UNIVERSIDAD DE CÓRDOBA
Escuela Politécnica Superior
Ingeniería Informática
Metaheurísticas
Francisco José Rodríguez Ramírez
Víctor Monserrat Villatoro

2. Objetivo.

El objetivo de esta práctica es acercar al alumnado a las primeras metaheurísticas completas basadas en trayectorias, enfriamiento simulado, búsqueda tabú, búsqueda local iterativa, métodos GRASP, greedy iterativo... Para ello, se presentará un guión de actividades a realizar para generar y analizar diversos métodos en los problemas de optimización comentados en la práctica anterior, el problema de la mochila (KP) y el del viajante de comercio (TSP).

3. El problema de la mochila.

3.1. Descripción.

El problema de la mochila consiste en introducir en una mochila con una capacidad C una serie de objetos con beneficio p_i y peso w_i, el peso de cada objeto es el espacio que ocupa en la mochila y el beneficio el valor que tiene cada objeto individualmente. El problema consiste en maximizar la suma de beneficios de combinaciones de objetos sin superar el peso máximo que soporta la mochila.

3.2. Análisis.

Para resolver problemas de la mochila leemos ficheros en los que se exponen varios problemas de este tipo y almacenaremos cada una de sus configuraciones. Cada problema consta de una mochila, de la cual se nos especifica su capacidad, y de una lista de objetos de los que se nos proporciona su beneficio y peso individual.

Nuestro trabajo consistirá en, una vez cargadas las instancias de este tipo de problema, para el enfriamiento simulado generar una solución aleatoria de este e ir generando soluciones vecinas a ésta, de forma que si mejora el beneficio la guardaremos; y si no, evaluaremos en función de una temperatura, que iremos enfriando, mediante una probabilidad si la guardaremos o no.

Para el método GRASP generamos una solución de este basada en una heurística determinada, en nuestro caso esta heurística será introducir de manera ordenada los objetos cuyo ratio (beneficio/peso) sea mejor. Más tarde, se optimizarán a través de una búsqueda local, teniendo en cuenta las restricciones propias del problema y se evaluarán.

Llenaremos la mochila con los elementos de mayor ratio hasta que ningún otro objeto de los que nos quedan por introducir quepa en nuestra mochila, esta será nuestra condición de parada. De esta forma, se generará una solución que llena al máximo la mochila de nuestro problema, a través de esta heurística.

Para optimizar estas soluciones, consideramos la búsqueda local. Sacaremos un elemento de la mochila e introduciremos otro de manera aleatoria, sin que el peso del nuevo elemento sobrepase la capacidad de nuestra mochila. De esta manera, obtendremos una nueva solución vecina a la generada en primera instancia de forma totalmente aleatoria.

Para la optimización, tendremos que generar soluciones vecinas y evaluarlas. En nuestro caso, hemos elegido dos estrategias de optimización, primera y mejor mejora. La primera estrategia consiste en generar soluciones vecinas hasta encontrar una vecina que obtenga mayor beneficio. La segunda estrategia consiste en generar un número de soluciones vecinas y almacenar la mejor de todas ellas.

Finalmente, se generarán cien mil soluciones del problema, se optimizarán a través de la estrategia de primera mejora y evaluaremos todas para poder elegir la que nos interese. En este caso, nuestro interés reside en maximizar el beneficio de la mochila, con lo que iremos almacenando la solución aleatoria que obtenga mayor beneficio.

3.3. Desarrollo.

Para la resolución de este tipo de problemas se han implementado una serie de clases para representar la información de las instancias de estos.

La clase KP::Instance se ha implementado como un vector de KP::Problem, que son problemas de este tipo. Esta clase se instancia a través de un constructor que recibe como único parámetro el nombre del fichero del cual queremos leer un conjunto de problemas de este tipo.

La clase anterior instancia todos los problemas del fichero a través de KP::Problem que se ha implementado como una representación de un problema en sí a través de una mochila KP::Knacksack y un vector de objetos KP::Object.

La mochila KP::Knacksack tiene una capacidad y los objetos KP::Object tienen un beneficio y un peso.

Con estas clases y a través del constructor de KP::Instance se instancian todos los problemas de un fichero en nuestro programa.

Para generar las soluciones se ha creado una clase KP::Solution cuyo constructor recibe un problema KP::Problem y genera una solución aleatoria para este, metiendo elementos del problema aleatoriamente hasta que la mochila no soporta más y así no sobrepasar la capacidad de esta, esta será nuestra condición de parada.

