|  |
| --- |
| Universidad de Granada |
| Viajante de Comercio (B&B) |
| Algorítmica |
| **Aarón Bueno Rodríguez**  **Bryan Moreno Picamán**  **Miguel Ángel Rodríguez Serrano** |
|  |
|  |

|  |
| --- |
| 2º Grado Ingeniería Informática |

**EXPLICACION DEL PROBLEMA**

El problema del viajante de comercio ya se ha comentado y utilizado en la práctica sobre algoritmos voraces, donde se estudiaron métodos de este tipo para encontrar soluciones razonables (no optimas necesariamente) a este problema. Si se desea encontrar una solución óptima es necesario utilizar métodos más potentes (y costosos), como la vuelta atrás y la radicación y poda, que exploren el espacio de posibles soluciones de forma m as exhaustiva.

Así, un algoritmo de vuelta atrás comenzar a en la ciudad 1 (podemos suponer sin pérdida de generalidad, al tratarse de encontrar un tour, que la ciudad de inicio y n es esa ciudad) e intentar a incluir como parte del tour la siguiente ciudad a un no visitada, continuando de este modo hasta completar un tour. Para agilizar la búsqueda de la solución se deben considerar como ciudades válidas para una posición (ciudad actual) solo aquellas que satisfagan las restricciones del problema (en este caso ciudades que a un no hayan sido visitadas). Cuando para un nivel no queden m as ciudades validas, el algoritmo hace una vuelta atrás proponiendo una nueva ciudad válida para el nivel anterior.

Para emplear un algoritmo de radicación y poda es necesario utilizar una cota inferior:

Un valor menor o igual que el verdadero coste de la mejor solución (la de menor coste) que se puede obtener a partir de la solución parcial en la que nos encontremos.

Una posible alternativa ser a la siguiente: como sabemos cuáles son las ciudades que faltan por visitar, una estimación optimista del costo que a un nos queda ser a, para cada ciudad, el coste del mejor (menor) arco saliente de esa ciudad. La suma de los costes de esos arcos, más el coste del camino ya acumulado, es una cota inferior en el sentido antes descrito.

Para realizar la poda, guardamos en todo momento en una variable C el costo de la mejor solución obtenida hasta ahora (que se utiliza como cota superior global: la solución óptima debe tener un coste menor o igual a esa). Esa variable puede inicializarse con el costo de la solución obtenida utilizando un algoritmo voraz (como los utilizados en la practica 2). Si para una solución parcial, su cota inferior es mayor que C entonces se puede realizar la poda.

Como criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda (la solución parcial que tratamos de expandir), se emplear a el criterio LC o \m as prometedor". En este caso consideraremos como nodo m as prometedor aquel que presente el menor valor de cota inferior. Para ello se debe de utilizar una cola con prioridad que almacene los nodos ya generados (nodos vivos).

Además de devolver el costo de la solución encontrada (y en su caso el tour correspondiente), se deben de obtener también resultados relativos a complejidad: número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos, número de veces que se realiza la poda y el tiempo empleado en resolver el problema.

Las pruebas del algoritmo pueden realizarse con los mismos datos empleados en la practica 2 (teniendo en cuenta que el tamaño de problemas que se pueden abordar con estas técnicas es mucho m as reducido que con los métodos voraces). La visualización de las soluciones también puede hacerse de la misma forma que en la practica 2 (usando grupo).

|  |
| --- |
| Viajante de Comercio con Tecnica Branch & Bound |

Teniendo en mente el problema propuesto se ha optado por abordar el problema intentando dividir las funciones más difíciles y/o costosas del proceso.

Por un lado tenemos “creaMatriz.cpp”, código encargado de leer los datos de los archivos y proporcionar las estructuras que posteriormente se usaran para realizar el problema, las cuales son una matriz de distancias entre ciudades (como es normal con la diagonal de la misma de valor 0), y dos vectores con las posiciones X e Y de cada una de las ciudades, el cual se podrá utiliza posteriormente para sacar los recorridos con objetivo de representarlos gráficamente con GNUPlot.

Por otro lado tenemos “viajantebb.cpp”, que haciendo uso del comentado anteriormente realiza el resto de funciones del algoritmo.

Dentro de este archivo hemos optado por crear una clase “Nodo” que representará un objeto o nodo, en el que se tendrá una cota y una lista de ciudades, además de esto se tiene una función less (menor que), que será utilizada para la ordenación de una cola de objetos de este tipo, haciendo más sencilla la tarea de añadir y quitar objetos durante la ejecución.

