles) para un tiempo de ejecución razonable, por ello los métodos exactos no son eficientes para este tipo de problemas, lo cual da paso a los algoritmos de aproximación.

Los algoritmos de aproximación o heurísticos (proveniente de la palabra griega “heuriskein” que significa encontrar o descubrir) son utilizados en problemas donde

la rapidez de la ejecución es tan importante como la calidad de la solución obtenida, puesto que encuentran un equilibrio razonable, dando una buena solución (aunque no sea necesariamente la óptima) en un tiempo de cómputo moderado. Según Díaz y otros (1996) una buena definición de algoritmo heurístico es la siguiente (Martí, 2003):

> *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 los últimos años ha habido un gran desarrollo de los métodos heurísticos enfocados a resolver problemas de optimización, habiéndose publicado muchos artículos relacionados con la materia en revistas especializadas como por ejemplo la *"Journal of Heuristics"* e *"INFORMS Journal on Computing",* además de la publicación de un importante número de libros.

Hay varias razones para utilizar los procedimientos heurísticos:

- Hay algunos problemas para los cuales no se conoce ningún método exacto que pueda resolverlos.

- En algunos casos en los que sí existe el método exacto, su coste computacional es tan grande que lo hace inviable.

- La flexibilidad de los métodos aproximados es mucho mayor que la de los exactos dejando, por ejemplo, agregar condiciones de modelización complicada.

- El método aproximado suele ser parte de un procedimiento global que garantiza el óptimo del problema. Hay dos opciones:

    - El método aproximado proporciona una buena solución inicial de partida.

    - El método aproximado es un paso intermedio del procedimiento global, utilizándolo por ejemplo para seleccionar variables en el método Simplex.

Los algoritmos heurísticos dependen del problema concreto para el que se diseñan, cosa que no ocurre por ejemplo con el algoritmo exacto de ramificación y poda cuyo procedimiento esta prefijado y es prácticamente independiente del problema para el que se utiliza. A pesar de que los métodos aproximados dependen del problema determinado, pueden ser trasladados a otros particularizando algunas técnicas e

ideas específicas del algoritmo. Por ello para aplicar un método heurístico hay que especificar el problema concreto a tratar.

En el caso concreto de este artículo, el problema de referencia que tenemos en cuenta a la hora de explicar los métodos es el Problema del Viajante (TSP).

2.2.1 Calidad de un algoritmo heurístico

Un buen algoritmo heurístico debe poseer las siguientes características (Martí, 2003):

- Eficiencia: Que el esfuerzo computacional sea realista y acorde a la calidad de la solución obtenida.

- Bondad: La solución que ofrece debe ser lo suficientemente buena, es decir, que esté cerca del óptimo global del problema.

- Robustez: Debe de tener una probabilidad muy baja de obtener una mala solución (alejada del óptimo).

Hay varios modos de medir la calidad del algoritmo con el que trabajamos:

- Compararlo con una solución óptima aplicándolo a un problema cuya solución óptima se conoce (aunque la colección es reducida). Para hacer esta comparación se mide la desviación porcentual del valor que nos ofrece nuestro método heurístico al aplicarlo al problema y el óptimo del mismo problema. Ésta medida se hace para varios ejemplos y se halla la el promedio de las desviaciones.

- Compararlo con una cota en los casos en que no disponemos ni siquiera de un conjunto limitado de ejemplos del problema del que conozcamos el óptimo. La bondad de esta medida, obviamente, depende de lo buena que sea la cota del problema escogida, es decir, de la cercanía de la cota al óptimo del problema. Por ello si no tenemos información de lo buena que es la cota esta medida carece de interés.

- Compararlo con un método exacto truncado, como por ejemplo con el de Ramificación y Acotación si le establecemos un límite de iteraciones, un tiempo máximo de ejecución, o algún otro criterio de parada que evite el problema de tiempo de estos métodos exactos. Aunque sabemos que así lo más probables es no lograr el óptimo, el valor obtenido nos sirve de cota para hacer la comparación con nuestro algoritmo heurístico.

- Compararlo con otros algoritmos heurísticos. Ésta comparación es de las más utilizadas para los problemas NP-hard ya que para ellos son conocidos buenos heurísticos. La efectividad de esta comparación depende de lo bueno que sea el otro método heurístico con el que nos estamos comparando.

- Analizar el comportamiento de nuestro algoritmo en el peor de los casos, es decir, ponerlo en práctica con los ejemplos más desfavorables para el algoritmo y acotar, de manera analítica, la máxima desviación respecto al óptimo. El problema de este método es que al coger el peor de los casos los resultados no son representativos del comportamiento medio del algoritmo.

2.2.2 Clasificación de los métodos heurísticos o aproximados

Debido a la gran variedad de métodos heurísticos existentes y a su diversa naturaleza (algunos incluso diseñados para problemas concretos sin opción de generalización), es muy difícil realizar una clasificación completa de los mismos. Un modo de clasificación es el que damos a continuación (Martí, 2003), formado por varias categorías no excluyentes:

- Métodos de descomposición: Se trata de descomponer el problema en sub-problemas más fáciles de resolver, sin olvidar a la hora de tratarlas, que todas esas divisiones pertenecen al mismo problema.

- Métodos inductivos: Este tipo de algoritmos usa la idea “opuesta” al grupo anterior, puesto que trata de generalizar versiones sencillas al caso completo. Para ello utiliza propiedades o técnicas válidas para casos de fácil análisis, en problemas más completos.

- Métodos de reducción: Trata de identificar e introducir como restricciones del problema algunas propiedades que las buenas soluciones cumplen. Con ello se simplifica el problema, puesto que se acotan las soluciones posibles. Debido a esa característica, este tipo de métodos tiene el riesgo de restringir de tal modo que deje fuera el óptimo del problema original.

- Métodos constructivos: Estos métodos construyen paso a paso una solución del problema tratado, añadiendo en cada paso un elemento hasta completar una solución. Se trata de métodos iterativos y deterministas que seleccionan la mejor elección en cada iteración del algoritmo. Son

ampliamente utilizados en problemas clásicos como el del viajante que nos concierne.

- Métodos de búsqueda local: Estos métodos parten de una solución del problema, al contrario que los constructivos que la van creando poco a poco. A partir de esa solución inicial, van mejorando progresivamente, obteniendo en cada paso una nueva solución, producto de un movimiento de la solución anterior a la nueva, con mejor valor. El método se da por terminado cuando para la solución existente no se encuentra ninguna otra solución accesible que mejore el valor actual.

Los métodos mencionados hasta ahora forman parte de los heurísticos tradicionales. En los últimos años ha aparecido una nueva categoría, los métodos Metaheurísticos, que usan como base los métodos de búsqueda local junto con los constructivos.

- Métodos Metaheurísticos: El término Metaheurístico fue introducido por Fred Glover en 1986 al definir una clase de algoritmos de aproximación que combinan heurísticos tradicionales con estrategias eficientes de explotación del espacio de búsqueda. Estos métodos son más recientes y complejos que los heurísticos clásicos y surgieron para mejorar los resultados obtenidos por éstos.

Puesto que nuestro estudio trata de un método de éste último tipo, en los siguientes apartados vamos a describir brevemente los métodos constructivos y los de búsqueda local, en el enfoque del TSP, por ser la base de la metaheurística y posteriormente profundizaremos en ella para finalmente llegar al algoritmo PSO que trataremos en el siguiente capítulo.

2.2.3 Métodos constructivos

En este apartado trataremos los procedimientos constructivos aplicados a nuestro problema específico, el TSP.

Se trata de procedimientos iterativos que seleccionan y añaden un elemento (el que mejor evaluación posee de todos los posibles) en cada paso hasta obtener la solución completa.

A continuación detallaremos de forma breve cuatro de los métodos constructivos más utilizados para resolver el problema del Viajante, los cuales están descritos en la revisión realizada por Jünger, Reinelt y Rinaldi (1995):

- Heurísticos del Vecino más Próximo.
  Éste procedimiento se lo debemos a Rosenkrantz, Stearns y Lewis (1977) y consiste en la construcción de un ciclo Hamiltoniano (ciclo que pasa exactamente una vez por cada vértice, salvo por el vértice del que parte ya que también es al que llega al final del ciclo) de coste reducido, buscando el vértice más cercano al vértice en que te encuentras.
  Este algoritmo presenta el problema denominado "miopía del procedimiento", lo cual significa que empieza muy bien escogiendo aristas de bajo coste pero no ve más allá de lo que escoge en cada momento, y por tanto no se preocupa de lo que va quedando, haciendo que cuando el procedimiento va evolucionando, las aristas que queden para elegir serán cada vez menos y probablemente su conexión requiera un coste muy elevado. Además la implementación de este método para ejemplos de gran tamaño es muy lenta.
  Para disminuir la miopía y el tiempo computacional del algoritmo se utiliza "subgrafo candidato", es decir, un subgrafo del grafo completo del problema en el que se representan todos los vértices (n), pero sólo se ponen las aristas (k) que resultan atractivas (las que suponen un bajo coste, en este caso, las que unen el vértice con sus vecinos más cercanos).
  De este modo se selecciona el próximo vértice a unir examinando solamente los adyacentes al vértice actual en el subgrafo candidato y en caso de que todos estén ya metidos en la ruta establecida hasta el momento, se examinan todos los vértices restantes (cuando un vértice se mete en la ruta se eliminan las aristas que inciden en él del subgrafo candidato debido a que no se puede pasar dos veces por la misma ciudad). Además se establece un número s (menor que k) que sirve para evitar aislamientos, es decir, cuando hay un vértice que no está aún incluido en el tour y está conectado a s o menos aristas en el subgrafo, se añade al tour para evitar que quede aislado y suponga un coste muy elevado al insertarlo en la solución.

- Heurísticos de Inserción.
  Este tipo de algoritmos comienzan creando ciclos que visitan únicamente ciertos vértices y poco a poco los amplían añadiendo el resto de los vértices hasta crear un ciclo Hamiltoniano.
  Este procedimiento fue desarrollado por los mismos autores que el del vecino más próximo y se ha comprobado experimentalmente que el ciclo inicial no condiciona mucho la solución del problema.
  Existen varios criterios de inserción del vértice: Inserción más cercana (meter el vértice j que más cerca esté del ciclo, sabiendo que la distancia del vértice al ciclo es la mínima distancia que hay desde ese vértice a cualquiera de los vértices del ciclo), inserción más lejana (se selecciona el vértice más lejano al ciclo, calculando la distancia del modo anterior),

inserción más barata (coger el que supone el coste más reducido) e inserción aleatoria (se selecciona al azar).

- Heurísticos Basados en Árboles Generadores.
  Este heurístico se basa en árboles generadores de coste mínimo, que posee información sobre la estructura del grafo, además de la información de coste que ya añadían los ciclos Hamiltonianos.
  Prim creó en 1957 el algoritmo que obtiene un árbol generador de mínimo peso (grafo donde todo par de vértices están unidos mediante un camino y que no posee ciclos, además al ser el de mínimo peso tiene que ser el árbol generador, de todos los posibles del grafo, que menos coste supone) y Christofides es quien, en 1976, creó el algoritmo que lleva su nombre y busca un camino Hamiltoniano de costo bajo en un grafo. Los pasos principales del algoritmo de Christofides son: Calcular un Árbol Generador de Peso Mínimo, obtener un conjunto de vértices (impares) del mismo y obtener un acoplamiento perfecto de peso mínimo sobre ellos, posteriormente añadir al árbol las aristas del acoplamiento y realizar el procedimiento de obtención del Tour (formar un ciclo de Euler y saltando los vértices visitados transformarlo en un ciclo Hamiltoniano).

- Heurísticos Basados en Ahorros
  Los fundadores de este método son Clarke y Wright (1964), cuya propuesta original fue para problemas de rutas de vehículos pero nosotros lo veremos adecuado al problema del viajante.
  En este método se crea un ciclo Hamiltoniano mediante la combinación de subtours, que comparten un vértice llamado base.
  El procedimiento consiste en crear los subtours y luego eliminar las aristas que conectan dos vértices de diferentes subtours con el vértice base, para después conectar dichos vértices entre sí, uniendo así los dos subtours (se hace con las que más ahorro proporcionen (entendiendo por ahorra, la diferencia que hay entre las aristas que se han eliminado respecto a la que se ha añadido). En este tipo de problemas también se puede usar el subgrafo candidato para mejorar los resultados del algoritmo y acelerar la ejecución.
  Podemos ver un ejemplo gráfico del método en la figura 5, donde el vértice base es el z y se eliminan las uniones del mismo con i y j para introducir posteriormente la nueva arista (i,j).

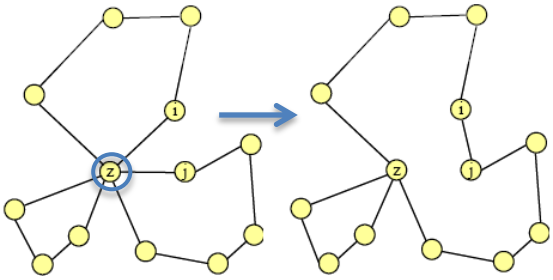


Figura 5. Heurísticos Basados en Ahorros. Martí, 2003.

A continuación se muestra en la tabla 2 la comparativa de los 4 métodos anteriormente descritos en la que se especifica la desviación de la solución obtenida al resultado óptimo y el tiempo computacional del algoritmo. El tiempo de ejecución está dado sobre el ejemplo de mayor tamaño de la librería TSPLIB (pr2392) debido a que al tratarse de métodos sencillos su tiempo es reducido.

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

Tabla 2. Resultados de los métodos constructivos. Martí, 2003.

Podemos apreciar que el método de los ahorros y el de inserción del elemento más lejano son lo que mejor resultado proporcionan, aunque también son más lentos.

2.2.4 Métodos de búsqueda local

Los métodos de búsqueda local, también llamados métodos de mejora, suelen obtener mejores soluciones que los constructivos. Lo que hacen es partir de una solución inicial y examinar sus alrededores en busca de otra solución mejor a la actual. Formalmente podemos decir:

Para un conjunto de soluciones X del problema, cada una de las soluciones x tiene asociado un subconjunto, perteneciente a X, que es el entorno o vecindad de x, N(x). Dada la solución x, cada solución de su vecindad x' (que pertenece a N(x)) se obtiene directamente aplicando un operación, llamada movimiento, a x.

De manera práctica, lo que hacemos es partir de una solución inicial x_0 de la que calculamos la vecindad N(x_0) y aplicando el movimiento m_1, obtenemos x_1

perteneciente al entorno. Este proceso se aplica iterativamente creando una trayectoria hasta que se satisfaga la condición de parada.

En este tipo de métodos es importante determinar cómo definimos el entorno N(x) y qué criterio tenemos para seleccionar la nueva solución (perteneciente al entorno). La especificación de la vecindad varía en función del problema que se va a resolver y en cuanto a los criterios de selección, existen varios. Uno de los más comunes es el de greedy, que consiste en coger de entre toda la vecindad, aquella solución que mejor valor tiene en la función objetivo del problema, siendo este valor mejor que el de la solución actual. Si no existe ninguna solución del entorno que posea un mejor valor (condición de parada), se detiene el algoritmo y la solución actual será la solución del algoritmo.

Este método no asegura obtener el óptimo global del problema a causa de la "miopía" debida a que solo se examina el entorno establecido, por lo que si dentro del entorno no hay una solución mejor, no irá más allá y dejará el resto del espacio sin examinar. Esto provoca que lo que lo único que se puede asegurar es que la solución hallada será el óptimo local respecto al entorno establecido.

El inconveniente de la miopía es una de las cosas que busca solucionar la metaheurística basada en la búsqueda local que detallaremos más adelante. Para ello es necesario en ocasiones ir a una solución peor, lo cual tiene algunos inconvenientes: nos podemos meter en un proceso cíclico al coger a veces soluciones mejores y otras veces peores y además el problema iteraría sin fin, por lo que habría que establecer una condición de parada diferente.

Los algoritmos de búsqueda local varían dependiendo del problema que se trate. A continuación describiremos brevemente tres de los procedimientos más utilizados para resolver el TSP según Jünger, Reinelt y Rinaldi (1995):

- Procedimientos de 2 intercambio.
  Está basado en el hecho de que se pueden acortar los ciclos Hamiltonianos cuando dos aristas del mismo se cruzan, eliminando las dos aristas secantes y conectando los dos caminos que resultan mediante aristas que no se crucen, obteniendo así un ciclo final más corto que el de origen. En la figura 6 se puede ver como se acorta la distancia del ciclo eliminando las aristas (i,j) y (l,k) y poniendo (i,k) y (l,j).

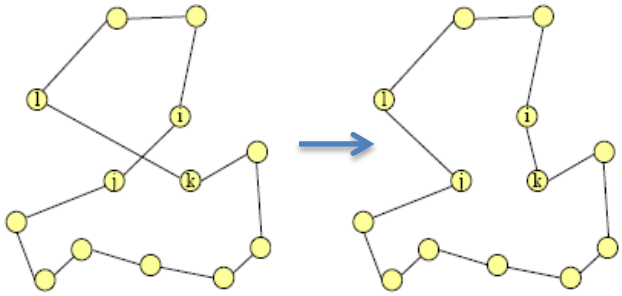


Figura 6. Procedimiento de 2 intercambio. Martí, 2003.

El movimiento 2-opt sigue este principio, de modo que lo que hace es eliminar dos aristas del ciclo y volver a unir los dos caminos que se obtienen de distinto modo, obteniendo así un nuevo ciclo.
El algoritmo heurístico de mejora 2-óptimo examina vértice a vértices y estudia todos sus posibles movimientos 2-opt, realizando el mejor de ellos (en caso de que éste reduzca la distancia del ciclo, de lo contrario sobre ese vértice no se realiza ningún movimiento), esto se repite de forma iterativa hasta que no existan movimientos 2-opt, que mejoren el ciclo, para ninguno de los vértices del problema.
Este método tiene un alto coste computacional por lo que se proponen dos medidas para su reducción: utilizar el subgrafo candidato (una de las aristas añadidas en el movimiento, como mínimo, tiene que pertenecer al subgrafo) y el consejo de interrumpir el algoritmo antes de que acabe (las primeras iteraciones mejoran mucho, pero las últimas apenas varían la distancia). Estas acciones suponen perder la garantía de llegar a un óptimo local, por lo que su aplicación dependerá de lo crítico que sea el tiempo de ejecución.

- Procedimientos de k – intercambio.
  La diferencia con el anterior es que en este caso el ciclo Hamiltoniano se divide en k partes (en vez de 2) y como antes, se reagrupan los caminos que quedan de la manera más beneficiosa, siendo esta modificación el movimiento k-opt. Cuanto mayor es k, más tiempo computacional lleva implementar el algoritmo.
  Nosotros vamos a ver el caso 3-opt. Un movimiento 3-opt consiste en eliminar 3 aristas y reconectar los 3 caminos que resultan, para ello hay 8 maneras de conectarlos diferentes que hay que analizar por lo que este algoritmo es costoso.
  Por ello se restringe el problema, definiendo para cada vértice i un conjunto N(i) de vértices (por ejemplo los contiguos a i), de manera que cuando se estudian los movimientos 3-opt para una arista (i,j), solo se tienen en consideración los casos en los que las otras 2 aristas poseen, al menos, uno de sus vértices dentro de N(i).

Los pasos del algoritmo son los mismos que para el caso 2-opt cambiando el movimiento 2-opt por el 3-opt y se añadiendo la restricción del conjunto de vértices N(i).

- Algoritmo de Lin y Kernighan.
  Este algoritmo tiene el mismo principio de funcionamiento que el 2-opt y 3-opt, de modo que hace movimientos que mejoran el ciclo Hamiltoniano inicial del que parte, hasta que llega a un óptimo local del problema.
  La diferencia principal con los anteriores es que aquí se utilizan movimientos compuestos, formados por movimientos consecutivos simples, que pueden mejorar o empeorar la solución por sí mismos, pero el conjunto del movimiento completo sí que mejora. Este algoritmo tiene versiones muy diferentes, dependiendo de los movimientos sencillos que formen el movimiento compuesto.
  En la figura 7 vemos un ejemplo donde el movimiento compuesto es: dos movimientos 2-opt y un movimiento de inserción.