De esta forma, y con esta clase, seremos capaces de generar el número de soluciones que deseemos a un determinado problema de este tipo.

Para la elección de nuestra estrategia de optimización local hemos creado dos nuevas clases, KP::FirstNeighExplorator para la estrategia de primera mejora, y KP::BestNeighExplorator para la estrategia de mejor mejora. Estas dos clases tienen un límite que indica a la búsqueda local el número de soluciones vecinas a generar de máximo hasta encontrar una mejor para la estrategia de primera mejora o como condición de parada para la estrategia de mejor mejora.

KP::FirstNeighExplorator y KP::BestNeighExplorator crean soluciones como se ha comentado en el análisis del problema.

La clase KP::LocalSearch recibe una estrategia de búsqueda local y tiene un método search(), que recibirá una solución aleatoria y aplicará la optimización correspondiente a la estrategia correspondiente. Este método devolverá una solución vecina de mejor beneficio o la propia solución si no ha sido capaz de encontrar una mejor sin sobrepasar el límite de la estrategia.

La clase KP::SimulatedAnnealing recibe una solución, la diferencia de beneficio que queremos permitir, la probabilidad inicial con la que queremos permitir esta diferencia, y el número de iteraciones de enfriamiento. Se calcula la temperatura inicial a través de la ley de la termodinámica, $to = \frac{-\Delta E}{log(P(\Delta E)}$. Se generar soluciones vecinas a la dada, se acepta o no según la probabilidad generada a través de la ley; que es directamente proporcional a la temperatura, que se irá enfriando en cada iteración de la forma $t = \frac{to}{1+log(1+i)}$, siendo *i* el número de iteración.

La clase KP::GRASP recibe un problema y el número de iteraciones. Se calculan los ratios de todos los elementos del problema, se introducen de forma ordenada hasta llenar la mochila y se realiza una búsqueda local de esta solución según el número de iteraciones.

Finalmente, a través de un programa principal que hace uso de las clases ya descritas se generan soluciones para un problema determinado que se introduce a través de un fichero y el número de problema que se quiere solucionar. Este programa genera las soluciones deseadas para los dos métodos, se optimizan y se almacena siempre la de mayor beneficio total, con lo que al final de su ejecución se nos informará de esta solución óptima encontrada, pudiendo acceder a su configuración y poder ver qué objetos componen esta solución.

3.4. Valor de aptitud para cada iteración (enfriamiento simulado).

El problema que se representa es la instancia número 50 del fichero knapPI_1_200_10000.csv, la capacidad de la mochila es de 471759 unidades y se consta de 200 objetos.

A continuación se representa el valor de aptitud de cada solución actual, la temperatura actual y el valor de aptitud de la mejor solución.

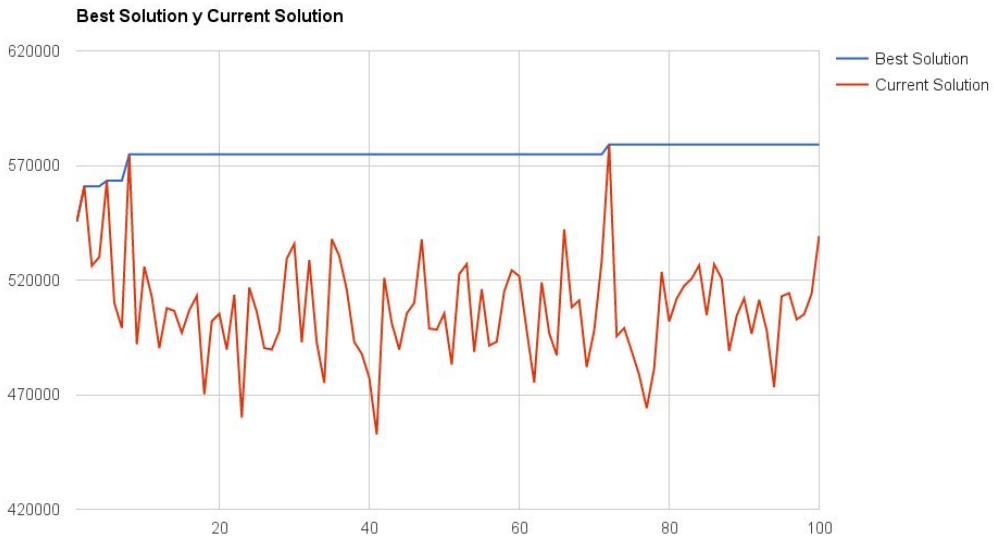
	Solución actual	Temperatura actual	Mejor solución
Iteración 10000	**525851**	**5728.23**	**574828**
Iteración 20000	**505301**	**5364.08**	**574828**
Iteración 30000	**535768**	**5171.76**	**574828**
Iteración 40000	**477340**	**5043.46**	**574828**
Iteración 50000	**505448**	**4948.25**	**574828**
Iteración 60000	**521709**	**4873.08**	**574828**
Iteración 70000	**497976**	**4811.28**	**574828**
Iteración 80000	**501909**	**4759.01**	**579146**
Iteración 90000	**511944**	**4713.83**	**579146**
Iteración 100000	**539249**	**4674.14**	**579146**