## Explicación del algoritmo.

* En primer lugar cuando llamamos a nuestro algoritmo, este realiza la llamada a matriz.cpp para crear como ya se comentó anteriormente la matriz de distancias y los vectores de posiciones de las ciudades.
* Una vez disponemos de esta matriz, podemos realizar la llamada a una función Greedy que nos servirá para obtener la variable “C”, siendo esta el costo mejor hasta el momento y también un recorrido inicial (que se guardará en caso de que por casualidad este fuera el mejor)
* Ya con estos dos pasos realizados, se procede a la ejecución del algoritmo Branch & Bound en si mismo:
  + Tomando la ciudad 1 como la inicial expandimos los nodos hijos que tiene y calculamos sus cotas, añadiéndolos posteriormente a la lista con prioridad de nodos.
  + Cogemos el mejor de la lista y miramos si su “cota” (que es una estimación como podemos recordar) es mejor que la que tenemos actualmente, si es así se procede a quitar este nodo de la lista y expandir sus hijos para añadirlos a la misma.
    - En caso de que no sea mejor, como la lista está ordenada el algoritmo podaría el resto de nodos uno detrás de otro finalizando el algoritmo al quedar la lista vacía
  + Ahora se vuelve a realizar el proceso, se coge el mejor nodo de la lista, se mira si es mejor que el que tenemos actualmente y se expande, esto no cambia hasta llegar a un nodo hoja, en el que comprobaremos si se trata de una solución, si es así y los costes son inferiores a los ya almacenados, se sustituye la solución que teníamos (variable “C” y lista solución) con los nuevos valores, tras lo cual se vuelve a realizar el proceso.

|  |
| --- |
| Viajante de Comercio con Tecnica Backtracking |

Como actividad opcional tenemos realizar el mismo problema usando la técnica de backtracking, aprovechando para ello el cálculo de cotas usado anteriormente.

Como en el caso anterior, por un lado tenemos “creaMatriz.cpp”, código encargado de leer los datos de los archivos y proporcionar las estructuras que posteriormente se usaran para realizar el problema, las cuales son una matriz de distancias entre ciudades (como es normal con la diagonal de la misma de valor 0), y dos vectores con las posiciones X e Y de cada una de las ciudades, el cual se podrá utiliza posteriormente para sacar los recorridos con objetivo de representarlos gráficamente con GNUPlot.

Por otro lado tenemos “viajanteBacktracking.cpp”, que haciendo uso del comentado anteriormente realiza el resto de funciones del algoritmo. En este caso el uso de la clase “Nodo” no es necesario, ya que como conocemos esta técnica utiliza una exploración en profundidad de todos los nodos, en este caso reducido por el uso de cotas.

## Explicación del algoritmo.

* En primer lugar cuando llamamos a nuestro algoritmo, este realiza la llamada a matriz.cpp para crear como ya se comentó anteriormente la matriz de distancias y los vectores de posiciones de las ciudades.
* Una vez disponemos de esta matriz, podemos realizar la llamada a una función Greedy que nos servirá para obtener la variable “C”, siendo esta el costo mejor hasta el momento y también un recorrido inicial (que se guardará en caso de que por casualidad este fuera el mejor)
* Ya con estos dos pasos realizados, se procede a la ejecución del algoritmo Backtracking en sí mismo:
  + Primero tenemos una función que usando la cota inicial o la mejor cota hasta el momento si ya se ha llegado a una solución, nos dice si debemos continuar o no por un camino (poda).
  + En caso afirmativo, entramos a un bucle, que va añadiendo una ciudad en cada iteración y llamando a la función backtracking, tras la cual elimina la ciudad correspondiente.
  + Por otra parte, si se llega a una profundidad de nodo hoja, el algoritmo intenta ver si se trata de una solución (evitando el proceso de llamada recursiva que pierde sentido en este caso). En caso de tratarse de una solución, mejor que la que se tiene actualmente, se almacenan sus nuevos valores y termina la recursión.
  + Al terminar todas las llamadas, en las variables pasadas inicialmente a la función, tendremos el orden de las ciudades así como el coste de la mejor de todas ellas.

|  |
| --- |
| Estudio Empírico de Eficiencia |

Para comprobar la eficiencia de Branch and Bound usando como función de acotación inicial un algoritmo greedy, se ha llevado a cabo un estudio empírico comprobando la ejecución con distintos tamaños de ciudades. De la misma manera, usando el algoritmo opcional Backtracking, se ha realizado una comparativa entre ambos, usando unos ejemplos ficticios de tamaño desde 2 hasta 16

A continuación se puede ver tanto en una tabla como de forma gráfica los tiempos de ejecución.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Branch and Bound |  |  |  |  |  |  |  |  |  |
| Backtracking con Poda |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 11 | 12 | 13 | 14 | 15 | 16 |
| Branch and Bound |  |  |  |  |  |  |
| Backtracking con Poda |  |  |  |  |  |  |