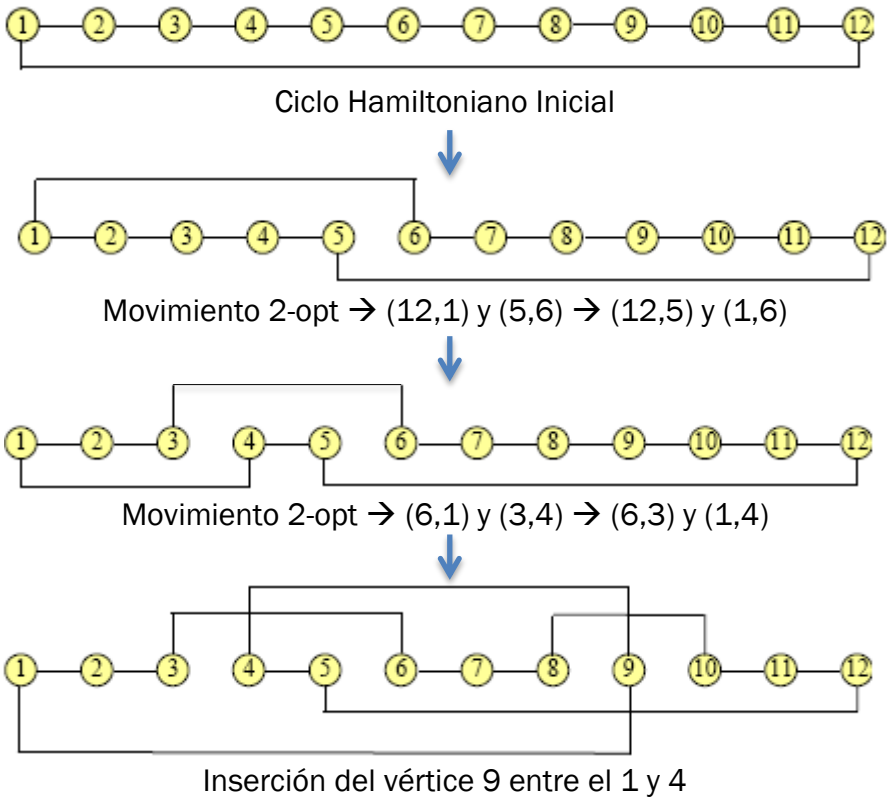


Figura 7. Algoritmo de Lin y Kernighan. Martí, 2003.

Debido al gran número de combinaciones posibles para realizar los movimientos, hay que restringir los movimientos a analizar en cada paso. Dos variantes de hacerlo son:

- Analizar los movimientos pertenecientes a un subgrafo candidato que contiene las aristas de los 6 vértices más cercanos al nodo. Se aceptan movimientos compuestos formados, como mucho, por 15 movimientos simples teniendo que ser todos del tipo 2-opt o inserción. Y por último, en el primero de los movimientos simples sólo se analizan 3 candidatos.

- En este caso el subgrafo candidato tiene en cuenta las aristas relativas a los 8 vecinos más cercanos. Al igual que el anterior el movimiento compuesto puede tener hasta 15 simples 2-opt o de inserción. Para finalizar, en los 3 primeros movimientos simples sólo se examinan 2 candidatos.

En la tabla 3 que se muestra a continuación se comparan los métodos anteriores (la solución inicial se ha hallado con el método del vecino más próximo) especificando en promedio la desviación respecto al resultado óptimo (se han utilizado los 30 ejemplos de la librería TSPLIB) y el tiempo de ejecución del algoritmo (para el ejemplo pr2392 de la librería).

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

Tabla 3. Resultados de los métodos de búsqueda local. Martí, 2003.

Podemos observar como pasar del método 2-óptimo al 3-óptimo incrementa mucho el tiempo computacional, cosa que también ocurre cuando en el método Lin y Kernighan pasamos de un subgrafo de 6 vecinos a uno de 8. También podemos ver que la desviación del óptimo para los casos de movimientos compuestos es mucho más eficiente que utilizar solo movimientos simples que mejoran el resultado (k-óptimo).

Teniendo en cuenta ambos factores el método más eficiente parece el de Lin y Kernighan para la variante de un subgrafo de 6 vecinos, puesto que el tiempo de ejecución es 3 veces menor y la diferencia de calidad de la respuesta es muy pequeña.

2.2.5 Métodos Metaheurísticos

Los procedimientos metaheurísticos son un tipo de procedimiento heurístico que surgen en los años 70 (Luque, G. 2006), bajo la idea de combinar varios métodos heurísticos a un nivel superior para lograr una eficiente y efectiva exploración del espacio de búsqueda, aunque no se les conoce así hasta el año 1986, cuando Glover les otorga el nombre de metaheurísticos por el que ahora los conocemos.

Hasta ese momento eran conocidos como heurísticos modernos. Por ello, aunque en algunos artículos se siguen utilizando los términos "heurísticos modernos" y "heurísticos clásicos", en la mayoría se utiliza "heurística" para denominar a la heurística clásica y "metaheurística", para hacer referencia a este tipo específico de procedimientos heurísticos, que son más recientes y complejos, y cuya definición puede ser esta que nos ofrecen (Martí, 2003) Osman y Kelly en 1995:

> *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.*

El prefijo "meta" quiere decir "más allá" o "a un nivel superior", denotando por tanto, que este tipo de métodos construyen algoritmos que van más allá de la heurística clásica. Se trata de inteligentes estrategias generales de diseño que crean o perfeccionan, con elevado rendimiento, procedimientos heurísticos generales con el objetivo de resolver un problema de forma eficiente.

Una definición formal de los elementos que intervienen en los métodos metaheurísticos es (Luque, G. 2006):

$$\mathcal{M} = < T, \mu, \lambda, \Theta, \Phi, \sigma, \Xi, \tau >$$

Donde:

- T es el conjunto de elementos que manipula la metaheurística, que suele coincidir con el espacio de búsqueda.

- μ es la cantidad de estructuras (soluciones) con las que trabaja la metaheurística.

- λ Indica el número de nuevas estructuras, generadas en cada iteración de la metaheurística, resultantes de aplicar Φ a Θ.

- $\Theta = \{ \theta_1,\ldots, \theta_\mu\}$ son las variables que almacenan las soluciones con las que trabaja la metaheurística $\mathcal{M}$, donde todas las θ_i del conjunto Θ pertenecen a T.

- Φ: T^μ x $\Xi \rightarrow \mathrm{T}^\lambda$ es el operador responsable de la modificación de las estructuras.

- σ: $\mathrm{T}^{\mu+\lambda}$ x $\Xi \rightarrow \mathrm{T}^\mu$ función responsable de la selección de las nuevas estructuras que se tratarán en la siguiente iteración de la metaheurística.

- $\Xi = \{\xi_1, \xi_2,\ldots\}$ variables de estado que poseen información sobre la evolución del algoritmo. El conjunto tiene que tener mínimo un elemento $\xi_1 = \theta^*$, que es la mejor solución encontrada del problema.

- τ: Θ x Ξ {true, false} función que determina el final del algoritmo (criterio de parada).

El funcionamiento de cualquier metaheurística $\mathcal{M}$ se podría definir de manera formal de la siguiente manera:

$$\Theta_{i+1} = \begin{cases} \{\theta^*\}^\mu & si\ \tau(\Theta_i \text{ x } \Xi) \rightarrow \text{fin} \\ \sigma\big(\Phi(\Theta_i \text{ x } \Xi)\big) & \text{en otro caso} \end{cases}$$

2.2.5.1 Características de los métodos Metaheurísticos

Los métodos metaheurísticos poseen ciertas características fundamentales (Luque, 2006), de entre las que destacamos las siguientes:

- Estos métodos son plantillas o estrategias generales que lo que hacen es guiar el proceso de búsqueda, con el objetivo de realizar eficientemente la exploración del espacio de búsqueda del problema para lograr un resultado óptimo o bastante aproximado a éste.

- Son algoritmos generalmente no deterministas y además son aproximados, por lo que no se puede tener la absoluta certeza de que la solución que te proporcionan sea el óptimo global del problema (la cual sí que obtienes en los algoritmos exactos).

- Pueden agregar restricciones que evitan explorar los espacios de búsqueda no óptimos.

- Los métodos metaheurísticos no son dependientes del problema para el cual se van a utilizar, sino que su estructura básica es general para todo problema.

- Son procedimientos que se pueden transformar fácilmente en algoritmos paralelos (ejecución de varias partes del algoritmo al mismo tiempo, lo contrario a los secuenciales), lo cual permite una notable reducción del tiempo computacional.

- Los metaheurísticos controlan una serie de heurísticos específicos que sí que hacen uso de las características del problema para el que se diseñan, puesto que la metaheurística es una estrategia de nivel alto, que para explorar el espacio de búsqueda se vale de diferentes métodos.

- Dado que la metaheurística debe ser válida para afrontar problemas con espacios de búsqueda de grandes dimensiones, es muy importante que exista un equilibrio dinámico entre la exploración del espacio (búsqueda en regiones distantes del espacio, para poder diversificar su estudio) y la explotación de las áreas concretas (esfuerzo empleado en la búsqueda dentro de la región actual, para poder hacer un estudio suficientemente intenso). Este equilibrio permite identificar rápidamente regiones potenciales de espacio de búsqueda global, a la vez que evita perder tiempo en estudiar regiones ya exploradas o en aquellas cuyas soluciones no son de alta calidad.

De los algoritmos metaheurísticos se esperan las siguientes propiedades [9]:

- Fácil comprensión:

    - Simplicidad en cuanto a la comprensión del método, que debe ser claro y sencillo.

    - Precisión y concretización en los pasos a seguir para crear el algoritmo, con el objetivo de permitir una fácil implementación del mismo.

    - Coherencia entre los elementos que forman la metaheurística y los principios de la misma.

- Alto rendimiento:

- Efectividad, eficacia y eficiencia. Debe proporcionar soluciones de alta calidad, tener una elevada probabilidad de llegar a ellas y aprovechar al máximo los recursos de que dispone (tiempo de computación y memoria). Para evaluar estas propiedades hay que ejecutar una cierto número de problemas con soluciones conocidas, teniendo en cuenta tanto la desviación al óptimo, como el tiempo de ejecución empleado.

- Aplicación amplia:

  - Utilidad general de la metaheurística, que permita que el algoritmo sirva para muchos problemas diversos.

  - Adaptabilidad a diferentes entornos o variaciones del modelo a aplicar.

  - Robustez suficiente para que el resultado del problema a penas varíe ante modificaciones del modelo o entorno al que se aplica.

- Utilidad del método:

  - Interactividad que permita que el usuario pueda ir modificando partes del proceso para mejorar su rendimiento.

  - Multiplicidad en las soluciones proporcionadas. Debe proporcionar varias soluciones cercanas al óptimo, con el objetivo de que el usuario escoja la que más le conviene.

  - Autonomía en su funcionamiento para evitar que el usuario este pendiente del programa durante su ejecución.

2.2.5.2 Clasificación de los métodos Metaheurísticos

No existe un modo único de clasificar las técnicas metaheurísticas, puesto que depende de la propiedad que tengamos en cuenta.

Algunas de las clasificaciones que se utilizan hoy en día están ligadas a las siguientes características: si se fundamentan en la naturaleza, si poseen memoria, si poseen una o varias estructuras de vecindario, la naturaleza de la función

objetivo, si se en cada iteración se trata un solo punto del espacio de búsqueda o si por el contrario se trabaja con una población (conjunto de "puntos")...

En nuestro caso hemos escogido la última de las clasificaciones enumeradas anteriormente, puesto que creemos que es la más utilizada en los artículos científicos, obteniendo por tanto dos grupos diferenciados (Luque, 2006):

- Metaheurísticas basadas en la trayectoria.
  Estas técnicas están, en su mayor parte, basadas en los métodos de búsqueda local con el añadido de que intentan escapar de esos óptimos locales para mejorar su resultado.
  En este tipo de métodos tomamos un punto como partida del problema y en cada paso del algoritmo se exploran los vecinos del mismo, y se va actualizando el punto actual creando así una trayectoria en el espacio de búsqueda del problema.
  Se termina la búsqueda cuando se alcanza la condición de parada que puede ser: un número máximo predefinido de iteraciones, alcanzar una calidad aceptable, detectar un estancamiento del proceso...
  Teniendo en cuenta la definición formal dada en el apartado 1.2.5, este tipo de metaheurísticas se caracterizan porque el elemento μ es 1, es decir, que se trabaja con una sola estructura, en lugar de usar una población de una cantidad μ de estructuras (también llamadas partículas o individuos de la población, dependiendo del método).
  A continuación se describen brevemente las metaheurísticas clasificadas dentro de este conjunto:

  - SA, Recocido, Templado, o Enfriamiento Simulado
    Se trata de una de las metaheurísticas más antiguas que se conocen y probablemente la primera que trata de evitar los mínimos locales, permitiendo para ello escoger soluciones cuyo valor de la función objetivo es peor que el que te proporciona la solución actual.
    Su funcionamiento es una simulación del proceso de recocido de los cristales y metales, y lo que hace es estudiar el vecindario de la partícula actual y, mediante un determinado criterio, escoger uno de sus vecinos. Posteriormente, si ese vecino proporciona un mejor fitness que el actual, directamente se sustituye, y si el fitness que proporciona es peor, también se realiza la sustitución pero con una determinada probabilidad que depende de la variación de fitness que provocaría y de la temperatura (la probabilidad es una analogía del proceso físico de enfriamiento, por lo que cuanto más avanzan las iteraciones (nos vamos acercando al óptimo), menos temperatura hay, y por tanto menos probabilidad de escoger un fitness peor).

Tomando como referencia la definición formal del apartado 1.2.5, en esta metaheurística tanto μ como λ tienen el valor de 1 (se utiliza una sola partícula o estructura), Ξ almacena la mejor solución encontrada y la temperatura, Φ determina la elección del vecino y σ determina si ese vecino sustituye al actual o no.

- ILS, Búsqueda Local Iterada
  Se trata de un método muy genérico en el cual en primer lugar se genera una solución inicial a la que se le aplica un método de búsqueda local para mejorarla, obteniendo así la solución inicial definitiva y posteriormente en cada iteración se realiza una perturbación sobre la solución actual y se le aplica un método de búsqueda local. Para terminar la iteración, si el nuevo candidato (ya perturbado y mejorado) pasa unos criterios de aceptación determinados (dependientes de la búsqueda histórica y del nuevo y actual candidato) se sustituye el actual por el nuevo candidato.
  El proceso más importante a la hora de determinar el funcionamiento de este algoritmo es el de la perturbación, puesto que no debe ser ni tan pequeño como para no ser capaces de salir de un óptimo local, ni tan grande como para provocar que caigamos en un método de búsqueda local reinicializado de forma aleatoria.
  Según la notación del apartado 1.2.5, μ y λ tienen el valor de 1, Ξ almacena la mejor solución encontrada y el histórico, Φ realiza la perturbación y su posterior búsqueda local y σ evalúa el criterio de aceptación determinando si el candidato pasa, o no, a ser el actual.

- VNS, Búsqueda con Vecindario Variable
  Es un algoritmo muy general y, por tanto, flexible y adaptable a posibles variaciones que se basa en ir cambiando entre diferentes vecindarios a lo largo de la búsqueda.
  En primer lugar se define un conjunto de vecindarios (k_{max} vecindarios) y después en cada iteración se realizan 3 fases que consisten en elegir aleatoriamente al candidato del vecindario N_k(s), siendo s el actual (comenzando por k=1), mejorar el candidato seleccionado mediante búsqueda local y finalmente si el resultado obtenido es mejor que el actual se realiza el movimiento (sustituyendo el actual por el candidato resultante de la búsqueda y volviendo a la primera fase con k=1) para esta nueva solución s, en caso de que el resultado no sea mejor, no se realiza sustitución y se vuelve a la primera fase, incrementando k (hasta que ésta alcance el valor de k_{max}).

De este modo conseguimos el equilibrio entre intensidad y diversidad, ya que la búsqueda local nos permite la explotación del área mientras que los diferentes vecindarios ayudan a la exploración del espacio de búsqueda.

Para la definición formal del apartado 1.2.5, las características de este método son: tanto μ como λ tienen el valor de 1 (se manipula una sola partícula), Ξ almacena la mejor solución encontrada y el vecindario utilizado k, Φ selecciona al vecino del k-ésimo vecindario y realiza sobre él la búsqueda local y σ determina si ese vecino sustituye al actual o no.

- GRASP, Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo

  Esta metaheurística fue desarrollada a finales de los 80 por Feo y Resende y se trata de una combinación de los métodos heurísticos constructivos y los de búsqueda local (apartados 1.2.3 y 1.2.4).

  El algoritmo se divide en dos etapas, la primera es la generación de soluciones utilizando el primero de los heurísticos mencionados y la segunda es la mejora de la solución anteriormente construida mediante el heurístico de búsqueda local.

  En la construcción de soluciones se parte de una solución parcial vacía s^p a la que se van añadiendo en cada paso diferentes componentes c. Los componentes son elegidos de forma aleatoria entre una lista restringida de candidatos (RCL), que está compuesta por los α mejores vecinos del conjunto $N(s^p)$. La variable α es elegida por el usuario y determina el funcionamiento del algoritmo (Los dos extremos: si α es 1 equivaldrá a un algoritmo voraz ya que siempre se añadirá el mejor c y si α es $N(s^p)$ la elección de c será totalmente aleatoria entre todo el vecindario).

  En la segunda etapa del algoritmo se mejora la solución generada en la fase anterior utilizando algoritmos de búsqueda local simples o complejos, dependiendo de la elección del usuario.

  El método GRASP, según la definición formal del apartado 1.2.5, utiliza tanto μ como λ con un valor de 1, Ξ almacena solamente la mejor solución encontrada, Φ genera una solución y sobre ella realiza la búsqueda local y σ sustituye la solución actual por la mejora encontrada.

- TS, Búsqueda Tabú.

  El origen de esta metaheurística se sitúa a finales de los años 70 de la mano de Glover. Se trata de una de las metaheurísticas más exitosas hasta el momento a la hora de resolver problemas

combinatorios y se basa en principios generales de Inteligencia Artificial como es el concepto de memoria.
Lo que hace es extraer información de lo ocurrido y actuar en consecuentemente, por lo que podemos decir que realiza una búsqueda inteligente. Para ello se utiliza una lista tabú (TL) con memoria a corto plazo que nos permite evitar los óptimos locales, a la vez de impedir buscar en regiones que ya han sido estudiadas. En dicha lista se almacenan los movimientos recientes y los descarta para posibles soluciones futuras.
En cada iteración del algoritmo se filtra el vecindario de la posición actual, N(s), con las soluciones excluidas (TL) obteniendo un vecindario que cumple la condición tabú, al cual se le añaden las soluciones que cumplen el denominado criterio de aspiración (el más común es permitir introducir soluciones que posean un fitness mejor que el mejor encontrado hasta el momento, aunque pertenezcan a la TL). De esta unión obtenemos el vecindario reducido N_a(s). Posteriormente escogemos el vecino que mejor fitness posee dentro del N_a(s), se actualiza la lista tabú añadiendo este nuevo movimiento y se reemplaza el vecino seleccionado por el actual.
Según la definición formal del apartado 1.2.5, en la búsqueda tabú μ vale 1, Ξ almacena la mejor solución encontrada y la lista tabú, Φ obtiene N_a(s) y σ determina cuál de los vecinos pertenecientes a N_a(s) sustituye al actual (el de mejor fitness).

- Metaheurísticas basadas en la población.
Como su propio nombre indica en este tipo de metaheurísticas manipulamos en cada iteración un conjunto de soluciones denominado población. La mayor diferencia que existe entre los diferentes métodos aquí clasificados radica en el modo de manipulación de la población que utilizan. A continuación se describen brevemente los más comunes (Luna, 2008):

  - EDA, Algoritmos de Estimación de la Distribución.
  El primer paso, como en la mayoría de los algoritmos, es crear una población inicial. Posteriormente, en cada iteración se selecciona un conjunto de individuos de la población original, se estudia su distribución probabilística y con ella se generan los nuevos individuos que formarán la nueva población.
  Muchos autores clasifican este método como una variación de los Algoritmos Evolutivos, que explicaremos a continuación, o incluso como un tipo de éstos, en el cual la fase de reproducción se hace de una forma especial (mediante la distribución de la probabilidad).