La maximización del beneficio se ha conseguido en la iteración 71308, el número de objetos introducidos en la mochila es de 106 y el peso total de estos es 464671.

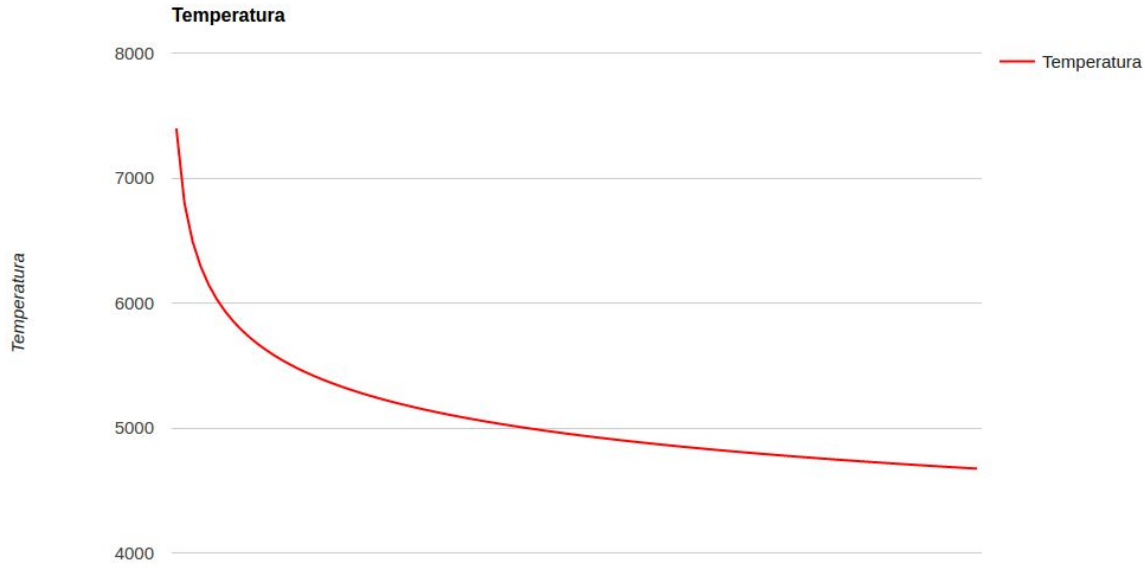
Por norma general, para un número de iteraciones parecido, se suelen obtener soluciones óptimas parecidas, aunque en algunos casos se encuentran diferencias significativas.

Podemos observar que conforme se avanza el número de iteración se reduce la probabilidad de conseguir una solución mejor a las ya generadas aleatoriamente. Es decir, al comienzo de la ejecución conseguimos mejorar rápidamente la solución óptima. Sin embargo, al ir avanzando tendremos que usar mayor número de iteraciones para conseguir mejorar esta solución óptima actual.

A continuación, se representa de forma gráfica los valores de aptitud de algunas de las soluciones generadas por enfriamiento simulado y cómo se enfría la temperatura.



Ver de forma interactiva



Ver de forma interactiva

3.5. Valor de aptitud para cada iteración (GRASP).

El problema que se representa es la instancia número 56 del fichero knapPI_1_200_10000.csv, la capacidad de la mochila es de 547906 unidades y se consta de 200 objetos.

A continuación se representa el valor de aptitud de la solución actual y de la mejor solución.

	Solución actual	Mejor solución
Solución inicial	**8836280**	**8836280**
Iteración 1	**8837620**	**8837620**
Iteración 3	**8838070**	**8838070**
Iteración 4	**8836750**	**8838070**
Iteración 5	**8838250**	**8838250**
Iteración 6	**8837780**	**8838250**
Iteración 7	**8838550**	**8838550**
Iteración 10	**8836750**	**8838550**
Iteración 14	**8838890**	**8838890**
Iteración 20	**8838550**	**8838890**
Iteración 28	**8842600**	**8842600**

Como podemos observar en la tabla, la maximización del beneficio se ha conseguido en la iteración 28 y ya no mejora más en todas las 100000 iteraciones.

Por norma general, las mejoras del beneficio en la optimización local, se consiguen en las primeras iteraciones y más tarde es muy poco probable encontrar otra mejor.

Podemos observar que conforme se avanza el número de iteración se reduce la probabilidad de conseguir una solución mejor a las ya generadas ya. Es decir, al comienzo de la ejecución conseguimos mejorar rápidamente la solución óptima. Sin embargo, al ir avanzando tendremos que usar mayor número de iteraciones para conseguir mejorar esta solución óptima actual.

3.6. Configuración de la mejor solución (enfriamiento simulado).

Como ya se ha comentado la mejor solución se consigue en la iteración 71308, pero ahora hablaremos de la configuración de esta.

Para esta solución se utilizan 106 de los 200 objetos disponibles, ocupando así con un peso total de 464671 unidades de la capacidad total de 471759 unidades que tiene la mochila en esta instancia del problema.

3.7. Configuración de la mejor solución (GRASP).

Como ya se ha comentado la mejor solución se consigue en la iteración 28, pero ahora hablaremos de la configuración de esta.

Para esta solución se utilizan 139 de los 200 objetos disponibles, ocupando así con un peso total de 547802 unidades de la capacidad total de 547906 unidades que tiene la mochila en esta instancia del problema.

4. El problema del viajante de comercio.

4.1. Descripción.

El problema del viajante de comercio consiste en recorrer un grafo totalmente conexo de *N* nodos de forma cíclica, de forma que se recorran todos los nodos volviendo al punto de partida. Los caminos entre nodos tienen una distancia d_{ij}. El problema consiste en minimizar la distancia total para recorrer todos los nodos del grafo, seleccionando el camino de menor coste.

4.2. Análisis.

Para resolver problemas del viajante de comercio leemos ficheros en los que se exponen varios problemas de este tipo y almacenaremos cada una de sus configuraciones. Cada problema consta de una serie de nodos, con unas coordenadas y una distancia entre cada uno de ellos.

Nuestro trabajo consistirá en, una vez cargadas las instancias de este tipo de problema, para el enfriamiento simulado generar una solución aleatoria de este e ir generando soluciones vecinas a ésta, de forma que si mejora el beneficio la guardaremos; y si no, evaluaremos en función de una temperatura, que iremos enfriando, mediante una probabilidad si la guardaremos o no.

Para el método GRASP generamos una solución de este basada en una heurística determinada, en nuestro caso esta heurística será construir el camino viajando siempre al nodo más cercano. Más tarde, se optimizarán a través de una búsqueda local, teniendo en cuenta las restricciones propias del problema y se evaluarán.

Recorremos el grafo sin repetir ningún nodo, viajando al nodo más cercano al actual hasta que ningún otro nodo quede sin visitar, esta será nuestra condición de parada. De esta forma, se generará una solución que recorre todos los nodos de nuestro problema, a través de esta heurística.

Para optimizar estas soluciones, consideramos la búsqueda local. Elegiremos dos nodos al azar del vector trayectoria y daremos la vuelta a todos los comprendidos, de forma que solo cambian dos distancias. De esta manera, obtendremos una nueva solución vecina a la generada en primera instancia de forma totalmente aleatoria

Para la optimización, tendremos que generar soluciones vecinas y evaluarlas. En nuestro caso, hemos elegido dos estrategias de optimización, primera y mejor mejora. La primera estrategia consiste en generar soluciones vecinas hasta encontrar una vecina que obtenga mayor beneficio. La segunda estrategia consiste en generar un número de soluciones vecinas y almacenar la mejor de todas ellas.

Finalmente, se generarán cien mil soluciones del problema, se optimizarán a través de la estrategia de primera mejora y evaluaremos todas para poder elegir la que nos interese. En este caso, nuestro interés reside en minimizar la distancia total que habrá que recorrer para pasar por todos los nodos del grafo, con lo que iremos almacenando la solución aleatoria que obtenga una menor distancia total.

4.3. Desarrollo.

Para la resolución de este tipo de problemas se han implementado una serie de clases para representar la información de las instancias de estos.