- SS, Búsqueda Dispersa.
  Este método se basa en el principio de que dadas dos soluciones, éstas se pueden combinar creando así una nueva solución que mejora las dos predecesoras.
  El primer paso del algoritmo es generar aleatoriamente la población inicial, luego con los elementos anteriores más representativos se genera un conjunto de referencia, RefSet (formado por soluciones con elevada diversidad y calidad).
  Posteriormente inician las iteraciones donde se crean varios subconjuntos de soluciones, S, a partir del RefSet, y las soluciones de estos subconjuntos son combinadas creando soluciones nuevas. Cuando ya tenemos creadas las nuevas soluciones, se mejoran mediante un algoritmo de búsqueda local elegido por el usuario y se actualiza el RefSet sustituyendo parte de las soluciones del conjunto de referencia inicial por algunas de las nuevas soluciones (solo las que mejoran a las iniciales).
  Al final de cada iteración se comprueba si RefSet converge y en caso de ser cierto, se vuelve a generar la población completa de forma aleatoria, se genera el conjunto de referencia (en este caso se tiene en cuenta tanto la población creada como el RefSet anterior) y se sigue iterando hasta que se alcance la condición de parada.
  Según la definición formal del apartado 1.2.5, en este caso Φ lo primero que hace es comprobar si converge RefSet, en caso afirmativo crea un nuevo conjunto de referencia y en caso contrario genera los subconjuntos, los combina y mejora los individuos resultantes de la recombinación anterior.

- EA, Algoritmos Evolutivos.
  Este tipo de algoritmos simula el comportamiento evolutivo de los seres en la naturaleza, que permite que se adapten a diferentes cambios producidos en su entorno.
  Se manipula un conjunto de individuos, donde cada uno de ellos constituye una solución del problema, y en cada iteración hace evolucionar a dicha población, obteniendo así una nueva más adaptada al entorno, lo que en nuestro caso significa que posee mejor valor de fitness.
  En primer lugar se crea, bien sea de forma aleatoria o con ayuda de un heurístico de construcción, una población inicial y se evalúa con la función objetivo (calculando por tanto su fitness), posteriormente en cada iteración se realizan tres fases diferenciadas.
  La primera es la selección de los padres, donde se crea una nueva población temporal de tamaño λ (población de padres), a partir de la

original de μ individuos, siguiendo las bases de la selección natural (prevalecen aquellos individuos que tienen mejor fitness).
En segundo lugar se lleva a cabo la reproducción durante la cual se realizan operaciones, con cierta probabilidad, sobre la población de padres (recombinación de parejas o cruce y luego la mutación), obteniendo así una nueva población, los hijos.
Para finalizar la iteración se evalúa el fitness de la población de hijos y se realiza el remplazo de la población original (Hay dos tipos: el que sólo tiene en cuenta a la población de hijos, y el que también tiene en cuenta a la antigua población, de modo que se mantienen a los mejores individuos originales y se sustituyen los peores por los mejores hijos).
El Φ de la definición del apartado 1.2.5 es, en este caso, el responsable de seleccionar a los padres entre la población original, posteriormente recombinarlos y luego mutarlos y σ se encarga de realizar el remplazo.
Este es un esquema general en el que se basan diferentes algoritmos como son la Programación Evolutiva (EP), las Estrategias Evolutivas (ES) y los Algoritmos Genéticos (AG), que son los más conocidos.

- PSO, Optimización por Enjambre de Partículas.
  Este método simula el modo en que los enjambres de abejas buscan el polen en la naturaleza (o el comportamiento de bandadas de aves y bancos de peces). Lo que hacen es sobrevolar el espacio buscando el área con mayor densidad de flores, teniendo en cuenta que siempre recuerdan el lugar donde ellas han visualizado la mayor cantidad y además saben cuál es el mejor sitio, encontrado por el conjunto del enjambre. Las abejas siguen volando, en busca de nuevas mejores soluciones, modificando su dirección en función de esos dos datos.
  La metaheurística comienza generando de forma aleatoria la velocidad y posición inicial de cada partícula que forma la población. Luego se evalúa el fitness de cada individuo, sabiendo con ello cuál es la mejor solución del enjambre.
  En cada iteración se actualiza la velocidad de las partículas teniendo en cuenta la mejor solución propia y la del conjunto, y con esa velocidad movemos la partícula, hallando así la nueva posición.
  Para terminar la iteración evaluamos las posiciones y actualizamos los valores de los óptimos encontrados por cada partícula y por el enjambre.

- ACO, Optimización basada en Colonia de Hormigas.
  El algoritmo ACO está basado en el comportamiento natural de las hormigas a la hora de buscar el camino más corto entre sus fuentes de alimento y el hormiguero. Cada hormiga inicia la búsqueda de forma aleatoria (ya que al principio no hay feromona) y al caminar va depositando un rastro de feromona (sustancia química) que con el tiempo se va evaporando, de modo que la hormiga que va por el camino más corto, es la que menos tiempo da a la feromona para evaporarse, por lo que su camino tendrá más rastro que los demás, y como cada hormiga elige el siguiente camino a tomar en función de la cantidad de feromona, cada vez se van acercando más al camino corto.
  La metaheurística se basa en ese proceso, de modo que el primer paso es generar la población inicial de forma aleatoria, luego se actualiza la matriz de feromona (formada por todos los arcos que forman los posibles caminos) y a continuación empiezan las iteraciones donde se actualiza la solución de cada individuo de la población teniendo en cuenta un método heurístico de construcción y el rastro de feromona.
  Para terminar la iteración se calcula el fitness y se actualiza la feromona (se realiza tanto la evaporación que evita caer en óptimos locales, como el aumento de la feromona en los lugares que forman parte de las mejores soluciones de esta iteración).
  También se pueden añadir otras operaciones al método que dependen de todo el conjunto de individuos, como por ejemplo, la mejora de soluciones mediante búsqueda local.
  Según la definición del apartado 1.2.5, lo más relevante es que Ξ almacena la mejor solución encontrada, la matriz de feromonas e información heurística y Φ determina la construcción de las nuevas soluciones a partir de la información que hay en Ξ.

En el siguiente capítulo se explicará en detalle la Optimización por Enjambre de Partículas, puesto que es la que la metaheurística en la que basamos este proyecto.

3. OPTIMIZACIÓN POR ENJAMBRE DE PARTÍCULAS

El concepto del método de Optimización por Enjambre de Partículas, denominado también PSO debido a sus siglas en inglés "Particle Swarm optimization", es que las partículas sobrevuelan el hiperespacio de soluciones potenciales de un problema, acelerando hacia mejores soluciones posibles. Se trata de una analogía con la naturaleza, concretamente está basado en el comportamiento de los enjambres de abejas en busca de alimento que veremos más adelante.

El PSO utiliza un conjunto de partículas (denominado población o enjambre), que representan soluciones candidatas. Cada una de esas partículas posee una posición concreta (x) y se va moviendo según una determinada velocidad (v) por todo el espacio de búsqueda, obteniendo así una nueva posición. Para calcular la velocidad de cada partícula se tiene en cuenta la mejor posición encontrada hasta el momento por la propia partícula (la denominamos p y cuyo valor en la función objetivo es Pbest) y la mejor posición encontrada hasta ese instante por todo el enjambre (llamada g y cuyo valor es Gbest).

Además el PSO forma parte de los métodos metaheurísticos, de resolución de problemas de optimización, por lo que las características de la metaheurística explicadas en el capítulo anterior, también son aplicables a este método en particular. Una de las más importantes es el hecho de que este método proporciona una solución cercana al óptimo global del problema, pero no asegura que coincida con el óptimo.

El PSO es un algoritmo simple y robusto, pero bastante efectivo, que optimiza un amplio rango de funciones. Tomando como referencia la naturaleza se podría decir que es un término medio entre los procesos evolutivos (los cuales requieren periodos de tiempo indefinidos, de larga duración) y los procesos neuronales (que ocurren en milisegundos).

La raíz del método está basada en dos componentes metodológicos (Eberhart y Kennedy, 1995):

- Está vinculado a la vida artificial en general y a un rebaño de pájaros, un banco de peces o un enjambre en particular.

- También está relacionado con la computación evolutiva, teniendo vínculos con los Algoritmos Genéticos y la Programación Evolutiva.
  El concepto de ajuste hacia el Pbest y Gbest que tiene el PSO es muy similar, conceptualmente hablando, al operador de cruce que utilizan los AG y además el PSO utiliza el concepto Fitness, al igual que se hace en la EP.

Finalmente, esta metaheurística, al igual que la programación evolutiva, se ve afectada por los factores estocásticos.

La optimización por enjambre de partículas es capaz de encontrar solución a muchos de los tipos de problemas que también son resueltos mediante los algoritmos genéticos, sin padecer, además, algunos de los problemas a los que éstos se enfrentan:

- Por un lado la interacción en grupo mejora el progreso hacia la solución.

- Por otro lado el PSO tiene memoria, la cual no poseen los AG, haciendo que todas las partículas retengan el conocimiento de las buenas soluciones, lo que permite mejorar el resultado.

La forma general del PSO lo que busca es optimizar funciones continuas no lineares, por lo que, como veremos más adelante, en el caso de este proyecto tendremos que utilizar una versión discreta del PSO para aplicarla al TSP puesto que éste es un problema discreto (posee un número concreto de vértices).

3.1 Origen

La metaheurística de Optimización por Enjambre de Partículas fue simulada por primera vez por James Kennedy (psicólogo) y Russell Eberhart (ingeniero eléctrico) en 1995. Ésta simulación fue una analogía de una bandada de pájaros en busca de maíz y tuvo mucha influencia del trabajo de Heppener y Grenander en el año 1990.

Reynolds y Heppner y Grenander son unos de los científicos más relevantes en el estudio del comportamiento social (Kennedy y Eberhart, 1995), en su caso simulando una bandada de pájaros. Reynolds se focalizó en la estética de las bandadas mientras que Heppener (zoólogo) lo hizo en las reglas subyacentes que permite a una gran cantidad de pájaros moverse de forma sincronizada (cambiando de dirección repentinamente, dispersándose y reagrupándose…). Ambos modelos están basados en gran medida en la manipulación de distancias inter-individuales, es decir, la sincronización del comportamiento de la bandada está basado en el esfuerzo de los pájaros por mantener una distancia óptima entre ellos mismos y sus vecinos.

Según (Kennedy y Eberhart, 1995), el término “Partícula” fue un compromiso puesto que los miembros de la población no tienen apenas masa ni volumen, por lo que podrían llamarse puntos, sin embargo la velocidad y aceleración es más aplicable a partículas. Es la segunda de las razones la que los hizo decantarse por la elección de “Partícula” frente a “Punto” y a consecuencia de dicha decisión, los

autores decidieron llamar a su representación Optimización por Enjambre de Partículas.

3.2 Evolución del PSO

A continuación describiremos brevemente el desarrollo conceptual del PSO (Kennedy y Eberhart, 1995).

El algoritmo comenzó siendo una simulación de un ambiente social simplificado, donde los agentes eran pájaros (tomando como base los estudios de Reynolds y Heppner) y la intención original fue simular gráficamente la impredecible coreografía sincronizada de una bandada de aves, pero como veremos acabó siendo la simulación de un enjambre.

A continuación se explican los precursores del PSO, en forma de modificaciones sobre ese PSO original, que han hecho que el algoritmo sea como lo conocemos hoy en día:

- Velocidad del vecino más cercano y la variable "locura".
  La población de pájaros es inicializada aleatoriamente con una posición para cada agente (pájaro) y con una velocidad. Ambos parámetros con sus correspondientes coordenadas es "x" e "y".
  En cada iteración un bucle del programa determinaba para cada agente, qué otro pájaro es su vecino más próximo y posteriormente asigna al primer agente la velocidad de dicho vecino. Ésta simple regla crea la sincronización del movimiento.
  Desafortunadamente tras pocas iteraciones la bandada acaba teniendo una dirección unánime e inmutable.
  Ese problema se soluciona añadiendo una variable estocástica llamada "locura". Lo que hace esa variable es añadir, en cada iteración, un cambio en algunas velocidades elegidas al azar, introduciendo así la variación suficiente para que la simulación resulte interesante y se asemeje más a la realidad.

- El vector Maizal (Cornfield vector).
  La simulación de Heppner introdujo una fuerza dinámica en la simulación. Sus pájaros se reunían en torno a un "roost", que es una posición determinada que los atrae hacia sí, hasta que finalmente aterrizan allí.
  Añadiendo esta fuerza, ya no era necesaria la variable "locura", ya que la simulación asumía vida en sí misma.
  Además en su modelo había otra cosa muy curiosa, los pájaros de Heppnes sabían dónde estaba su "roost", pero en la vida real los pájaros aterrizan en cualquier lugar que satisface sus necesidades más inmediatas (árbol, antena...), sobretodo, donde hay comida (si sacas comida de pájaros a la

calle, al momento hay muchos pájaros, aunque nunca hayan tenido un conocimiento previo de su localización).
Esto nos hace pensar que algo de la dinámica de las bandadas permite a sus miembros sacar provecho del conocimiento de otros.
Como consecuencia, nuestros autores crean el "Vector Maizal", que es un vector bidimensional con coordinadas "x" e "y" del plano, tal que:

- Cada agente evalúa su posición actual, donde x_x y x_y, son las coordenadas x e y de la posición actual del agente y la fórmula de evaluación es la siguiente:

  $$\text{Evaluación} = \sqrt{(x_x - 100)^2} + \sqrt{(x_y - 100)^2}\ ,$$

- Cada pájaro recuerda su mejor evaluación (valor Pbest) y la posición p que produjo ese valor en sus respectivas coordenadas: p_x y p_y.

- Mientras que cada agente se mueve en el espacio evaluando sus posiciones, sus velocidades se ajustan de forma sencilla aproximándose a la posición de su Pbest de la siguiente manera:

  Si el agente está a la derecha de su p (x_x>p_x), su v_x (velocidad en la coordenada x) se ajustará de forma negativa mediante una cantidad aleatoria ponderada con un parámetro del sistema de la siguiente forma: v_x= v_x – rand()*p_increment.
  Si está a la izquierda de su p (x_x<p_x), v_x se ajustará positivamente: v_x= v_x + rand()*p_increment.
  Si está por encima de p (x_y>p_y), la velocidad en la coordenada y será: v_y= v_y - rand()*p_increment.
  Y si se encuentra por debajo de su p (x_y<p_y): v_y = v_y + rand()*p_increment.

- Además cada agente conoce la mejor posición global del conjunto (g) y su valor (Gbest). Para almacenarlo se asigna a la variable g el índice del vector del agente con el mejor valor, por lo que $p_x(g)$ es la mejor posición "x" del grupo y $p_y(g)$ es la mejor posición "y" del mismo. Toda esta información está disponible para cada uno de los agentes del conjunto.

- De nuevo se ajustan las velocidades de cada agente, del mismo modo que antes pero teniendo en cuenta ahora la posición g (g_increment es un parámetro del sistema):

  Si x_x>$p_x(g)$: v_x = v_x– rand()*g_increment.
  Si x_x<$p_x(g)$: v_x = v_x+ rand()*g_increment.
  Si x_y>$p_y(g)$: v_y = v_y– rand()*g_increment.
  Si x_y<$p_y(g)$: v_y = v_y+ rand()*g_increment.

En la simulación un círculo marca la posición (100,100) que equivale a la evaluación = 0, y los agentes se representan con puntos de colores en su posición. De este modo un observador puede ver de forma gráfica e intuitiva

la evolución de la bandada de agentes hasta que encuentran el simulado maizal.
Cuando los parámetros p_increment y g_increment son relativamente altos, la bandada parece ser violentamente absorbida por el maizal. En pocas iteraciones todos los agentes parecen ser reagrupados dentro de un pequeño círculo que rodea al objetivo.
Cuando p_increment y g_increment son bajas, la bandada se arremolina alrededor del objetivo, acercándose de un modo realista, balanceándose rítmicamente con subgrupos sincronizados, para finalmente aterrizar en objetivo.

- Eliminación de variables auxiliares
Puesto que el modelo puede optimizar de forma simple, es importante identificar las partes del mismo que no son necesarias.
Kennedy y Eberhart pronto se dieron cuenta de que el algoritmo funcionaba igual de bien y parecía igual de realista sin la variable "locura", por lo que se eliminó.
Después se dieron cuenta de que la optimización ocurría un poco más rápido cuando la velocidad del vecino más cercano era eliminada, aunque el efecto visual cambiaba.
El cambio visual producido fue que movimiento ya no era tan sincronizado, por lo que la simulación pasaba de tener un comportamiento de bandada a comportarme más como un enjambre, pero con este nuevo comportamiento también era capaz de encontrar el maizal. En este punto fue en el que se dieron cuenta de que el PSO era más una simulación de enjambre que de bandada.
Las variables p y g y sus incrementos sí que son necesarias.

  - La primera es la memoria autobiográfica, cómo cada individuo recuerda su propia experiencia.
    El ajuste de la velocidad mediante p_increment es llamado "simple nostalgia", que muestra cómo cada individuo tiende a volver al lugar que más le ha satisfecho.

  - Por otro lado la variable g hace referencia al conocimiento público, aquel que es conocido por todo el enjambre.

Un valor alto de p_increment respecto a g_increment tiene como resultado el aislamiento individual en el espacio, mientras que lo contrario (alto valor de g_increment respecto al p_incerment), hace que la bandada vaya rápidamente a un mínimo local. Por tanto hay que buscar el equilibrio entre las dos parámetros del sistema.
Experimentalmente se ha comprobado que valores bastante similares de p_increment y g_increment son la búsqueda más efectiva del problema.

- Búsqueda multidimensional.
  A pasar de que el algoritmo parece modelar bastante bien una bandada buscando un maizal, los problemas de optimización más interesantes no son ni lineares, ni bidimensionales.
  Uno de los objetivos de los autores es modelar el comportamiento social, el cual es multidimensional.
  Para ello se realiza el "simple salto" de cambiar tanto x_x y x_y, como también v_x y v_y, de vectores unidimensionales a matrices D*N, donde D es el número de dimensiones y N el número de agentes.

- Aceleración por distancia.
  Aunque el algoritmo funcionaba bien, había algo estéticamente desagradable.
  Hasta ahora los ajustes de velocidad estaban basados en una prueba de desigualdad (si x_x>p_x disminuirlo, y si x_x<p_x aumentarlo).
  Ciertos experimentos revelaron que cuánto más lejos se revise el algoritmo, más fácil es de entender y mejorar su rendimiento, por lo que a partir de este momento, en lugar de comprobar el signo de la desigualdad, las velocidades fueron ajustadas de acuerdo con sus diferencias dimensionales, con respecto a sus mejores localizaciones, de la siguiente manera:

  - $v = v$ + rand()*p_increment*$(p - x)$
  - $v = v$ + rand()*g_increment*$(p(g) - x)$

- Versión actual simplificada.
  Pronto se dieron cuenta de que no había una buena forma de adivinar el correcto valor para p_increment y g_ingrement, por ello esos dos términos fueron también eliminados del algoritmo.
  El factor estocástico fue multiplicado por 2 para dar una media de 1, por lo que ese agente sobrevolaría el objetivo la mitad de tiempo.
  Esta versión supera la anterior. Investigaciones posteriores mostrarán si existe un valor óptimo para la constante actualmente fijada en 2, si el valor debe ser desarrollado para cada problema, o si puede ser determinado a partir de algún conocimiento de un problema particular.
  El PSO simplificado actual ajusta la velocidad del siguiente modo:

  - $v = v$ + 2*rand()*$(p - x)$ + 2*rand()*$(p(g) - x)$

- Otros experimentos.
  Otras variaciones del algoritmo han sido probadas, pero ninguna parece mejorar la versión simplificada anteriormente explicada.

3.3 Analogía con la naturaleza

El PSO es una técnica de optimización iterativa, siendo estas técnicas tan antiguas como la vida en sí misma, muestra de ello es que incluso los seres más primitivos actuaban de acuerdo a un impulso muy simple, el de mejorar su situación.

Observando la naturaleza encontramos muchas acciones y estrategias naturales, que demuestran su efectividad con la persistencia de las especies que participan en ellas, por lo que no es sorprendente que bien de forma explícita o implícita, varios modelos matemáticos de optimización tengan su base en comportamientos biológicos.

Además de la efectividad de algunos comportamientos de las especies, otra razón para utilizar métodos basados en la naturaleza es, que no parece ilógico suponer que algunas mismas normas sociales subyacen en el comportamiento social de los animales y de las personas (Kennedy y Eberhart, 1995).

El socio-biólogo E. O. Wilson, referenciándose a los bancos de peces, dijo: "Al menos en teoría, los miembros individuales de un banco de peces pueden beneficiarse de los descubrimientos y de la experiencia previa de todos los demás miembros del banco durante la búsqueda de alimento. Esta ventaja puede llegar a ser decisiva, pesando más que la desventaja de la competencia por los productos alimenticios, siempre que los recursos estén distribuidos impredeciblemente en bancales".

De la declaración de Wilson podemos interpretar, de manera general, que el intercambio social de información entre especies ofrece una ventaja evolutiva. Esta conclusión es una hipótesis fundamental para el desarrollo del PSO.

Entre los modelos que se basan en conductas de la naturaleza, podemos distinguir aquellos que se basan en comportamientos individuales de los que lo hacen en comportamientos colectivos (Clerc, 2010):

- En el caso individual, la estrategia más utilizada es tratar de beneficiarse en todo momento de cualquier mejora inmediata (que mejore el objetivo), este modo de actual puede hacer que se caiga en un óptimo local.
  Para evitar estos óptimos locales lo que se hace es permitir que en algunos caso se empeore, con la esperanza de obtener al final un mejor resultado, pero para evitar el caos total, el sujeto debe almacenar la mejor posición para poder volver a ese punto en caso de ser necesario.

- En el caso colectivo, esa información puede ser conservada de forma natural, basta con que el individuo con la mejor posición no se mueva y sea el resto del grupo el que siga explorando los alrededores, necesitando dos

parámetros adicionales al caso anterior, que son el tamaño del grupo (puede ser fijo o variable durante la búsqueda) y su estructura (entendiendo por estructura al modo de transmisión de la información que se da entre los miembros del grupo). Es más probable caer en un óptimo local cuando se utilizan métodos individuales, que cuando se usan los colectivos.

El PSO es un método colectivo además de iterativo, cooperativo y parcialmente aleatorio.

A lo largo del tiempo muchos investigadores han creado simulaciones computacionales basadas en el movimiento de organismos animales (bandadas de pájaros, bancos de peces…). Cabe remarcar la labor tanto de Reynold como de Heppner y Grenander, cuyos descubrimientos fueron tenidos muy en cuenta por Kennedy y Eberhart a la hora de adentrarse en el mundo del PSO.

Por supuesto, el comportamiento humano no es idéntico al de un banco de peces, una bandada de pájaros o un enjambre de abejas, los cuales se mueven para evitar depredadores, buscar comida, buscar compañeros, optimizar su entorno…mientras que los humanos no solo ajustan su trayectoria por factores físicos, sino que también tienen en cuenta las variables cognitivas y experimentales, por lo que los métodos no son una copia exacta del comportamiento animal.

En el desarrollo del PSO lo que se simula es el modo de actuar de los enjambres, aunque, como vimos en el apartado anterior, comenzó simulando bandadas de pájaros.

En la naturaleza las abejas vuelan por el espacio buscando el lugar donde haya mayor concentración de flores, puesto que allí será probablemente el sitio donde encuentren más polen para alimentarse.

Durante su vuelo, cada abeja es capaz de recordar el sitio donde ella ha localizado mayor densidad floral y además todas las abejas del enjambre saben cuál ha sido el sitio con más flores localizado por el conjunto del enjambre.

Cada abeja va variando individualmente su movimiento aproximándose hacia las dos posiciones que recuerda, probablemente hacia algún punto entre ambas.

Si durante el vuelo alguna abeja encuentra un lugar con más flores que el encontrado hasta el momento por el enjambre, lo comunicaría y todo el enjambre reorientaría su búsqueda hacia ese nuevo punto (si el punto encontrado supera el mejor sitio que ella ha encontrado, pero no supera el mejor global encontrado por el enjambre, en ese caso, sólo su dirección se modificaría).

Para desarrollar el algoritmo hay que tener en cuenta los 5 principios básicos de la inteligencia de los enjambres que desarrolló Millonas en su modelo de vida artificial (Eberhart y Kennedy, 1995):

- Principio de proximidad: La población debe ser capaz de llevar a cabo cálculos simples del espacio y del tiempo.

- Principio de calidad: la población debe de ser capaz de responder a los factores de calidad del entorno.

- Principio de diversidad: La población no debe realizar sus actividades por canales excesivamente estrechos.

- Principio de estabilidad: La población no debe cambiar su forma de comportarse cada vez que cambie el entorno.

- Principio de adaptabilidad: La población tiene que cambiar su comportamiento cuando merece la pena la ganancia computacional. (este principio y el anterior son lados opuestos).

El PSO cumple estos 5 principios. Se trata cálculos en un espacio n-dimensional llevados a cabo sobre una serie de pasos de tiempo (proximidad). La población responde a los factores de calidad Pbest y Gbest. La distribución de respuestas entre p y g asegura la diversidad. Y finalmente, la población sólo cambia su estado (comportamiento) cuando g cambia (estabilidad porque sólo cambia en ese caso y a su vez adaptabilidad porque cuando varía g es capaz de ajustarse).

3.4 Descripción del método

El PSO está compuesto por un número de partículas localizadas en el espacio de búsqueda de alguna función o problema.

Cada partícula realiza un seguimiento de sus coordenadas en el hiperespacio, las cuales se asocian con la mejor solución que ha logrado la propia partícula hasta el momento (p), cuyo fitness (valor de la función objetivo) se almacena en Pbest, y con la mejor solución que ha logrado el conjunto del enjambre (g), cuyo valor se almacena en Gbest.

El fundamento del algoritmo consiste ir cambiando, a cada paso del tiempo, la posición de cada partícula según la velocidad de la misma. La velocidad se ve afectada por la posición actual, su posición p y por g. El usuario debe determinar la velocidad máxima a la que se pueden mover las partículas para evitar que sea tan

alta como para hacer que las partículas vuelen fuera del espacio útil y se produzcan desbordamientos.

La inicialización del PSO es, al igual que para los algoritmos genéticos, un conjunto de soluciones aleatorias (posiciones), pero el PSO añade además, que a esas soluciones potenciales se les asigna una velocidad inicializada también se forma estocástica, de modo que después las soluciones potenciales (llamadas partículas) "vuelan" por el espacio.

Cada una de las partículas evalúa el objetivo de la función en su posición actual y posteriormente determinan su movimiento en el espacio de búsqueda combinando la información de su posición actual, su mejor posición encontrada y la mejor posición del conjunto de partículas del enjambre; todo ello afectado por algunas perturbaciones aleatorias.

La siguiente iteración se lleva a cabo cuando todas las partículas que forman el enjambre se hayan movido.

Finalmente el enjambre, como un todo, probablemente se sitúe muy próximo a un óptimo de la función a estudiar.

Mientras el sistema realiza iteraciones, las partículas individuales son conducidas hacia un óptimo global basado en la interacción de sus búsquedas particulares y las búsquedas públicas del conjunto. El criterio de parada (ya sea en forma de umbral de error, de iteraciones máximas o al presionar una tecla) debe ser definido y cuando sea satisfecho, las iteraciones deben cesar y el mejor de los vectores encontrados se deberá escribir en un archivo como solución al problema.

Parece obvio que cuantas más iteraciones se realicen (lo cual implica más posiciones analizadas), mejor será la solución, pero hay que tener en cuenta que los recursos necesarios para cada iteración tienen un precio y hay que evaluar si merece la pena la mejora que puedas hallar en relación al coste que supone. Por ello se marca un criterio de parada.

3.4.1 Pasos del algoritmo

En primer lugar introducimos la nomenclatura que vamos a utilizar posteriormente para describir el algoritmo paso a paso. Cada individuo (partícula) del PSO se compone de tres vectores dimensionales:

- x_i que es la posición actual de la partícula i. Conjunto de coordenadas (tantas coordenadas como dimensiones tiene el espacio) que describe el punto actual en espacio de búsqueda.

- p_i que es el conjunto de coordenadas de la mejor posición encontrada hasta el momento por la partícula i.
  Además cada partícula también posee la variable $Pbest_i$ ("previous best") que almacena el fitness de la posición p_i, con el objetivo de hacer comparaciones de manera más sencilla.

- v_i que se refiere a la velocidad de la partícula i.

Puesto que el PSO no se trata de una colección de partículas individuales, sino que interactúan entre ellas para poder así progresar y lograr una buena solución al problema, también se tiene en cuenta el siguiente vector común para todo el enjambre:

- g que es el conjunto de coordenadas de la mejor posición encontrada hasta el momento por cualquier partícula del conjunto.
  Análogo a $Pbest_i$, existe la variable $Gbest$ que almacena el valor de g.

En resumen, podemos describir el algoritmo PSO como una técnica de optimización para usar en un espacio de números reales, cuyas soluciones potenciales del problema son representadas como partículas que poseen coordinadas x_i y velocidades de cambio v_i en un espacio D-dimensional. Cada partícula i tiene el registro de su mejor rendimiento previo en el vector p_i y a la variable g se le asigna la posición del mejor rendimiento encontrado por todo el conjunto hasta el momento. En cada iteración del algoritmo se realiza: el ajuste de v_i hacia la dirección de p_i y de g y la actualización y evaluación de cada partícula y ajuste de los parámetros p_i y de g.

A continuación se expondrán detalladamente paso a paso ese proceso [10]:

En primer lugar hay que inicializar la población por lo que para cada partícula i se hace lo siguiente:

- Se inicializa aleatoriamente x_i con la condición de que se encuentre dentro del límite inferior y superior del espacio de búsqueda.

- Se asigna a p_i la misma posición que se acaba de hallar aleatoriamente para x_i, puesto que al ser la iteración inicial, el mejor valor encontrado por la partícula corresponde con la única posición que ha explorado, la actual.

- Se calcula el fitness de x_i para la partícula y se otorga el valor calculado a su correspondiente $Pbest_i$.

- Se inicializa g con la posición x_1 de la primera partícula analizada y el fitness de x_1 se otorga a $Gbest$.
  Durante el proceso de inicialización, para el resto de las partículas lo que se hace es mirar si su $Pbest_i$ es mejor que el $Gbest$, en cuyo caso se reemplaza g por la posición x_i y el valor de x_i se le da a la variable $Gbest$.

- Se inicializa de manera aleatoria la velocidad v_i.

Comienzan las iteraciones del algoritmo en las que se llevan a cabo los siguientes pasos para cada partícula i (mientras no se cumpla el criterio de parada):

- Se actualiza la velocidad de la partícula según la siguiente fórmula:
  $$v_i = \alpha * v_i + \beta * \text{rand}() * (p_i - x_i) + \gamma * \text{rand}() * (g - x_i)$$
  Donde rand() son números aleatorios, y α, β y γ son parámetros muy importantes que se deben definir bien ya que regulan el comportamiento y la eficacia del modelo. Son los responsables de que haya un equilibrio entre la exploración y explotación del algoritmo.

- Se actualiza la posición actual de la partícula, añadiendo a x_i las coordinadas de v_i y obteniendo así la nueva x_i que utilizaremos para esta iteración. La fórmula es la siguiente:
  $$x_i = x_i + v_i.$$

- La posición actual x_i se evalúa mediante la función objetivo como posible solución al problema.

- Si la evaluación de la posición es mejor que ninguna de las que han sido encontradas hasta el momento por la partícula (Fitness de x_i mejor que $Pbest_i$):

  - Se actualiza p_i, sustituyendo sus coordenadas por las de x_i.

  - Se actualiza el valor de $Pbest_i$ otorgándole el valor del fitness de x_i.

  En caso contrario el p_i y $Pbest_i$ de la partícula no se modifican.

- Sólo en caso de que la partícula haya mejorado su p_i, se examina si el $Pbest_i$ de esa partícula es mejor que $Gbest$, y en caso de ser cierto:

  - Se actualiza g, sustituyendo sus coordenadas por las de p_i.

  - Se actualiza el valor de $Gbest$ otorgándole el valor de $Pbest_i$.

En caso contrario g y $Gbest$ conservan sus valores.

- Se evalúa la condición de parada (establecida por el usuario) y:
  - Si no se ha alcanzado la condición de parada, se vuelve al primer paso del proceso de las iteraciones.
  - Si la condición de parada se cumple, se devuelve g como la mejor posición encontrada y $Gbest$ como el mejor valor hallado. Y se finaliza el algoritmo.

3.5 Versiones del PSO

A pesar de que el método es simple y bastante conciso, existen varias versiones en función del espacio de búsqueda que se debe de analizar (continuo, discreto…), de cómo se establecen los parámetros…

A continuación mostramos dos versiones concretas del PSO.

La primera de ellas cambia el concepto de "intercambio de información" que teníamos en la versión original del PSO, ya que en lugar de compartir la información con todo el enjambre, cada partícula sólo comparte información con una parte concreta del mismo.

La segunda versión es una adaptación del PSO para poder utilizar este método en problemas con espacios discretos, como es el caso del TSP.

3.5.1 Modelo Lbest

Se trata de una versión local del PSO (Eberhart y Kennedy, 1995). En esta variante las partículas sólo tienen información de ellas mismas y de sus vecinas más cercanas, en lugar de tener relación con la totalidad del grupo.

La diferencia es que las partículas en vez de moverse hacia las posiciones p_i y g, con mejor valor propio ($Pbest_i$) y mejor valor global ($Gbest$) respectivamente, lo que hacen es dirigir su movimiento en función de las posiciones p_i y l_i, es decir, la de mejor valor propio y la de mejor valor local ($Lbest_i$).

Por tanto el modelo apenas varía. Lo que hace es, que en lugar de utilizar las variables g y $Gbest$ ya conocidas, utiliza l_i, que es el conjunto de coordenadas de la

mejor posición encontrada hasta el momento por el vecindario de la partícula i, y $Lbest_i$, que es la variable que almacena el fitness de la posición l_i.

Haciendo la sustitución de esas dos variables en el PSO general explicado en el apartado anterior, obtenemos el algoritmo para el modelo Lbest.

Por un lado esta versión tiene la ventaja de que es menos propensa a encontrar un óptimo local, frente a la versión estándar del PSO. Esto se debe a que un número de grupos de partículas espontáneamente se separa y explora diferentes regiones, por lo que resulta más flexible que el modelo original. De todos modos esta versión también tiene posibilidad de caer en un óptimo local (la mejor manera de evitarlo es establecer una velocidad máxima pequeña, aunque sabiendo que es un algoritmo aproximado, tampoco lo asegura), pero aun existiendo la posibilidad, ésta es menor que en el modelo original.

Por otro lado, aunque está versión rara vez se queda atrapado en óptimos locales, está probado experimentalmente que necesita más iteraciones, de media, que el PSO estándar para cumplir el mismo criterio de nivel de error. Por ello se puede decir que la versión local parece funcionar peor que el algoritmo general.

3.5.2 Modelo Binario Discreto, DPSO

El PSO ajusta las trayectorias de sus partículas a lo largo del espacio del problema en función de la mejor solución previa de cada partícula y la mejor del conjunto.

La versión estándar opera en un espacio continuo (de números reales), donde las trayectorias se definen como cambios en la posición (en un número de dimensiones que están representadas por las D coordenadas del vector x_i).

El caso, es que muchos problemas de optimización poseen un espacio discreto de búsqueda (no utilizan variables continuas, sino que cuentan con niveles de variables). Los típicos ejemplos son aquellos que requieren el orden u organización de elementos discretos, como en el problema que nos concierne a nosotros, el ATSP.

Es la razón anterior, junto con el hecho de cualquier problema, discreto o continuo, puede ser expresado en notación binaria, las que hacen que un optimizador que opere con funciones de dos valores (binarias) parezca ventajoso. Esto propulsó el desarrollo de esta versión: el algoritmo de Optimización por Enjambre de Partículas Discreto, DPSO.

En esta versión (Kennedy y Eberhart, 1997) el algoritmo opera con variables binarias discretas (solo pueden tomar el valor 1 o el valor 0).

Y dado que el PSO general trabaja ajustando trayectorias a través de la manipulación de cada coordinada (dimensión del espacio) de la partícula y algunos de los éxitos del algoritmo son derivados de que sobrevuela óptimos locales conocidos, explorando más allá de ellos y entre ellos, conceptos claves a desarrollar en este modelo son, cómo tratar en un espacio discreto la trayectoria, la velocidad, la exploración "entre y más allá"…

En el espacio binario una partícula puede moverse de la esquina más cercana a la más lejana del espacio cambiando diversos números de bits. La velocidad de la partícula en conjunto puede ser descrita por el número de bits cambiados por iteración, es decir, la velocidad de una partícula podría ser una secuencia de operadores de intercambio o permutaciones. Una partícula con 0 bits cambiados no se mueve, mientras que el movimiento más lejano se da cuando se cambian todas las coordinadas binarias (bits).

Hay que definir la velocidad como cambios de las probabilidades de que un bit (coordenada de los vectores) de x_i de tome un estado o el otro.

Cada coordenada de v_i representa la probabilidad de que su bit correspondiente en x_i tome el valor 1. (Sí para cierta coordenada v_i=0.04, hay un 40% de posibilidades de que su bit en x_i sea 1, y un 60% de que sea 0).

La fórmula de la velocidad (aplicable a cada coordenada de cada partícula) se mantiene sin cambios, excepto por el hecho de que ahora p_i, x_i y g son enteros en {1, 0} y, teniendo en cuenta que v_i es una probabilidad, su valor debe estar en el intervalo [0'0, 1'0]. Su aspecto es el siguiente:

$$v_i = \alpha * v_i + \beta * \text{rand}() * (p_i - x_i) + \gamma * \text{rand}() * (g - x_i)$$

En cuanto a la actualización de x_i, una transformación logística S(v_i) puede ser usada para lograr esta última modificación del modelo. S() es una transformación sigmoidea que se aplica del siguiente modo:

$$S(X) = \frac{1}{1 + \exp(-X)}$$

El cambio resultante en la posición es definido por la siguiente regla, donde rand() es un número aleatorio seleccionado de una distribución uniforme [0.0, 1.0] :

Si(rand()<S(v_i)); entonces x_i = 1; sino x_i = 0

Al igual que en el caso continuo se limitaba v_i con un v_{max} (parámetro del sistema), en el caso discreto también se hace de modo que $|v_i| < v_{max}$, pero esto simplemente limita la probabilidad última de que el bit x_i tome como valor un 0 o

un 1, es decir, si v_{max} = 6.0, entonces las probabilidades de S(v_i) se limitarán entre 0,9975 y 0,0025 (probabilidades resultantes cuando se utilizan las velocidades extremas 6 y -6).

Especificar un valor alto de v_{max} hace a los nuevos vectores menos prometedores. Parte de la función de v_{max} en el DPSO es fijar un límite a la exploración adicional una vez que la población haya convergido.

Cuando v_{max} era alto en el problema continuo significaba que aumentaba el rango explorado por la partícula, en la versión binaria ocurre lo contrario, bajos v_{max} permiten mayores tasas de mutación.

Originalmente las trayectorias se ajustaban alterando la velocidad (o tasa de cambio de posición) según:

$$x_i = x_i + v_i.$$

En la presente versión discreta binaria, dado que v_i funciona como un umbral de probabilidad, los cambios en v_i podrían representar un cambio de posición en sí mismo (no tiene en cuenta la posición anterior a la hora de actualizar x_i, sino que solo tiene en cuenta la velocidad):

Si(rand()<S(v_i)); entonces x_i = 1; sino x_i=0.

A medida que S(v_i) se aproxima a cero, entonces la posición de la partícula se fija más probablemente en el valor 0.

La trayectoria en el modelo binario es probabilística y la velocidad, en una sola dimensión, es la probabilidad de que un bit de x_i cambie; por lo que aunque v_i deba permanecer fijo (según la fórmula de la velocidad v_i permanece fijo cuando x_i = p_i = g), la "posición" de la partícula en esa dimensión es dinámica ya que cambia la polaridad (de 1 a 0 y viceversa) con la probabilidad dependiente del valor v_i.

La probabilidad de que un bit sea 1 es S(v_i) y la probabilidad de que sea 0 es (1 – S(v_i)). Por tanto las probabilidades de cambio son:

- Si el bit ya está en 0, la probabilidad de cambio es S(v_i)

- Si el bit tiene valor 1, la probabilidad de cambio es (1 – S(v_i)).

Esto hace que la probabilidad de que un bit cambie, p(Δ), (siendo ésta un ratio de cambio absoluto, no direccional para un bit) dado el valor v_i de su coordenada es:

$$p(\Delta)= S(v_i)*(1 - S(v_i)) = S(v_i)- S(v_i)^2$$

En conclusión podríamos decir que el mayor cambio de esta versión es que los miembros de la población no son vistos como soluciones potenciales, sino que son probabilidades.