La clase TSP::Instance se ha implementado como un vector de TSP::Problem, que son problemas de este tipo. Esta clase se instancia a través de un constructor que recibe como único parámetro el nombre del fichero del cual queremos leer un problema de este tipo.

La clase anterior instancia el problema del fichero a través de TSP::Problem que se ha implementado como una representación de un problema en sí a través de un vector de nodos TSP::Node.

Los nodos TSP::Node tienen unas coordenadas *x* e *y* que representan su situación.

Con estas clases y a través del constructor de TSP::Instance se instancian todos los problemas de un fichero en nuestro programa.

Para generar las soluciones se ha creado una clase TSP::Solution cuyo constructor recibe un problema TSP::Problem y genera una solución aleatoria para este, recorriendo el grafo aleatoriamente hasta que se alcanza el nodo inicial pasando sólo una vez por cada nodo intermedio.

De esta forma, y con esta clase, seremos capaces de generar el número de soluciones que deseemos a un determinado problema de este tipo.

Para la elección de nuestra estrategia de optimización local hemos creado dos nuevas clases, TSP::FirstNeighExplorator para la estrategia de primera mejora, y TSP::BestNeighExplorator para la estrategia de mejor mejora. Estas dos clases tienen un límite que indica a la búsqueda local el número de soluciones vecinas a generar de máximo hasta encontrar una mejor para la estrategia de primera mejora o como condición de parada para la estrategia de mejor mejora.

TSP::FirstNeighExplorator y TSP::BestNeighExplorator crean soluciones como se ha comentado en el análisis del problema.

La clase TSP::LocalSearch recibe una estrategia de búsqueda local y tiene un método search(), que recibirá una solución aleatoria y aplicará la optimización correspondiente a la estrategia correspondiente. Este método devolverá una solución vecina de menor distancia o la propia solución si no ha sido capaz de encontrar una mejor sin sobrepasar el límite de la estrategia.

La clase KP::SimulatedAnnealing recibe una solución, la diferencia de distancia que queremos permitir, la probabilidad inicial con la que queremos permitir esta diferencia, y el número de iteraciones de enfriamiento. Se calcula la temperatura inicial a través de la ley de la termodinámica, $to = \frac{-\Delta E}{log(P(\Delta E)}$. Se generar soluciones vecinas a la dada, se acepta o no según la probabilidad generada a través de la ley; que es directamente proporcional a la temperatura, que se irá enfriando en cada iteración de la forma $t = \frac{to}{1+log(1+i)}$, siendo *i* el número de iteración.

La clase KP::GRASP recibe un problema y el número de iteraciones. Se calculan las distancias de todos los nodos del problema al actual, se recorren de forma ordenada los nodos hasta recorrer el grafo por completo y se realiza una búsqueda local de esta solución según el número de iteraciones.

Finalmente, a través de un programa principal que hace uso de las clases ya descritas se generan soluciones para un problema determinado que se introduce a través de un fichero y el número de problema que se quiere solucionar. Este programa genera las soluciones deseadas para los dos métodos, se optimizan y se almacena siempre la de menor distancia total, con lo que al final de su ejecución se nos informará de esta solución óptima encontrada, pudiendo acceder a su configuración y poder ver qué camino compone esta solución.

4.4. Valor de aptitud para cada iteración (enfriamiento simulado).

El problema que se representa es la instancia del fichero berlin52.tsp, la dimensión del grafo es de 52 nodos.

A continuación se representa el valor de aptitud de cada solución inicial, sus soluciones optimizadas y de la mejor solución con una estrategia de primera mejora.