El valor v_i para cada dimensión determina la probabilidad de que el bit x_i de esa dimensión tome un valor u otro, pero x_i por sí mismo no tiene valor hasta que no es evaluado (la actual posición de la partícula en el espacio d-dimensional es efímera). Una partícula individual i con su particular v_i podría tener una posición diferente x_i en cada iteración.

Esto no ocurre en la versión general del PSO donde los individuos son una representación de una solución potencial al problema. Cada individuo es evaluado, actualizado y vuelto a analizar.

En el capítulo siguiente se explicará de forma detallada la aplicación del algoritmo DPSO al problema específico ATSP y se analizarán sus resultados experimentales.

4. IMPLEMENTACIÓN Y EXPERIMENTACIÓN

Este capítulo comienza con la explicación, de forma teórica, de la manera elegida para implementar el método PSO al problema ATSP, es decir, la forma de resolver una instancia dada del ATSP, mediante la aplicación de un algoritmo basado en la Optimización por Enjambre de Partículas.

De forma paralela al desarrollo de este documento, hemos programado un algoritmo en lenguaje C++, que lo que hace es resolver una instancia del ATSP mediante la realización de unos pasos concretos que están fundamentados en el algoritmo PSO, o dicho de otro modo, hemos puesto en práctica la implementación anteriormente mencionada, obteniendo así un algoritmo capaz de encontrar una solución aceptable a cualquier problema ATSP (o STSP ya que es un caso particular del ATSP) al que se aplique.

Para el desarrollo del algoritmo hemos utilizando el programa CodeBlocks, que es un entorno de desarrollo integrado (posee editor del código fuente, depurador y constructor entre otras cosas) y libre para el desarrollo de programas en lenguaje C y C++.

Los pasos que se han llevado a cabo para construir el programa o algoritmo serán explicados en detalle en el segundo apartado de este capítulo.

En el momento en que acabamos de construir el programa, éste fue utilizado para resolver 19 problemas ATSP de la librería TSPLIB. Los resultados de las ejecuciones del programa, así como el enunciado y las variables utilizadas en cada caso se muestran en el apartado “resultados experimentales”.

Para concluir el capítulo, se muestra y comenta el análisis estadístico llevado a cabo sobre los resultados experimentales obtenidos por el programa.

4.1 Aplicación de la metaheurística PSO al problema TSP

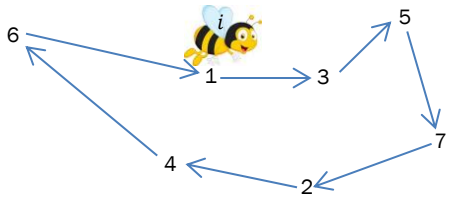
En este documento lo que pretendemos es utilizar el PSO para resolver el problema del TSP, por lo que el método debe de buscar soluciones en un espacio de búsqueda discreto.

Para ello necesitamos adaptar el PSO original, que trabaja con espacios continuos, y lo haremos utilizando variables discretas. En los últimos años, varios científicos han hecho versiones para utilizar el PSO en problemas de optimización discretos, los primeros autores que propusieron el DPSO fueron Kennedy y Eberhart, cuyo

modelo es que se ha utilizado como base en este trabajo para explicar el apartado 3.5.2.

En el desarrollo de nuestro programa vamos a tener en cuenta, tanto el fundamento de la versión DPSO de Kennedy y Eberhart como el DPSO desarrollado por Huilian Fan (Fan, 2010).

Partiendo de la base de que en nuestro problema ATSP tenemos n ciudades, la posición actual de cada partícula que forma el enjambre está establecida por el vector x_i donde la secuencia $x_{i1}, x_{i2}, x_{i3}, \dots, x_{in}, x_{i(n+1)}$ indica la trayectoria seguida por la partícula i (la ciudad x_{i1} es la que visita en primer lugar, luego va a la ciudad x_{i2}...). Un ejemplo muy sencillo de cómo se representa x_i es el que se muestra en la figura 8, donde la ciudad de partida es la 1. El vector es de tamaño n+1 porque la trayectoria debe de ser un ciclo, es decir, tiene que acabar en la misma ciudad que empezó ($x_{i1} = x_{i(n+1)}$).



x_i = {1, 3, 5, 7, 2, 4, 6, 1}

Figura 8. Vector de la posición actual de la partícula i. Elaboración propia.

Los vectores p_i y el vector g se representan del mismo modo que x_i.

En nuestra versión redefiniremos la velocidad (entendiéndola como la responsable del movimiento o actualización de las partículas), inspirándonos en el operador de permutación ("Swap Operator"). Para ello, añadimos la operación de cruce, la de inyección o la inversa y el factor de adaptación al ruido. Estas operaciones van a ser las responsables de la actualización de las partículas en cada iteración.

A continuación vemos el diagrama de flujo del algoritmo y la explicación de los pasos que lo componen, incluido el funcionamiento de las operaciones ya mencionadas.

4.1.1 Pasos del algoritmo

En el figura 9 se muestra esquemáticamente el diagrama de flujo que vamos a seguir para implementar el PSO al ATSP. Posteriormente explicaremos cada uno de los pasos del algoritmo.

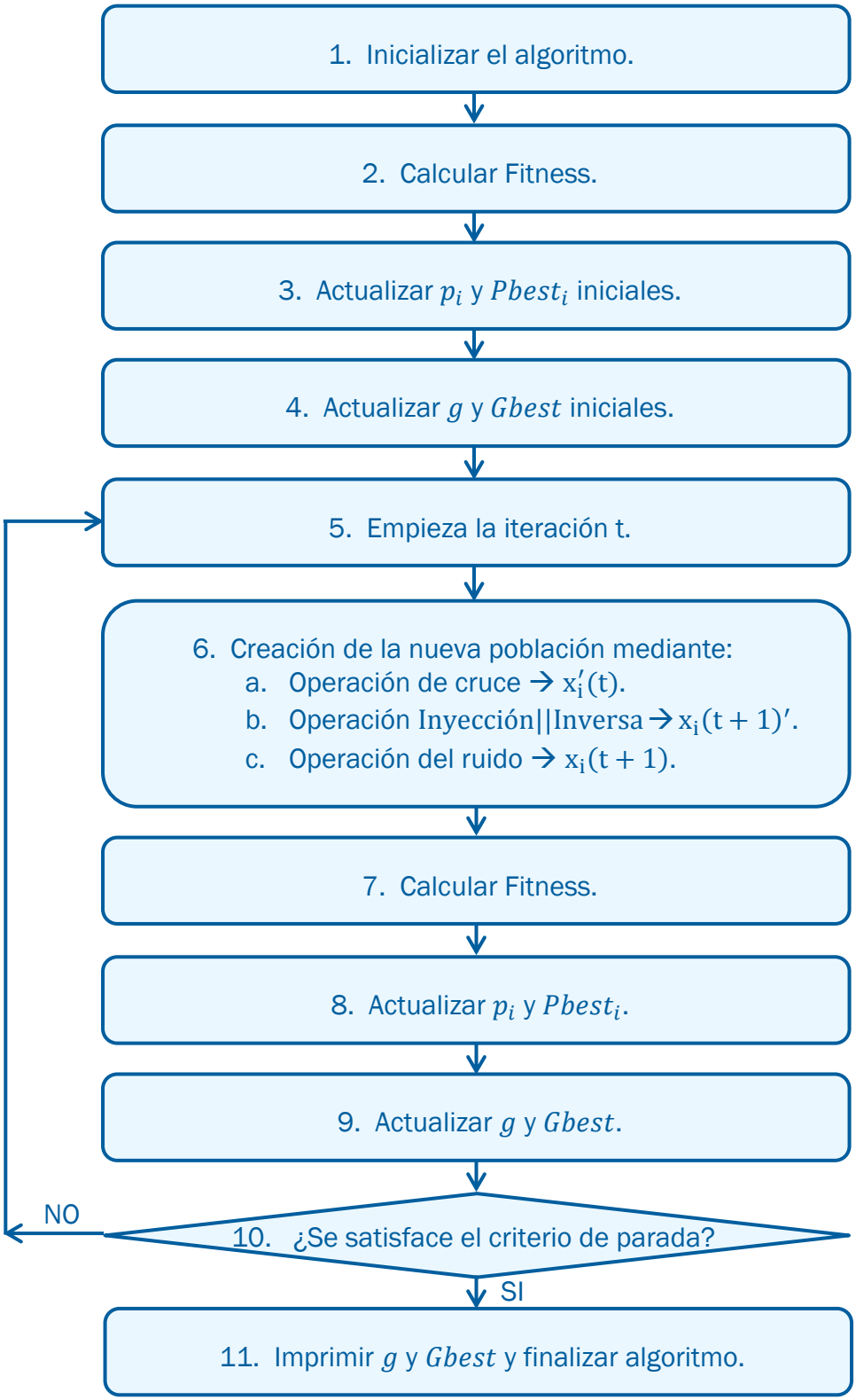


Figura 9. Pasos para la implantación del método PSO al ATSP. Elaboración propia.

En el primer paso hay que inicializar el algoritmo, asignando valores fijos a algunas variables, pidiéndolos por pantalla al usuario o incluso generándolos según algún procedimiento establecido. La diferencia entre establecer un valor o pedirlo es, que si lo asigna el programador ese valor va a ser constante para el programa (queda establecido internamente) independientemente de que sea una instancia u otra, mientras que si el valor se pide por pantalla, el usuario decide cada vez que ejecute del programa qué valor le quiere asignar. En el algoritmo debemos inicializar:

- La matriz de distancias utilizando los datos que nos ofrece el enunciado.

- Número de partículas que forma la población. Nuestro programa está diseñado para pedir el valor por pantalla, dado que no sabemos cuál es la elección óptima y así se puede ir variando para ver con cuál funciona mejor.

- Ciudad de partida de las partículas. También es un dato pedido al usuario, puesto que en muchos enunciados no viene establecida. Nótese que la ciudad de partida no afecta al resultado del problema porque es un ciclo Hamiltoniano, esta variable sólo afecta a la percepción visual del vector a la hora de mostrar el resultado.

- Longitud de cruce (m), es un valor constante para cada ejecución que determina el funcionamiento de los operadores que utilizaremos para actualizar la población. Se pide al usuario, porque, al igual que ocurre con el número de partículas, no conocemos su valor óptimo.

- Número de iteraciones que debe realizar el algoritmo antes de su finalización. El número de iteraciones máximas es el criterio de parada que nosotros hemos elegido y también se pide su valor al usuario.

- Hay que generar el recorrido inicial x_i para cada partícula, que comenzará y acabará por la ciudad de partida, siendo el resto de la secuencia de su trayectoria elegida al azar.

- Tras establecer x_i, asignamos valores a p_i, que en el caso de la inicialización son una copia exacta de la posición inicial. Además la distancia de dichas trayectorias se guardan en su correspondiente Pbest_i.

- Otorgamos a g la trayectoria p_i de la partícula i que haya obtenido mejor Pbest_i. Ese mismo valor Pbest_i se le asigna a la variable Gbest.

El segundo paso es análogo al paso 7, lo que hay que hacer en ambos es calcular la distancia total que implica la trayectoria x_i de cada partícula (a esa distancia también se le conoce como fitness y es el valor que se quiere minimizar en la

función objetivo). Para ello utilizamos la matriz de distancias y la secuencia de esa trayectoria y sumamos la distancia que hay de cada ciudad a la ciudad siguiente según esa secuencia, de modo que al final obtenemos la distancia total del ciclo recorrido por la partícula. Ese valor se almacena para utilizarlo en comparaciones venideras.

Los pasos número 3 y 8 lo que hacen es actualizar el vector p_i y $Pbest_i$ de cada partícula. Para ello lo que se hace es comparar el valor del fitness de $\mathrm{x_i}$ calculado en el paso anterior, con el $Pbest_i$. Si la distancia total de la trayectoria $\mathrm{x_i}$ (fitness) es menor que $Pbest_i$, se actualiza p_i asignándole la trayectoria de x_i ya que es mejor que la mejor encontrada hasta el momento y se otorga a $Pbest_i$ el valor del fitness de $\mathrm{x_i}$, en caso contrario no se modifican ni p_i, ni $Pbest_i$.

En los pasos 4 y 9 se actualizar la trayectoria del vector g y su valor Gbest. El modo de hacerlo es comparar el valor $Pbest_i$ de cada partícula con el valor Gbest. Si alguna partícula posee un $Pbest_i$ menor que Gbest, entonces se actualiza el vector g con la trayectoria p_i de dicha partícula y Gbest pasará a tener el valor de $Pbest_i$. En caso de que varias partículas posean mejor fitness que el valor de Gbest, se escogería aquella que tuviese mejor $Pbest_i$, es decir, el más pequeño. Y en el caso en que ninguna partícula mejore la trayectoria de g, se queda tanto el vector g, como la variable Gbest igual que estaban.

En el paso 5 comienza la iteración, por lo que si se trata de la primera (venimos del paso 4) t=1, y en el caso de que vengamos del paso 10, la t se va incrementando (cada vez que vuelve al paso 5) indicándonos en que iteración nos encontramos.

El paso 6 de creación de una nueva población (actualización las x_i), lo que se hace es obtener una nueva posición actual para las partículas mediante la aplicación de la operación de cruce, la inyecta o inversa y la operación de ruido. Estas operaciones serán explicadas detalladamente en los siguientes sub-apartados.

En el paso 10 lo que hacemos es evaluar el criterio de parada para saber si debemos de seguir iterando o si debemos finalizar el algoritmo. Para ello lo que se hace es comprobar si la iteración t en la que nos encontramos es igual o mayor que el número de iteraciones máximas que tenemos del paso 1.

- Si el criterio de parada no se satisface (no se ha llegado aún al número de iteraciones máximas establecido), se volverá al paso 5 para empezar una nueva iteración.

- Si t es igual o mayor que tmax, se satisface el criterio de parada por lo que pasaríamos al último paso, el 11.

Este último paso consiste en imprimir el vector g y el valor Gbest (es el resultado de nuestro problema, que nos indica la mejor trayectoria encontrada y la distancia que implica) y posteriormente se terminaría la ejecución del algoritmo.

A continuación explicaremos en qué consisten las operaciones de cruce, de inyección, de inversión y de adaptación al ruido, para entender mejor el modo en que se actualizan las partículas que forman la población.

4.1.1.1 Operación de cruce

Partiendo de la posición actual x_i de cada partícula, lo que hace la operación de cruce es intercambiar algunos de sus coordenadas obteniendo la posición x_i' mediante la siguiente fórmula:

$$x_i'(t) = x_i(t) \oplus \big((c_1 \times r_1) \quad p_i(t)\big) \oplus \big((c_2 \times r_2) \quad g(t)\big)$$

Donde t es el número de iteraciones del algoritmo, $x_i(t)$ es el ciclo que hace la partícula i en la iteración t, $p_i(t)$ es el mejor ciclo encontrado por la partícula i hasta la iteración t y $g(t)$ es la mejor trayectoria encontrada por todo el conjunto hasta el momento t.

Los valores $m_1 = (c_1 \times r_1)$ y $m_2 = (c_2 \times r_2)$ representan las longitudes de cruce, m_1 representa el número de ciudades (coordenadas del vector) que se cruzan entre x_i y $p_i(t)$, y Por otro lado m_2 indica la cantidad de ciudades que se Cruzan entre el vector resultante de hacer el cruce anterior y $g(t)$.

Los valores óptimos de m_1 y m_2 los desconocemos por lo que queda como tema pendiente a ser estudiado en un futuro. En este trabajo hemos asumido que m_1 = m_2 = m y que es constante para todo el algoritmo, siendo la m una variable que se pide por pantalla al usuario del programa. En la experimentación probaremos con varios (siempre que no sea mayor que n, ya que si es n se copia completamente un vector en otro).

$\oplus$ es el operador de cruce que funciona de la siguiente manera, partiendo de $x_i(t) = \{x_{i1}^t, x_{i2}^t, \dots, x_{i(n+1)}^t\}$ y $p_i(t) = \{p_{i1}^t, p_{i2}^t, \dots, p_{i(n+1)}^t\}$ la operación de cruce es:

$$x_i(t) \oplus p_i(t) = \{x_{i1}^t, p_{i(k+1)}^t, p_{i(k+2)}^t, \dots, p_{i(k+m)}^t, x_{il}^t, x_{i(l+1)}^t, \dots, x_{i(n+1)}^t\}$$

Donde k un número entero aleatorio [1, n-1] que se halla cada vez que se realiza una operación de cruce y $x_{il}^t, x_{i(l+1)}^t$,... significa que se cogen todas las posiciones

de la trayectoria $x_i(t)$, en orden, que no hayan sido cogidas ya de $p_i(t)$, es decir, que no estén en $(p^t_{i(k+1)}, \dots, p^t_{i(k+m)})$ puesto que no se pueden repetir ciudades.

La operación de cruce hay que realizarla en orden, en primer lugar se cruza $x_i(t)$ con $p_i(t)$ y después el resultado obtenido del cruce anterior, se cruza con $g(t)$, obteniendo x'_i.

Nótese además que la ciudad de partida y la de llegada no se cambian, todas las operaciones se realizan sobre el resto de coordenadas. Además en caso de que k+m resulte ser mayor que n, lo que se hace es coger desde k hasta n (inclusive) y se siguen cogiendo las que falten a partir de la segunda coordenada.

Dada m = 5 y las posiciones de la partícula i, $x_i(t) = \{1, 8, 7, 3, 2, 6, 9, 4, 5, 1\}$, $p_i(t) = \{1, 7, 3, 2, 4, 6, 9, 8, 5, 1\}$ y $g(t) = \{1, 4, 8, 6, 7, 2, 9, 5, 3, 1\}$, un ejemplo de su operación de cruce completa es:

- Se halla k → $k = 2$ → $p_i(t) = \{1, 7, \underbrace{3, 2, 4, 6, 9}_{K+1 \quad k+m}, 8, 5, 1\}$ → $x_i(t) \oplus p_i(t) = \{1, 3, 2, 4, 6, 9, 8, 7, 5, 1\}$

- Se halla de nuevo k → $k = 6$ → $g(t) = \{1, \underbrace{4, 8}_{k+m}, 6, 7, 2, \underbrace{9, 5, 3}_{K+1}, 1\}$ → $x'_i(t) = \{1, 9, 5, 3, 4, 8, 2, 6, 7, 1\}$

4.1.1.2 Operación de inyección u operación inversa

En esta operación lo que hacemos es realizar una serie de permutaciones sobre la posición x'_i obtenida en el paso anterior. La operación se representa así:

$$x_i(t+1)' = \text{Inyección}\,(x'_i(t), Vc) \;||\; \text{Inversa}\,(x'_i(t), s, e)$$

Lo que significa || es que para cada partícula sólo se realiza una operación, o la inyección o la inversa, cada una con una probabilidad del 50% de ser escogida. Para determinar cuál de las dos se lleva a cabo lo que hacemos es hallar un número aleatorio [1, 100] y si está por debajo de 50 se realiza la inyección y en caso contrario se realizará la inversa. Estas operaciones consisten en:

- Operación de inyección
  Para realizar esta operación partimos del vector $x'_i(t)$ y lo primer que se hace es seleccionar el punto de inyección de entre todas sus coordenadas. El método para hacerlo es el siguiente:

- Hallar la probabilidad de selección para cada coordenada j de $x_i'(t)$, utilizando la fórmula: $Pj = d(x'_{ij}, x'_{i(j+1)}) / \sum_{j=1}^{n} d(x'_{ij}, x'_{i(j+1)})$ lo que significa que la probabilidad de seleccionar la ciudad que está en la ciudad que está en j como punto de inyección es igual a la distancia que hay de esa ciudad a la siguiente (en la trayectoria $x_i'(t)$), dividido por la distancia total del recorrido $x_i'(t)$.
  Lo que quiere decir que cuanto más lejos estén dos ciudades consecutivas, más probabilidad hay de escoger la primera de ellas como punto de inyección.

- Después se halla la probabilidad acumulada, de cada coordenada y se genera un número aleatorio. Dependiendo del valor de aleatorio, se seleccionará una coordenada u otra según esas probabilidades acumuladas. Esa coordenada será el punto de inyección y la ciudad correspondiente (el valor de la coordenada) es la ciudad c.

Ahora que ya tenemos el punto de inyección, lo que hacemos es calcular el vector Vc = $\{v_{c_1}, v_{c_2}, \dots, v_{c_m}\}$, que es la secuencia que indica la ciudad más cercana para cada ciudad, tomando como partida la ciudad c. Dicho de otro modo, c_1 es la más cercana a c , $c_2$es la más cercana a c_1...la más cercana a c_m está entre c_{m-1} y c_1. La variable m es la longitud de cruce utilizada también en la operación anterior (utiliza el mismo valor durante toda la ejecución y es el mismo para una operación que para la otra).
Partiendo del vector $x_i'(t)$ y del Vc = $\{v_{c_1}, v_{c_2}, \dots, v_{c_m}\}$ calculado, se hace la operación de inyección en dos pasos:

- Inyección (x_i'(t), Vc) = $\{x_{i1}, x_{i2}, \dots, c, v_{c_1}, v_{c_2}, \dots, v_{c_m}, \dots, x_{i(n+1)}\}$

- Eliminar todas las ciudades x_{ij} que estén repetidas, es decir quitar las ciudades correspondientes al vector Vc.

A continuación se muestra un ejemplo de una operación de inyección para m = 5, la posición anterior hallada en el cruce $x_i'(t)$= {1, 9, 5, 3, 4, 8, 2, 6, 7, 1} y V_5={8, 3, 7, 4, 9} :

- Inyección (x_i', V_5) = {1, 9, 5, 8, 3, 7, 4, 9 3, 4, 8, 2, 6, 7, 1}

- Inyección (x_i', V_5) = {1, 9, 5, 8, 3, 7, 4, 9 3, 4, 8, 2, 6, 7, 1} = {1, 5, 8, 3, 7, 4, 9, 2, 6, 1} = $x_i(t+1)'$

- Operación inversa

Partimos de vector $x_i'(t)$ que obtuvimos en la operación de cruce y de dos números, s y e, aleatorios entre [2, n] (para evitar cambiar la ciudad de partida y llegada) cuya única condición es que e sea mayor que s.
La operación consiste en invertir el intervalo de coordenadas, que van desde la coordenada de la posición s hasta la de la posición e, del vector $x_i'(t)$ de partida. Se hace de la siguiente forma:

Dado $x_i' = \{x_{i1}, x_{i2}, \dots, x_{is}, x_{i(s+1)}, \dots, x_{i(e-1)}, x_e, \dots, x_{i(n+1)}\}$

Reverse (x_i', s, e) = $\{x_{i1}, x_{i2}, \dots, x_{ie}, x_{i(e-1)}, \dots, x_{i(s+1)}, x_s, \dots, x_{i(n+1)}\}$

Un ejemplo de operación inversa para $x_i'(t) = \{1, 9, 5, 3, 4, 8, 2, 6, 7, 1\}$ es:

- Se generan dos número aleatorios → $s = 3$ y $e = 7$ → $x_i'(t) = \{1, 9, \underset{s}{5}, 3, 4, 8, \underset{e}{2}, 6, 7, 1\}$
- Reverse (x_i', 3, 7) = $\{1, 9, \underset{e}{2}, 8, 4, 3, \underset{s}{5}, 6, 7, 1\} = x_i(t+1)'$

4.1.1.3 Operación de adaptación al ruido

Esta operación parte del vector $x_i(t+1)'$ generado por la operación inyecta||inversa y su modo de actuar depende de la diversidad, entendiendo por diversidad a la similitud que existe entre las diferentes posiciones, $x_i(t+1)'$, $p_i(t)$ y $g(t)$ de la partícula, denotado por $diversidad_i$. La operación es:

$$x_i(t+1) = \eta \times disturb(x_i\,(t+1)', k)$$

La variable η es binaria [0, 1] y hay que calcularla en función de la diversidad de la siguiente manera:

- En primer lugar hay que hallar la $diversidad_i$ siguiendo los pasos:
  - Hallamos las similitudes $s_{x_i(t+1)', p_i(t)}$, $s_{x_i(t+1)', g(t)}$ y $s_{p_i(t), g(t)}$ que hay entre las diferentes posiciones de la partícula, teniendo en cuenta que S_{ij} es la similitud que hay entre i y j. La similitud se calcula así: $S_{ij} = \frac{1}{(n-1)} \sum_{k=2}^{n} \delta \rightarrow si(x_{ik} == x_{jk});\ \delta = 1;\ sino\ \delta = 0$, lo que quiere decir que se compara cada coordenada, k, de los vectores i y j, y si coinciden δ vale 1, luego hace la media de todos los δ, obteniendo S_{ij} en un intervalo de [0.0, 1.0].
    En nuestro caso, por poner un ejemplo, $s_{x_i(t+1)', p_i(t)}$, será un valor entre 0 y 1, que en el caso de que valiese 0,8 quiere decir que el 80% de las ciudades que forman el problema son recorridas en la

misma posición de la trayectoria $x_i(t+1)'$, y de la trayectoria $p_i(t)$. Si valiera 1, significaría que la trayectoria $x_i(t+1)'$, y la trayectoria $p_i(t)$ son exactamente iguales.

- Finalmente calculamos la diversidad de la partícula i con la fórmula: $diversidad_i = \frac{1}{3}(s_{x_i(t+1)',p_i(t)} + s_{x_i(t+1)',g(t)} + s_{p_i(t),g(t)})$.

- Asignamos el valor a η de la siguiente manera:

  - Si $diversidad_i$ < 0,4; entonces η = 1

  - Si $diversidad_i \geq$ 0,4; entonces η = 0

  Esto quiere decir que si la diversidad $\geq$ 0,4, significa que la posición $x_i(t+1)'$ es muy parecida a $p_i(t)$ y $g(t)$ y las dos últimas parecidas entre ellas también, por eso no se aplica la operación de ruido (η = 0), para no cambiar la partícula ya que es de las mejores de la población.

La operación $disturb(x_i\,(t+1)', k)$, donde k es un número entero aleatorio que generamos entre [2, n] y $x_{ik}{}'$ es la ciudad más cercana a x_{ik} que tenemos que calcular, funciona de la siguiente forma:

Dado $x_i(t+1)' = \{x_{i1}, x_{i2}, \dots, x_{ik}, \dots, x_{i(n+1)}\}$

disturb(X_i', k)= $\{x_{i1}, x_{i2}, \dots, x_{ik}{}', \dots, x_{i(n+1)}\}$, y donde estaba $x_{ik}{}'$ ponemos x_{ik}

Esto quiere decir que la operación de ruido consiste en intercambiar la ciudad que está en la coordenada k, por la ciudad más próxima a ella, si y sólo si hay poca similitud entre la trayectoria de la partícula y los las trayectorias óptimas. Tras realizar la operación de diversidad obtenemos $x_i(t+1)$ que es la nueva posición actual de la partícula que utilizaremos como $x_i(t)$ en la siguiente iteración y coincidirá con $x_i(t+1)'$ en caso de que η = 0.

Un ejemplo de una operación de ruido para $x_i(t+1)' = \{1, 9, 2, 8, 4, 3, 5, 6, 7, 1\}$, $p_i(t) = \{1, 7, 3, 2, 4, 6, 9, 8, 5, 1\}$ y $g(t) = \{1, 4, 8, 6, 7, 2, 9, 5, 3, 1\}$, es el siguiente:

- En primer lugar se calcula la similitud entre cada una de esas posiciones (sin tener en cuenta la ciudad de partida y llegada porque es constante) contando cuantas coordenadas coinciden.

  - $s_{x_i(t+1)',p_i(t)}$ = 1/8 = 0,125. Como vemos, sólo coincide la ciudad 4.

- $s_{x_i(t+1)',g(t)}$ = 0/8 = 0. No hay ciudades en la misma posición.
- $s_{p_i(t),g(t)}$ = 1/8 = 0,125. Sólo coincide la ciudad 9 en 7ª posición.

- Hallamos la diversidad de la partícula y con ello η:

$$diversidad_i = \frac{1}{3}(0{,}125 + 0 + 0{,}125) = 0{,}083 < 0{,}4 \rightarrow \eta = 1$$

- Realizamos la operación de ruido:
  - Generamos un número aleatorio → k = 6, en la posición 6 está la ciudad 3 → supongamos que la ciudad más cercana a la 3 es la 9.
  - $x_i(t+1) = 1 \times disturb(x_i\,(t+1)', k)$ = {1, 3, 2, 8, 4, 9, 5, 6, 7, 1}.

Ahora que ya sabemos cómo queremos realizar el programa, de forma teórica, en el siguiente apartado exponemos como se llevaron a la práctica este conjunto de pasos.

4.2 Algoritmo desarrollado

El algoritmo lo hemos desarrollado en el lenguaje de programación C++, utilizando para ello el entorno de desarrollo CodeBlocks (tomado de la web [11]).

El programa internamente funciona llamando a funciones, es decir desde el archivo principal (main) se van nombrando las funciones o procedimientos que están definidas en otros archivos que forman el programa y se ejecutan.

En nuestro caso utilizaremos tres archivos diferenciados:

- El principal fichero principal, también conocido como main, es donde vamos diciendo al programa los pasos de debe ir haciendo, uno tras otro.
- La clase Fichero está compuesta por varios procedimientos y funciones que sirven para cargar los datos que nos ofrece el enunciado del problema y guardarlos en variables y matrices para su posterior utilización.
- Finalmente la clase Algoritmo, posee las funciones y los procesos necesarios para la implementación del PSO al ATSP, es decir, está compuesta de todos los procedimientos necesarios para generar una

solución (e imprimirla), a partir de los de los datos del problema que le proporción la clase Fichero.

El programa crea tres ficheros de salida de extensión ".txt" con los siguientes valores:

- "RESULTADO.txt" almacena el nombre del enunciado y el resultado final Gbest de la ejecución completa.

- "VariablesUtilizadas.txt" tiene guardadas todas las variables que establece el usuario para cada ejecución del programa y tiene el aspecto que vemos en la figura 10.

```
ENUNCIADO: br17
El numero de ciudades a recorrer es: 17

Numero de particulas que forma la poblacion: 50
La ciudad de partida del recorrido es: 0
La longitud de cruce es: 3
El numero de iteraciones es: 500
```

Figura 10. "VariablesUtilizadas.txt". Elaboración propia.

- "SalidaCompletaResultados.txt" tiene almacenadas todas las variables que establece el usuario, igual que en el apartado anterior, pero además posee:

  - La matriz de distancias del problema

  - El recorrido actual de cada partícula en la iteración inicial, así como los recorridos p_i y el mejor de ellos, el g. Todos los recorridos se muestran con sus respectivos fitness.

  - Para cada iteración de la ejecución se muestra la trayectoria g obtenida y su distancia total, Gbest.

  - Para concluir el fichero se ponen los recorridos x_i y p_i para cada partícula y el recorrido final g con sus respectivos valores.

Las variables más importantes que utiliza el algoritmo son las que están descritas a continuación:

- Cadena "nombreFichero" donde se almacena el enunciado que hay que resolver.

- Variable “dimension” (n) que almacena la cantidad de ciudades que posee la instancia que estamos resolviendo.

- Matriz “distancias[i][j]” de tamaño (n x n), donde está recogida la distancia que hay de la ciudad cada ciudad i a cada ciudad j.

- La variable “particulas” indica el número de individuos que forman la población.

- La variable “ciudadPartida” es la ciudad de la que parte el agente viajero y a la que debe volver.

- “m” es la longitud de cruce, que nos sirve para determinar, tanto el funcionamiento de la operación de cruce, como el de la de inyección.

- “iteracionesMax” establece el criterio de parada, en ella se almacena la cantidad de iteraciones que debe de hacer el programa hasta detenerse y dar la solución.

- La variable “t” almacena la iteración actual en la que se encuentra el algoritmo, está será la que se compare con “iteracionesMax” para comprobar si se ha cumplido el criterio de parada.

- La matriz “recorridoActual [i][j]” es de tamaño (partículas x (n+2)), donde las filas son las partículas que forman la población y en las columnas de [1] a [n+1] ponemos la trayectoria actual x_i que sigue la partícula i, reservando la última posición ([n+2]) para almacenar el fitness que supone esa trayectoria.

- “recorridoPbest [i][j]” está estructurada igual que la anterior, solo que en este caso en vez de guardar el recorrido actual de las partículas, se guarda el mejor recorrido encontrado por la partícula i hasta el momento, es decir de [1] a [n+1] tendremos p_i y en [n+2] estará $Pbest_i$.

- “recorridoGbest [j]” en este caso el recorrido se guarda en un vector, donde j=n+2. El vector almacena en (n+1) columnas el recorrido g y el Gbest en la [n+2], esto es lo mismo que decir que es una copia de una fila de “recorridoPbest [i][j]”, concretamente de la fila i que posea el mejor $Pbest_i$.

- El vector “x_1[j]” lo utilizamos a lo largo del algoritmo como vector auxiliar, para poder ir creando las nuevas posiciones de la partícula paso a paso sin sobrescribir la matriz “recorridoActual [i][j]”, ya que

perderíamos su información. Es de tamaño (n+1) y lo que se hace es ir almacenando la nueva trayectoria de la partícula i, que estamos actualizando, generada por la operación en que nos encontremos (ya sea cruce, inyección, inversión o ruido), y cuando ya esté completo el vector se vuelca la información en "recorridoActual [i][j]" y pasamos a hacer lo mismo para la siguiente partícula.

A continuación vamos a explicar, a grandes rasgos, la forma de proceder del algoritmo, mostrando para ello la secuencia de pasos establecida en el main:

- Llamada al procedimiento "cargarFichero" del siguiente modo: "fichero.cargarFichero", donde "fichero." significa que es un proceso de la clase fichero. Si fuese "algoritmo." significaría que el proceso está declarado en la clase algoritmo.
  En ese procedimiento lo que hacemos es pedir por pantalla al usuario que introduzca el enunciado que quiere resolver y almacenar lo que escribe en la cadena "nombreFichero".
  A continuación se abre el fichero, y de él se obtiene:

  - El número de ciudades que posee y lo guardamos en la variable "dimensión".

  - La matriz de distancias, que se almacena en "distancias[i][j]".

- "fichero.mostrarMatriz" en este paso se imprime en el fichero de salida "SalidaCompletaResultados.txt" la matriz de distancias obtenida en el paso anterior y además se saca también por pantalla.

- "algoritmo.pedirDatos" el main llama así al procedimiento "pedirDatos" de la clase Algoritmo donde se pide al usuario por pantalla:

  - El número de partículas que forma la población, que se almacena en la variable "partículas".

  - La ciudad de partida de la trayectoria. Su valor se almacena en "ciudadPartida".

  - La longitud de cruce que se guarda en la variable "m".

  - El número de iteraciones máximas que determina cuándo va a finalizar el algoritmo. Su valor se guarda en "iteracionesMax".

Todos esos valore son imprimidos por pantalla y además se guardan en los ficheros de salida "SalidaCompletaResultados.txt" y "VariablesUtilizadas.txt".

- Ejecutamos "algoritmo.generarInicial" donde se inicializan:

  - La matriz que almacena las trayectorias actuales de cada partícula, "recorridoActual[i][j]". Para ello ponemos la posición [i][1] = [i][n+1] = "ciudadPartida" y el resto de las posiciones se rellenan aleatoriamente entre las ciudades que no forman parte aún de la trayectria.
    La posición [i][n+2] almacena el fitness de la trayectoria que va desde j=1 a j=n+1. Para ello sumamos las distancias que hay de la ciudad [i][j] a [i][j+1], para j=1, 2, …, n

  - "recorridoPBest[i][j]", en la primera iteración esta matriz es exactamente igual que "recorridoActual[i][j]" por lo que copiamos sus valores.

  - "recorridoGBest[j]", para actualizarla comparamos los fitness (posición [i][n+2]) de todas las partículas, hallando la que posee menor valor. Después otorgamos a "recorridoGBest[j]" la trayectoria "recorridoPBest[i][j]" (i= partícula con mejor valor).

- "algoritmo.mostrarGbest" este procedimiento lo que hace es imprimir por pantalla y en el fichero "SalidaCompletaResultados.txt" el vector "recorridoGBest[j]" completo, es decir, imprime g y su valor Gbest (posición [i][n+2]).

- Comienzan las iteraciones partimos de t = 0 → mientras "t"<"iteracionesMax" → t = t+1 {

  - "algoritmo.crucePbest" este paso realiza la operación de cruce de "recorridoActual[i][j]" con "recorridoPbest[i][j]" para cada partícula y el procedimiento es análogo al descrito en el apartado 4.1.1.1.
    Para realizar la operación usamos el vector auxiliar $x_1(j)$. Utilizamos este vector, en lugar de la matriz de recorrido actual, para ir rellenando las nuevas posiciones una a una sin sobrescribir la matriz original ya que perderíamos su trayectoria y nos hace falta para acabar de rellenar el recorrido después de poner los valores que se crucen de la matriz de recorrido Pbest.

Cuando ya tenemos el vector auxiliar completo actualizamos la posición de la partícula → "recorridoActual [i][j]" = x_1[j].

- "algoritmo.cruceGbest" es un procedimiento muy parecido al anterior (es la segunda parte del apartado 4.1.1.1), donde partimos de la trayectoria "recorridoActual [i][j]" que es la consecuencia de aplicar el "crucePbest" y la cruzamos en este caso con "recorridoGbest [j]". Al terminar el proceso volvemos a actualizar las partículas → "recorridoActual [i][j]" = x_1[j].

- "algoritmo.InjectOrReverse" determina si se realiza la operación de inyección o la de inversión. Para ello haya un número aleatorio y si es menor que 51 llama realiza la operación de inyección y en caso contrario la inversa.
  Al igual que las operaciones de cruce, estas también utilizan el vector x_1[j] y su operación se realiza como se describe en 4.1.1.2. Cuando ya se ha concluido la operación se vuelca el vector auxiliar en la matriz de recorrido actual de la partícula.

- "algoritmo.Ruido" la llamada a este proceso hace que se realice la operación de adaptación al ruido del modo descrito en 4.1.1.3 almacenando el resultado de dicha operación en x_1[j]. Como ya sabemos, dependiendo del valor de diversidad que obtengamos algunas partículas se modificarán y otras (las más cercanas a los puntos óptimos) no mutarán. Al finalizar el proceso para cada partícula se almacena x_1[j] en "recorridoActual [i][j]".

- "algoritmo.actualizarPbest", este proceso está dividido en dos etapas.
  En primer lugar se calcula el fitness del recorrido actual de cada partícula y se almacena en "recorridoActual [i][n+2]".
  Y en segundo lugar esa posición se compara con su análoga de la matriz de recorrido $Pbest_i$ y en caso de que sea mejor, se sustituye "recorridoPbest [i][j]" por "recorridoActual [i][j]" para dicha partícula, si fuera peor no se modificaría el recorrido Pbest.

- "algoritmo.actualizarGbest", este procedimiento es más simple que el anterior porque ya tenemos calculado el fitness. Lo que se hace es comparar el valor $Pbest_i$ (obtenido en el paso anterior) de cada partícula con el valor Gbest.

Si es mejor el $Pbest_i$ de una partícula i que Gbest, entonces "recorridoGbest [j]" = "recorridoPbest [i][j]", en caso contrario conservamos el Gbest que tenemos.

- "algoritmo.mostrarGbest"

} Fin de la iteración. Si se sigue cumpliendo que "t"<"iteracionesMax", se realiza otra iteración y en caso de que se haya alcanzado el criterio de parada pasamos al siguiente procedimiento.

- Se acaba la ejecución y se cierra el programa.

4.3 Resultados experimentales

Una vez el programa estuvo terminado, se pasó a evaluar su comportamiento frente a 19 enunciados ATSP sacados del apartado "Asymmetric traveling salesman problem" de la biblioteca TSPLIB [12], que contiene tanto los enunciados mencionados (presentados como en la figura 11), como los mejores resultados encontrados hasta el momento para esas instancias.

```
NAME:  br17
TYPE: ATSP
COMMENT: 17 city problem (Repetto)
DIMENSION:  17
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
 9999    3    5   48   48    8    8    5    5    3    3    0    3
5    8    8
    5
    3 9999    3   48   48    8    8    5    5    0    0    3    0
3    8    8
    5
    5    3 9999   72   72   48   48   24   24    3    3    5    3
0   48   48
   24
   48   48   74 9999    0    6    6   12   12   48   48   48   48
74    6    6
                                  ...

    5    5   26   12   12    8    8    0    0    5    5    5    5
26    8    8
 9999
EOF
```

Figura 11. Enunciado del problema br17. [12].