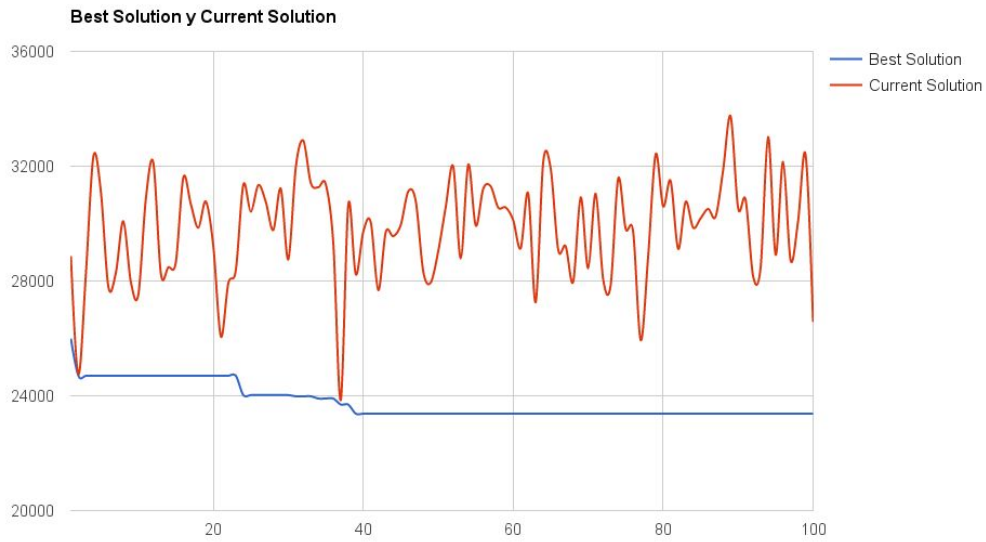
	Solución inicial	Solución optimizada	Mejor solución
Iteración 10000	**27526.7**	**5728.23**	**24698.2**
Iteración 20000	**29211.6**	**5364.08**	**24698.2**
Iteración 30000	**28750.5**	**5171.76**	**24023.8**
Iteración 40000	**29716.9**	**5043.46**	**23376.1**
Iteración 50000	**29035.2**	**4948.25**	**23376.1**
Iteración 60000	**30118**	**4873.08**	**23376.1**
Iteración 70000	**28444.2**	**4811.28**	**23376.1**
Iteración 80000	**30602.7**	**4759.01**	**23376.1**
Iteración 90000	**30546**	**4713.83**	**23376.1**
Iteración 100000	**26577.5**	**4674.14**	**23376.1**

Como podemos observar en la tabla, la minimización de la distancia total se ha conseguido en la iteración 32854.

Por norma general, para un número de iteraciones parecido, se suelen obtener soluciones óptimas parecidas, aunque en algunos casos se encuentran diferencias significativas.

Podemos observar que conforme se avanza el número de iteración se reduce la probabilidad de conseguir una solución mejor a las ya generadas aleatoriamente. Es decir, al comienzo de la ejecución conseguimos mejorar rápidamente la solución óptima. Sin embargo, al ir avanzando tendremos que usar mayor número de iteraciones para conseguir mejorar esta solución óptima actual.

A continuación, se representa de forma gráfica los valores de aptitud de algunas de las soluciones generadas por enfriamiento simulado y cómo se enfría la temperatura.



Ver de forma interactiva

Ver de forma interactiva

4.5. Valor de aptitud para cada iteración (GRASP).

El problema que se representa es la instancia del fichero berlin52.tsp, la dimensión del grafo es de 52 nodos.

A continuación se representa el valor de aptitud de cada solución actual y de la mejor solución.

	Solución actual	Mejor solución
Solución inicial	**9938.73**	**9938.73**
Iteración 1	**10228**	**9938.73**
Iteración 2	**9808.81**	**9808.81**
Iteración 3	**9888.35**	**9808.81**
Iteración 4	**9307.66**	**9307.66**
Iteración 5	**9585.82**	**9307.66**
Iteración 6	**10228**	**9307.66**
Iteración 7	**9417.74**	**9307.66**
Iteración 8	**9065.98**	**9065.98**
Iteración 9	**9065.98**	**9065.98**

Como podemos observar en la tabla, la minimización de la distancia total se ha conseguido en la iteración 8 y ya no mejora más en todas las 100000 iteraciones.

Por norma general, para un número de iteraciones parecido, se suelen obtener soluciones óptimas parecidas, aunque en algunos casos se encuentran diferencias significativas.

Podemos observar que conforme se avanza el número de iteración se reduce la probabilidad de conseguir una solución mejor a las ya generadas aleatoriamente. Es decir, al comienzo de la ejecución conseguimos mejorar rápidamente la solución óptima. Sin embargo, al ir avanzando tendremos que usar mayor número de iteraciones para conseguir mejorar esta solución óptima actual.

4.6. Configuración de la mejor solución (primera mejora).

Como ya se ha comentado la mejor solución se consigue en la iteración 32854, pero ahora hablaremos de la configuración de esta.

Para esta solución se recorre una distancia total de 23376.1 unidades a través de los 52 nodos del problema.

4.7. Configuración de la mejor solución (mejor mejora).

Como ya se ha comentado la mejor solución se consigue en la iteración 700, pero ahora hablaremos de la configuración de esta.

Para esta solución se recorre una distancia total de 25798.3 unidades a través de los 52 nodos del problema.