En la tabla 4 se muestra el enunciado de cada uno de los 19 problemas ATSP de la librería TSPLIB con su correspondiente dimensión y valor óptimo.

Enunciados	Br17	ftv33	ftv35	ftv38	p43	ftv44	ftv47	ry48p	ft53	ftv55
Dimensión	17	34	36	39	43	45	48	48	53	56
Valor óptimo	39	1286	1473	1530	5620	1613	1776	14422	6905	1608
Enunciados	ftv64	ft70	ftv70	kro124p	ftv170		rbg323	rbg358	rbg403	rbg443
Dimensión	65	70	71	100	171		323	358	403	443
Valor óptimo	1839	38673	1950	36230	2755		1326	1163	2465	2720

Tabla 4. Librería TSPLIB. Elaboración propia.

4.3.1 Parámetros utilizados

Para resolver cada una de las instancias se han tenido que seleccionar los siguientes parámetros, que como ya dijimos, dejamos a elección del usuario debido a la falta de conocimiento sobre su valor óptimo:

- Número de partículas: 50.
  En este estudio hemos decidido establecer un valor fijo para el tamaño de la población. La elección de un número pequeño de partículas está fundamentada en el hecho de que, según parece, los algoritmos iterativos avanzan más aumentando el número de iteraciones, que aumentando el número de partículas, y puestos a elegir el modo de distribución de nuestros recursos, elegimos poner este valor pequeño y la cantidad de iteraciones grande para permitir una evolución mayor de la solución, optimizando en lo posible el tiempo de ejecución.

- Ciudad de partida: 0.
  Puesto que el enunciado no nos dice de qué punto debe partir la trayectoria, y sabemos que el punto de partida no afecta a la solución final por tratarse de un ciclo Hamiltoniano, ponemos este valor fijo eligiendo para esta variable la primera ciudad de la matriz de distancias.

- Longitud de cruce: 20%, 30%, 40%, 50%, 60% y 70%.
  La variable longitud de cruce la definimos en función de la dimensión de la instancia a resolver (una longitud de cruce del 20% significa que m=0,2*n), ya que para un problema de 17 ciudades, hacer que se crucen 15 es cambiar casi por completo la trayectoria, mientras que para un problema de 443 ciudades ese cruce sería prácticamente insignificante.

Para cada enunciado vamos a utilizar esta variable con los 6 valores porcentuales especificados (entre 20% y el 70% de la dimensión del problema) y así podremos comprobar con nuestra experimentación si este algoritmo funciona mejor para longitudes de cruce altas o bajas.

- Iteraciones máximas: 500, 1.000 y 5.000 // 5.000, 10.000 y 30.000.
  Para los 15 problemas de la librería, de dimensión relativamente pequeños (hasta dimensión 171, que equivale al problema ftv170) las iteraciones que utilizamos son: 500, 1.000 y 5.000.
  Los problemas rbg323, rbg358, rbg403 y rbg443, tienen dimensiones muy altas, por lo que necesitarán más iteraciones para poder obtener una buena solución. En este caso se utilizarán los valores 5.000, 10.000 y 30.000.

Para poder hacer una comparación estadística adecuada, cada una de las combinaciones posibles de esos parámetros (hay 18 combinaciones para cada enunciado) se ha ejecutado 30 veces para cada instancia, ya que si se ejecuta sólo una vez el resultado de la experimentación no será significativo debido al factor estocástico que envuelve al método utilizado.

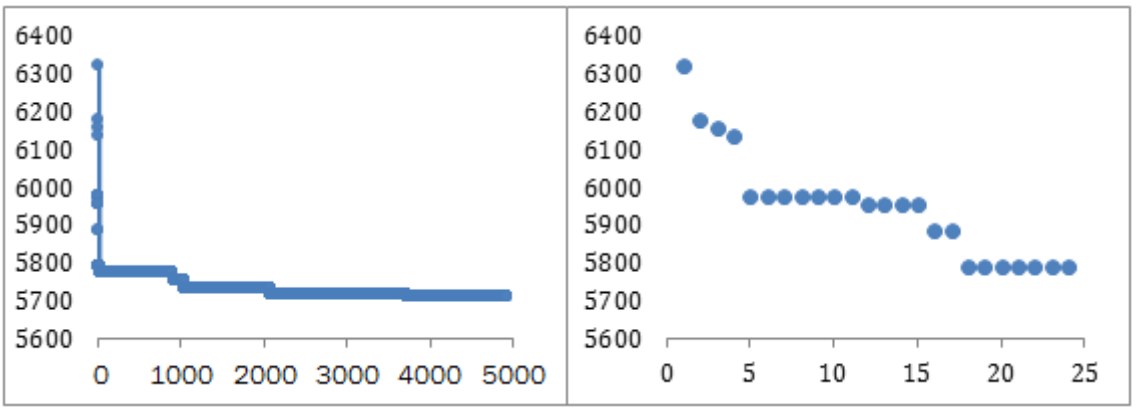
4.3.2 Resultados

A continuación vamos a mostrar una serie de gráficas que muestran la evolución completa de una determinada ejecución (1 de las 30) del enunciado correspondiente, por lo que se va marcando el valor Gbest correspondiente a la iteración 1, 2,..., iteración máxima.

En la gráficas 1 y 2 se muestra la evolución para el enunciado p43 con un 20% de longitud de cruce y 5.000 iteraciones (cogemos 5.000 porque es el caso más genérico, también se puede ver de forma aproximada su valor en 500 y en 1.000). Esta ejecución concreta que hemos escogido parte de una distancia total (Gbest) de 6.323 en la primera iteración y acaba con un resultado de 5.709, nótese que en la iteración 25 ya se ha alcanzado el valor 5.788 de Gbest.

En la primera podemos observar el proceso iterativo completo, y de él concluir que la mayor evolución se da para las primeras iteraciones, por ello en la gráfica 2 mostramos las 25 primeras iteraciones aisladas para observar con detalle la gran mejora de Gbest que hay entre la primera ejecución (aleatoria) y el Gbest obtenido tras unas pocas iteraciones.

Ese mismo fenómeno se ha observado en las gráficas con valores de 30%, 40%, 50%, 60% y 70% de este mismo problema.



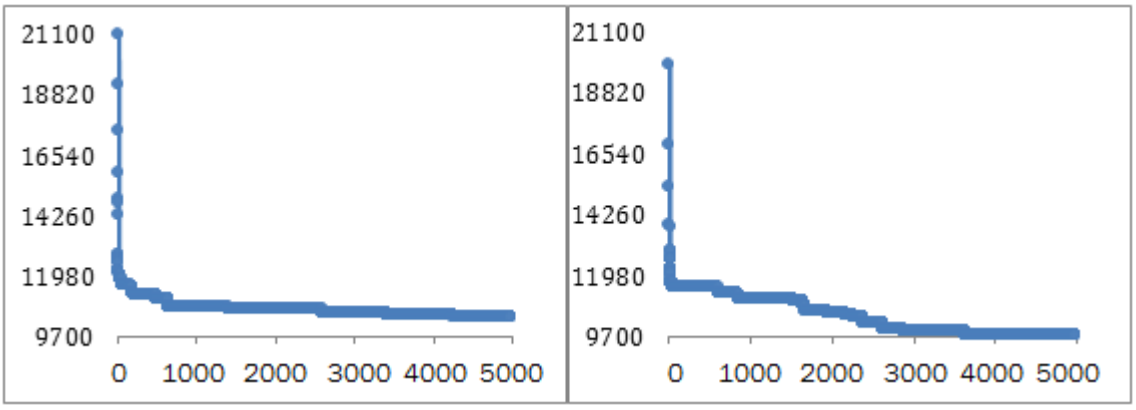
Gráficas 1 y 2. Evolución del resultado Gbest de p43 para 5.000 iteraciones y longitud de cruce del 20%. Elaboración propia.

Dado que el problema p43 nos da un valor muy cercano al óptimo, en todas las combinaciones de variables, no hay gran dependencia del resultado obtenido con los porcentajes de longitud de cruce utilizados.

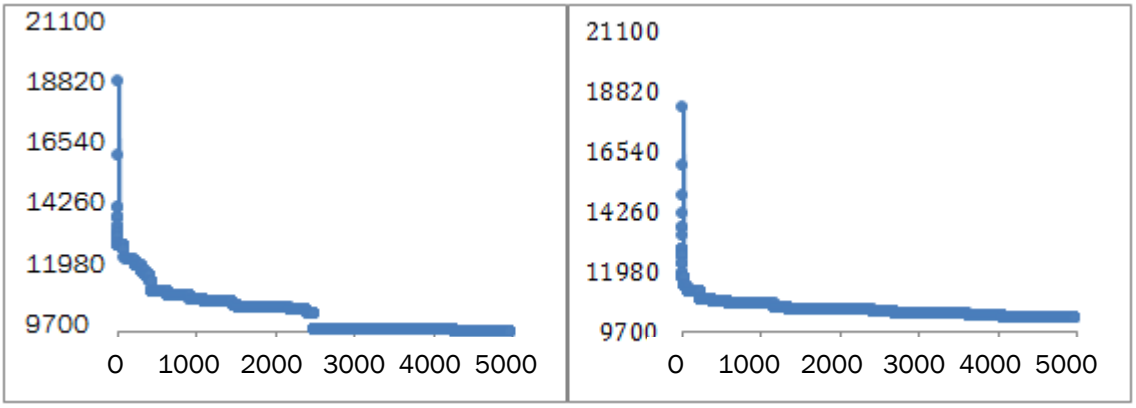
Para poder comprobar si se puede ver la evolución del resultado en función de la longitud de cruce exponemos las gráficas 3, 4, 5, 6, 7 y 8 de la evolución del problema ft170 con porcentajes del 20, 30, 40, 50, 60 y 70 respectivamente.

Lo que podemos decir tras observar estas gráficas es que la dependencia del resultado con la longitud de cruce no parece tan clara como la dependencia con la cantidad de iteraciones realizadas.

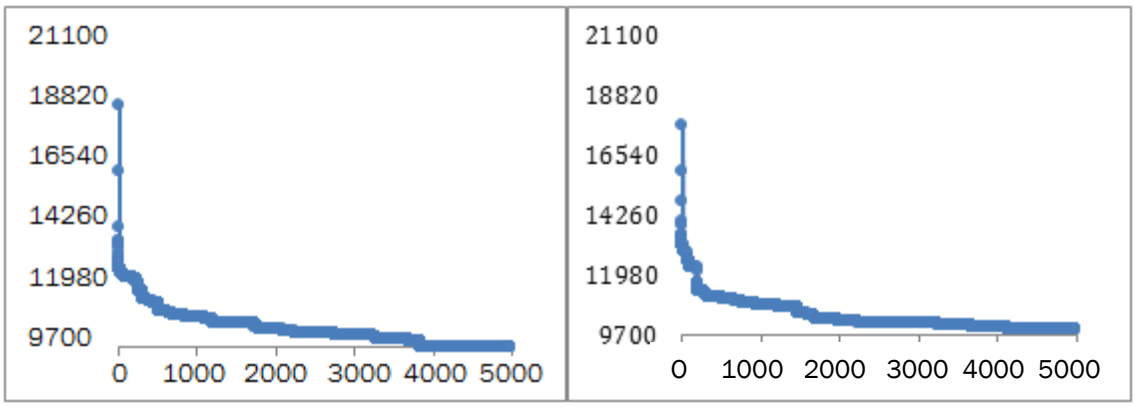
De hecho, de estas 6 gráficas concretas no podríamos concluir si es mejor una longitud de cruce mayor o menor, ya que se puede observar claramente que su progreso es estocástico. Por esa razón, para hacer el estudio estadístico ejecutamos cada instancia 30 veces, para poder obtener una media objetiva de la que sí que se podrán sacar conclusiones.



Gráficas 3 y 4. Evolución de Gbest de ft170 para 5.000 iteraciones y longitud de cruce del 20% y 30% en las respectivas gráficas. Elaboración propia.



Gráficas 5 y 6. Evolución de Gbest de ft170 para 5.000 iteraciones y longitud de cruce del 40% y 50% en las respectivas gráficas. Elaboración propia.



Gráficas 7 y 8. Evolución de Gbest de ft170 para 5.000 iteraciones y longitud de cruce del 60% y 70% en las respectivas gráficas. Elaboración propia.

Para poder seguir con el análisis de los resultados, vamos a hablar a partir de ahora, de resultados medios, en lugar de utilizar uno concreto.

Para cada una de las posibles combinaciones de cada enunciado se obtienen 30 resultados en su archivo "RESULTADO.txt" correspondiente. Para poder manipular de una forma cómoda estas soluciones, se han pasado los resultados de cada problema a una tabla en Excel.

El formato de la tabla completa es el que vemos en la tabla 5 para el problema concreto p43, donde la zona azul clara muestra los resultados obtenidos para 500 iteraciones, la del medio para 1.000 iteraciones y la más oscura para 5.000.

Mejor resultado de p43	Nº Partículas que forma la población - Nº de Iteraciones (Criterio de parada) - Longitud de cruce en función de el tamaño del problema																	
5620	50-500-20%	50-500-30%	50-500-40%	50-500-50%	50-500-60%	50-500-70%	50-1.000-20%	50-1.000-30%	50-1.000-40%	50-1.000-50%	50-1.000-60%	50-1.000-70%	50-5.000-20%	50-5.000-30%	50-5.000-40%	50-5.000-50%	50-5.000-60%	50-5.000-70%
Gbest Ejecución 1	5745	5703	5679	5703	5684	5688	5750	5720	5703	5664	5715	5711	5709	5689	5652	5681	5703	5631
Gbest Ejecución 2	5746	5719	5728	5688	5739	5713	5758	5724	5725	5711	5689	5639	5693	5713	5660	5678	5659	5673
Gbest Ejecución 3	5764	5737	5742	5723	5716	5681	5684	5692	5684	5686	5719	5702	5708	5697	5689	5673	5717	5650
Gbest Ejecución 4	5759	5720	5730	5755	5684	5689	5728	5714	5703	5699	5682	5676	5691	5666	5697	5674	5645	5668
Gbest Ejecución 5	5779	5727	5733	5718	5705	5690	5726	5697	5670	5718	5707	5680	5711	5693	5678	5672	5721	5656
Gbest Ejecución 6	5757	5717	5738	5685	5730	5703	5745	5677	5717	5716	5693	5722	5704	5676	5663	5662	5652	5663
Gbest Ejecución 7	5740	5702	5686	5688	5705	5707	5715	5708	5680	5722	5726	5669	5672	5682	5661	5697	5667	5644
Gbest Ejecución 8	5765	5699	5738	5696	5731	5690	5764	5729	5679	5692	5714	5663	5717	5707	5656	5683	5683	5703
Gbest Ejecución 9	5725	5720	5737	5687	5709	5699	5728	5731	5700	5702	5688	5708	5724	5672	5644	5678	5691	5665
Gbest Ejecución 10	5745	5735	5690	5694	5697	5712	5710	5683	5670	5692	5703	5703	5711	5700	5650	5662	5676	5704
Gbest Ejecución 11	5749	5714	5757	5713	5723	5715	5683	5723	5742	5715	5674	5684	5685	5684	5693	5671	5655	5656
Gbest Ejecución 12	5726	5759	5749	5678	5712	5696	5723	5723	5693	5698	5686	5655	5715	5667	5677	5680	5661	5661
Gbest Ejecución 13	5725	5723	5717	5714	5732	5732	5748	5718	5733	5712	5698	5711	5716	5673	5720	5682	5654	5662
Gbest Ejecución 14	5723	5721	5705	5716	5709	5736	5717	5715	5694	5696	5729	5702	5703	5670	5682	5658	5660	5665
Gbest Ejecución 15	5736	5735	5725	5719	5727	5663	5727	5708	5671	5680	5690	5707	5719	5677	5691	5660	5675	5654
Gbest Ejecución 16	5733	5736	5687	5733	5704	5670	5745	5712	5668	5691	5706	5704	5707	5703	5692	5686	5679	5644
Gbest Ejecución 17	5776	5729	5703	5745	5716	5709	5714	5738	5765	5695	5690	5694	5689	5708	5665	5668	5664	5666
Gbest Ejecución 18	5730	5735	5725	5693	5684	5712	5718	5709	5705	5701	5720	5666	5725	5662	5698	5656	5664	5666
Gbest Ejecución 19	5781	5722	5697	5711	5698	5650	5697	5732	5729	5720	5730	5662	5670	5686	5643	5666	5682	5659
Gbest Ejecución 20	5758	5737	5726	5711	5718	5734	5743	5710	5714	5712	5674	5710	5727	5694	5654	5658	5659	5655
Gbest Ejecución 21	5786	5727	5676	5727	5716	5702	5737	5737	5714	5679	5710	5709	5712	5682	5641	5671	5665	5622
Gbest Ejecución 22	5753	5727	5706	5706	5687	5726	5728	5733	5676	5727	5706	5676	5724	5672	5645	5683	5685	5666
Gbest Ejecución 23	5768	5741	5708	5750	5699	5702	5732	5734	5701	5687	5703	5698	5695	5681	5707	5658	5672	5653
Gbest Ejecución 24	5744	5688	5690	5688	5664	5655	5745	5724	5707	5686	5712	5683	5734	5680	5689	5644	5698	5647
Gbest Ejecución 25	5758	5740	5743	5678	5736	5694	5746	5744	5677	5704	5692	5713	5704	5681	5683	5705	5690	5663
Gbest Ejecución 26	5719	5719	5733	5696	5677	5708	5744	5750	5710	5711	5697	5679	5716	5686	5649	5666	5688	5694
Gbest Ejecución 27	5766	5717	5729	5717	5671	5717	5723	5714	5715	5691	5689	5673	5708	5689	5664	5666	5672	5663
Gbest Ejecución 28	5790	5685	5745	5692	5763	5738	5749	5758	5683	5715	5681	5703	5703	5676	5695	5669	5683	5686
Gbest Ejecución 29	5716	5714	5705	5709	5731	5684	5730	5671	5670	5695	5720	5678	5694	5721	5664	5684	5680	5676
Gbest Ejecución 30	5790	5736	5729	5713	5699	5704	5740	5735	5692	5716	5682	5708	5708	5694	5649	5666	5672	5659
Gbest Medio	5751,73	5722,8	5718,53	5708,2	5708,87	5700,63	5729,9	5718,77	5699,67	5701,1	5700,83	5689,6	5706,47	5686,03	5671,7	5671,9	5675,73	5662,47
Calidad: Error Medio %	2,34%	1,83%	1,75%	1,57%	1,58%	1,43%	1,96%	1,76%	1,42%	1,44%	1,44%	1,24%	1,54%	1,17%	0,92%	0,92%	0,99%	0,76%

Tabla 5. Resultados de la experimentación completa de p43. Elaboración propia.

En cada fila de la tabla se muestran los 30 resultados obtenidos en la ejecución y en cada columna se detallan los parámetros utilizados para dicha ejecución de la siguiente forma: "Número de partículas que forma la población – Número de iteraciones máximas realizadas (criterio de parada establecido) – Longitud de cruce en valor porcentual sobre la dimensión del problema".

En las penúltima fila se muestra el resultado (Gbest) medio de las 30 ejecuciones realizadas del problema para los parámetros descritos en la columna.

Y para terminar, la última fila muestra el error medio del algoritmo para esos parámetros. La forma en que se ha hallado ese valor medio es:

$$error\ medio = \frac{Gbest\ medio - Gbest\ óptimo}{Gbest\ óptimo} * 100$$

Donde el valor de Gbest óptimo utilizado es el que nos ofrece la librería TSPLIB y se puede observar en la esquina superior izquierda de la tabla.

Finalmente hemos elaborado una tabla resumen donde se muestra el error medio de cada uno de los problemas para cada una de sus variantes, es decir se muestra el error medio de cada de las instancias estudiadas. Se puede ver en la tabla 6.

Enunciado	n	Nº Partículas que forma la población - Nº de Iteraciones (Criterio de parada) - Longitud de cruce en función de el tamaño del problema																	
		50-500-20%	50-500-30%	50-500-40%	50-500-50%	50-500-60%	50-500-70%	50-1.000-20%	50-1.000-30%	50-1.000-40%	50-1.000-50%	50-1.000-60%	50-1.000-70%	50-5.000-20%	50-5.000-30%	50-5.000-40%	50-5.000-50%	50-5.000-60%	50-5.000-70%
Br17	17	0,09%	0,00%	0,60%	0,26%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
ftv33	34	58,06%	50,48%	42,94%	39,82%	38,74%	37,24%	51,26%	39,51%	38,03%	28,95%	30,64%	34,69%	39,45%	31,77%	23,63%	19,08%	19,63%	23,82%
ftv35	36	60,24%	46,31%	44,28%	39,34%	37,41%	39,61%	54,12%	40,20%	38,02%	29,35%	33,71%	31,86%	43,87%	27,92%	23,92%	20,98%	20,80%	23,19%
ftv38	39	63,07%	53,49%	45,15%	43,51%	44,47%	40,68%	58,49%	46,96%	38,25%	37,33%	36,73%	32,67%	46,44%	34,36%	23,72%	21,26%	22,64%	23,92%
p43	43	2,34%	1,83%	1,75%	1,57%	1,58%	1,43%	1,96%	1,76%	1,42%	1,44%	1,44%	1,24%	1,54%	1,17%	0,92%	0,92%	0,99%	0,76%
ftv44	45	83,88%	74,68%	68,40%	62,06%	59,62%	57,89%	78,63%	64,41%	55,63%	55,05%	52,97%	50,85%	64,06%	52,33%	41,85%	34,46%	37,73%	37,30%
ftv47	48	78,31%	72,64%	66,93%	62,47%	64,94%	62,91%	73,34%	67,30%	56,79%	55,93%	52,83%	58,76%	60,31%	53,73%	44,53%	37,48%	38,53%	42,90%
ry48p	48	52,45%	43,51%	38,04%	36,27%	32,87%	30,34%	49,76%	40,06%	34,48%	28,80%	26,99%	27,15%	42,56%	32,20%	27,60%	19,84%	16,52%	16,51%
ft53	53	79,56%	75,47%	72,19%	71,56%	67,17%	66,52%	74,87%	69,73%	64,89%	61,10%	60,03%	58,55%	62,58%	57,94%	47,63%	45,03%	42,26%	45,10%
ftv55	56	106,21%	97,14%	90,84%	82,18%	83,40%	85,50%	99,11%	89,16%	80,03%	68,31%	72,65%	72,36%	86,35%	73,08%	60,79%	57,44%	55,57%	54,94%
ftv64	65	124,14%	115,10%	108,39%	104,80%	101,35%	94,99%	118,02%	110,82%	99,62%	89,06%	89,17%	86,51%	104,73%	85,55%	77,23%	66,95%	69,88%	65,41%
ft70	70	38,74%	35,73%	35,28%	32,81%	32,01%	32,38%	36,55%	33,51%	31,80%	31,47%	30,26%	30,10%	32,72%	29,66%	27,52%	25,49%	25,07%	25,09%
ftv70	71	130,90%	123,81%	117,00%	110,78%	110,50%	108,97%	125,71%	117,09%	108,81%	99,61%	104,35%	92,37%	112,55%	99,99%	91,66%	73,28%	72,88%	70,42%
kro124p	100	107,89%	101,45%	92,33%	87,77%	80,41%	75,23%	104,02%	93,73%	90,26%	80,47%	71,57%	68,13%	95,16%	82,73%	74,06%	59,26%	53,54%	50,84%
ftv170	171	295,93%	292,73%	288,88%	291,29%	280,89%	263,25%	286,84%	282,57%	274,87%	270,40%	268,53%	247,78%	271,35%	264,85%	253,22%	235,17%	228,03%	210,93%
Calidad: Error Medio		85,45%	78,96%	74,20%	71,10%	69,02%	66,46%	80,84%	73,12%	67,53%	62,49%	62,12%	59,53%	70,91%	61,82%	54,55%	47,78%	46,94%	46,07%
		Nº Partículas que forma la población - Nº de Iteraciones (Criterio de parada) - Longitud de cruce en función de el tamaño del problema																	
		50-5.000-20%	50-5.000-30%	50-5.000-40%	50-5.000-50%	50-5.000-60%	50-5.000-70%	50-10.000-20%	50-10.000-30%	50-10.000-40%	50-10.000-50%	50-10.000-60%	50-10.000-70%	50-30.000-20%	50-30.000-30%	50-30.000-40%	50-30.000-50%	50-30.000-60%	50-30.000-70%
rbg323	323	223,41%	224,78%	215,08%	206,95%	196,08%	186,38%	214,89%	218,85%	207,65%	195,60%	182,63%	174,61%	203,97%	207,35%	191,05%	176,66%	165,40%	158,76%
rbg358	358	333,38%	330,59%	310,99%	289,54%	274,71%	271,28%	324,10%	316,73%	295,23%	275,74%	256,26%	251,79%	314,98%	298,11%	274,63%	248,71%	233,67%	228,75%
rbg403	403	146,74%	138,92%	128,82%	120,71%	115,64%	112,20%	142,55%	133,64%	121,23%	113,27%	107,70%	103,89%	135,25%	124,59%	111,12%	103,16%	95,99%	94,78%
rbg443	443	146,35%	137,92%	130,61%	124,92%	117,67%	112,85%	143,28%	134,14%	124,68%	117,23%	109,80%	107,34%	136,76%	124,69%	116,33%	107,82%	99,90%	97,49%
Calidad: Error Medio		212,47%	208,05%	196,38%	185,53%	176,03%	170,68%	206,21%	200,84%	187,20%	175,46%	164,10%	159,41%	197,74%	188,68%	173,28%	159,09%	148,74%	144,94%

Tabla 6. Resultados de la experimentación: Error medio. Elaboración propia.

A simple vista podemos observar, tanto caso por caso, como en el error medio, que cuantas más iteraciones realizamos, mejor resultado vamos obteniendo.

Mientras que de la longitud de cruce podemos decir que en valor medio, cuanto mayor es su porcentaje mejor solución obtenemos, pero no se puede generalizar para cada problema, ya que por ejemplo en el enunciado ftv35 es mejor el resultado que se obtiene para el 60% de longitud de cruce en el caso de 500 y 5.000 iteraciones y para el 50% en el caso de 1.000 que los sus resultados respectivos con longitud de cruce del 70%.

4.4 Estudio estadístico de resultados

Dado que con la simple observación de la tabla 6 ya podemos concluir que la mejor situación se da para el mayor número de iteraciones máximas (5.000 en el caso de los problemas pequeños y 30.000 para los grandes), esta variable no la analizaremos estadísticamente.

La variable longitud de cruce es la que vamos a analizar en este apartado con los estadísticos de Friedman y Holm, para determinar qué longitud es la más adecuada, es decir, ante qué valor se comporta mejor nuestro algoritmo.

Tanto el test de Friedman como el test a posteriori de Holm (García el al., 2009) son test no paramétricos (se basan en un orden y utilizan datos de tipo nominal, ordinal o que representan un orden en forma de ranking) que se utilizan para analizar el comportamiento de algoritmos evolutivos en problemas de optimización. Estos test se describen brevemente a continuación:

- Test de Friedman
  Este test lo que hace es establecer un orden de los resultados obtenidos por el algoritmo para cada longitud de cada problema, es decir hay tantos rankings como problemas estemos analizando.
  Para establecer el orden se asigna a cada resultado un puesto en un ranking r_j para j= 1, 2, ...k (El mejor resultado se le asigna el orden 1 y al peor el orden k, donde k= número de longitudes de cruce estudiadas).
  Este test utiliza la siguiente hipótesis nula: suposición de que el comportamiento de los algoritmos es equivalente y por tanto los rankings R_j también son similares, donde R_j es el ranking medio de los N problemas analizados (nosotros tendremos 6 rankings medios, los correspondientes a las longitudes de cruce), por tanto responde a la siguiente fórmula:

$$R_j = \frac{1}{N}\sum_i r_i^j$$

La distribución del estadístico se da según X_F^2 que se distribuye según una distribución X^2 con k-1 grados de libertad. Su fórmula es:

$$X_F^2 = \frac{12N}{k(k+1)}\left[\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right]$$

Si se rechaza la hipótesis nula, procedemos con el test a posteriori.

- Test de Holm

  El test propuesto por Holm es un tipo de test a posteriori.

  Este test prueba de manera secuencial las hipótesis ordenadas según su significancia, entendiendo por hipótesis a:

$$h_i = \frac{\alpha}{(k-i)}$$

  Donde tenemos que α es un valor fijo [0,1] y k es el valor que se ha obtenido anteriormente mediante el test de Friedman

  Representamos los resultados ordenados como p_1, p_2, … donde p_1 < p_2< …<p_{k-1}. Lo que hacemos en este test es comparar cada p_i con su hipótesis α/(k-i), empezando por el valor más relevante.

  El modo de hacerlo es: Si p_1 < α/(k-1), la hipótesis se rechaza y continuamos comparando, p_2 < α/(k-2)…hasta que una de las hipótesis no pueda ser rechazada. En ese momento todos los demás valores se mantienen como aceptables.

  El estadístico que permite comparar un valor de longitud de cruce con otro es:

$$Z = \frac{(R_i - R_j)}{\sqrt{\frac{k(k+1)}{6N}}}$$

  Donde Z es el valor con el que entramos en la tabla de la normal para obtener la probabilidad. Y esa probabilidad es la que se compara con el valor de α.

Dado que la variable a analizar es la longitud de cruce, se ha elaborado una tabla (tabla 7) con el resultado medio de las 30 ejecuciones de cada enunciado para cada una de las instancias que poseen las iteraciones máximas de 5.000 en pequeños y 30.000 en problemas grandes, es decir para sus 6 variantes según la longitud de cruce. Los valores de esa tabla son los que se utilizan para realizar el estudio estadístico.

	Nº de Partículas - Nº de Iteraciones - Longitud de cruce					
	50-5.000-20%	50-5.000-30%	50-5.000-40%	50-5.000-50%	50-5.000-60%	50-5.000-70%
Br17	39,00	39,00	39,00	39,00	39,00	39,00
ft53	11225,90	10905,57	10193,97	10014,60	9823,30	10019,20
ft70	51326,33	50144,67	49317,10	48531,57	48369,87	48374,13
ftv33	1793,33	1694,53	1589,93	1531,33	1538,40	1592,30
ftv35	2119,13	1884,27	1825,33	1782,00	1779,40	1814,53
ftv38	2240,53	2055,73	1892,93	1855,27	1876,33	1896,03
ftv44	2646,27	2457,03	2288,03	2168,80	2221,53	2214,60
ftv47	2847,17	2730,23	2566,87	2441,57	2460,23	2537,90
ftv55	2996,50	2783,07	2585,47	2531,70	2501,60	2491,50
ftv64	3765,03	3412,23	3259,27	3070,20	3124,03	3041,83
ftv70	4144,67	3899,90	3737,37	3378,93	3371,10	3323,10
ftv170	10230,67	10051,70	9731,20	9233,90	9037,33	8566,17
kro124p	70705,37	66201,30	63062,70	57698,27	55627,57	54647,67
p43	5706,47	5686,03	5671,70	5671,90	5675,73	5662,47
ry48p	20559,97	19066,17	18402,67	17283,30	16804,83	16803,57
Valor Medio	12823,09	12200,76	11744,24	11148,82	10950,02	10868,27
	Nº de Partículas - Nº de Iteraciones - Longitud de cruce					
	50-30.000-20%	50-30.000-30%	50-30.000-40%	50-30.000-50%	50-30.000-60%	50-30.000-70%
rbg323	4030,70	4075,40	3859,30	3668,57	3519,27	3431,13
rbg358	4826,23	4630,07	4356,93	4055,53	3880,60	3823,37
rbg403	5798,87	5536,07	5204,20	5007,87	4831,17	4801,33
rbg443	6440,00	6111,67	5884,07	5652,63	5437,20	5371,60
Valor Medio	5273,95	5088,30	4826,13	4596,15	4417,06	4356,86

Tabla 7. Resultados medios utilizados para el estadístico. Elaboración propia.

4.4.1 Resultados

Tras realizar los estudios estadísticos obtenemos los siguientes rankings (tabla 8) medios para los 6 algoritmos que comparamos:

Algorithm	Ranking
20	5.8157894736842115
30	4.973684210526316
40	3.7631578947368407
50	2.447368421052632
60	2.1315789473684204
70	1.868421052631578

Tabla 8. Ranking medio para los 6 algoritmos. Elaboración propia.

Como se puede ver el % que mejor resultado da es el del 70%. El test de Friedman obtiene un P-valor = 5^{-11}, por lo que se rechaza la hipótesis nula.

Sabiendo ya que esa hipótesis es rechazada procedemos a realizar el test de Holm.

En el test de Holm procedemos comparando la longitud de cruce 70%, ya que es la más relevante, con todas las demás.

Para α = 0,05 (nivel de precisión del 95%) obtenemos del test de Holm los resultados de la tabla 9. La hipótesis se rechaza si tiene un p-valor menor o igual que 0,025, por lo que como podemos ver en la columna p (es el p-valor) de la tabla, los porcentajes 20%, 30% y 40% rechazan la hipótesis de igualdad con el algoritmo de longitud de cruce 70%.

i	algorithm	$z = (R_0 - R_i)/SE$	p	Holm/Hochberg/Hommel
5	20	6.503324771430904	7.856392566427658E-11	0.01
4	30	5.11594882019231	3.121677039209956E-7	0.0125
3	40	3.1215958902868315	0.0017987365863089445	0.016666666666666666
2	50	0.9538209664765339	0.3401742745401112	0.025
1	60	0.4335549847620603	0.6646116295410062	0.05

Tabla 9. Test de Holm para α = 0,05. Elaboración propia.

En la tabla 10 mostramos el test de Holm pero con α = 0,1 (nivel de precisión del 90%). Para ese valor de α, el test de Holm determina que la hipótesis se rechaza si el p-valor es menor o igual que 0,05.

Por tanto para este nivel de α podemos ver de nuevo, en la columna p, que las hipótesis que se rechazan son la del 20%, 30% y 40%, mientras que para los algoritmos con longitud de cruce de 50% y 60% no se puede rechazar la hipótesis de igualdad respecto al 70%.

i	algorithm	$z = (R_0 - R_i)/SE$	p	Holm/Hochberg/Hommel
5	20	6.503324771430904	7.856392566427658E-11	0.02
4	30	5.11594882019231	3.121677039209956E-7	0.025
3	40	3.1215958902868315	0.0017987365863089445	0.03333333333333333
2	50	0.9538209664765339	0.3401742745401112	0.05
1	60	0.4335549847620603	0.6646116295410062	0.1

Tabla 10. Test de Holm para α = 0,1. Elaboración propia.

Podemos concluir este capítulo diciendo que el método que hemos implementado se comporta mejor cuantas más iteraciones hagan en cada ejecución, y en cuando a la variable longitud de cruce debemos indicar, que aunque la longitud de cruce que mejores resultados obtiene es la del 70%, no podemos decir, estadísticamente halando, que la longitud de cruce de 70% sea significativamente mejor que la del 50% y 60%. Lo que si podemos asegurar, es que el 50%, 60% y 70% son mejores que el 20%, 30% y 40%.

Por tanto de ahora en adelante, si se va a ejecutar este algoritmo, recomendamos que se establezcan el mayor número de iteraciones posibles (siempre que no sobrepase la capacidad operativa del ordenador) y que la longitud de cruce sea igual o mayor que el 50% de la dimensión del problema.

5. CONCLUSIÓN E INVESTIGACIÓN DE LÍNEAS FUTURAS

Analizando los resultados experimentales que se han obtenido, podemos concluir que el algoritmo desarrollado para implementar el método de Optimización por Enjambre de Partículas al Problema del Agente Viajero parece funcionar correctamente, obteniendo valores muy cercanos al óptimo en problemas donde el número de ciudades a recorrer es pequeño, incluso alcanza el mejor resultado conocido hasta el momento para el menor de los problemas de la librería, el br17. Sin embargo, para problemas de grandes dimensiones el resultado que se obtiene dista bastante del valor óptimo conocido.

En cuanto al valor de las variables utilizadas a la hora de ejecutar el algoritmo, podemos decir que el algoritmo se comporta mejor ante iteraciones máximas y longitudes de cruce elevadas. No obstante, queda pendiente para investigaciones futuras cuál sería el valor óptimo para estas variables. También sería recomendable indagar en el tamaño de la población que optimiza el funcionamiento del algoritmo.

6. BIBLIOGRAFÍA

- Applegate, D., Bixby, R., Chavátal, V. y Cook, W. (2011). The Traveling Salesman Problem: A Computational Study. Pág. 1-3.

- Clerc, M. (2010). Particle Swarm Optimization. Pág.17-19.

- Eberhart, R. y Kennedy, J. (1995). A New Optimizer Using Partcle Swarm Theory. Sixth International Symposium on Micro Machine and Human Science. 0-7803-2676-8/95. Pág. 39-43.

- Fan, H. (2010). Discrete Particle Swarm Optimization for TSP based on Neighborhood. Journal of Computational Information Systems 6:10. Pág. 3407-3413.

- García, S., Molina, D., Lozano, M. and Herrera, F. (2009). A study on the use of non-parametric test for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization. Journal of Heuristics, 15. Pág. 639-641.

- Guerequeta, R. y Vallecillo, A. (1998). Técnicas de Diseño de Algoritmos, Capítulo 7: Ramificación y poda. Pág. 255-257.

- Gutin, G. y Punnen, A. (2007) The Traveling Salesman Problem and Its Variations. Pág. 9-15.

- Hoffman, K. (2000). Combinatorial optimization: Current successes and directions for the future. Journal of Computational and Applied Mathematics. Nº 124. Pág. 344-348.

- Kennedy, J. y Eberhart, R. (1995). Particle Swarm Optimization. 0-7803-2768-3/95. Pág. 1942-1948.

- Kennedy, J. y Eberhart, R. (1997). A Discrete Binary Version of the Particle Swarm Algorithm. 0-7803-4053-1/97. Pág. 4104-4108.

- Larré, O. (2012) El problema del vendedor viajero en grafos cúbicos. Tesis, Universidad de Chile. Pág. 1-6.

- Lin, S. (1965). Computer Solutions of the Traveling Salesman Problem. The Bell System Technical Journal. Volume 44, Issue 10. Pág.2245-2269.

- Luna, F. (2008). Metaheurísticas avanzadas para problemas reales en redes de telecomunicaciones. Tesis doctoral, Universidad de Málaga. Pág. 19-28.

- Luque, G. (2006). Resolución de Problemas Combinatorios con Aplicación Real en Sistemas Distribuidos. Tesis doctoral, Universidad de Málaga.

- Martí, R. (2003). Procedimientos Metaheurísticos en Optimización Combinatoria. Universidad de Valencia, Facultad de matemáticas.

- Ross, P. (2005). Search Methodologies, Chapter 17: Hyper heuristics. Pág. 529-531.

Webs Consultadas:

[1] http://www.dc.uba.ar/materias/aed3/2014/2c/teorica/handout_compl.pdf. Último acceso: Junio 2015

[2] https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_computaci%C3%B3n#Teor.C3.ADa_de_la_computabilidad. Último acceso: Junio 2015.

[3] http://ocw.uc3m.es/ingenieria-informatica/teoria-de-automatas-y-lenguajes-formales/material-de-clase-1/tema-8-complejidad-computacional. Último acceso: Junio 2015.

[4] https://es.wikipedia.org/wiki/NP-hard. Último acceso: Junio 2015.

[5] https://upload.wikimedia.org/wikipedia/commons/6/66/P_np_np-completo_np-hard.svg. Último acceso: Junio 2015.

[6] http://www.dii.uchile.cl/~daespino/PApers/TSP_and_IP_chile_050820.pdf. Último acceso: Julio 2015.

[7] http://es.wikipedia.org/wiki/Problema_del_viajante. Último acceso: Junio 2015.

[8] http://www.csd.uoc.gr/~hy583/papers/ch11.pdf. Último acceso: Junio 2015.

[9] http://www.tebadm.ulpgc.es/almacen/seminarios/MH%20Las%20Palmas%202.pdf. Último acceso: Junio 2015.

[10] http://es.wikipedia.org/wiki/Optimizaci%C3%B3n_por_enjambre_de_part%C3%ADculas. Último acceso: Junio 2015.

[11] www.codeblocks.org. Último acceso: Mayo 2015.

[12] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Último acceso: Julio 2015.