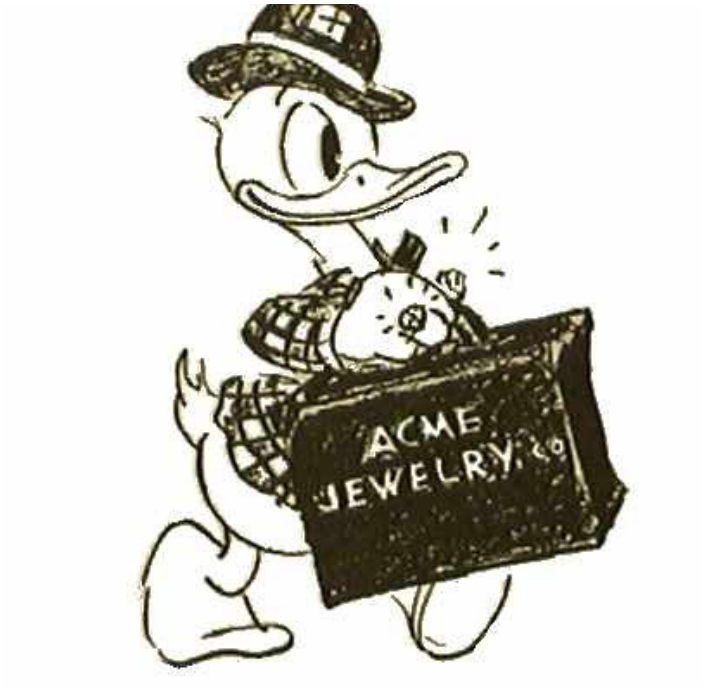
Algorítmica

Memoria Práctica 4

(Parte 2)



Grupo 1

Alba Moyano, Mª Mercedes

Andrades Molina, Christian

Azpeitia Muñoz, Fco.Javier
Keane Cañizares, Miguel
Muriel Sánchez lafuente, Guillermo

1.- Introducción

Esta práctica consiste en la resolución del **problema del viajante de comercio** mediante la técnica Branch and Baund.

Como sabemos, el problema del viajante de comercio es de los denominados problemas NP-Completos. Esto quiere decir que no se puede encontrar una solución óptima mediante algoritmos Greedy. Pero, en este caso, con Branch and Bound se encuentra una solución óptima.

Por eso, deberemos de implementar un algoritmo Branch and Bound que resuelva este problema.

2.- Descripción 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 óptimas 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 ramificación y poda, que exploren el espacio de posibles soluciones de forma más exhaustiva.

Así, un algoritmo de vuelta atrás comenzaría en la ciudad 1 (podemos suponer sin pérdida de generalidad, al tratarse de encontrar un tour, que la ciudad de inicio y fin es esa ciudad) e intentaría incluir como parte del tour la siguiente ciudad aún 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) sólo aquellas que satisfagan las restricciones del problema (en este caso ciudades que aún no hayan sido visitadas). Cuando para un nivel no queden más ciudades válidas, el algoritmo hace una vuelta atrás proponiendo una nueva ciudad válida para el nivel anterior.

Para emplear un algoritmo de ramificación 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 sería la siguiente: como sabemos cuáles son las ciudades que faltan por visitar, una estimación optimista del costo que aún nos queda será, 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 práctica 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 empleará el criterio LC o “más prometedor”. En este caso consideraremos como nodo más prometedor aquel que presente el menor valor de cota inferior. Para ello se debe 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 para resolver el problema.

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

3.- Algoritmo Branch and Bound

Como ya se ha descrito en el apartado anterior, tendremos que usar los siguientes datos, además del siguiente algoritmo greedy:

```
------Datos usados
matriz[][]              //matriz de distancias
solucion_final[]        //vector con el recorrido de la solución óptima
sol[]                   //vector con las soluciones temporales, iniciamente con todos las ciudades
aux[]                   //vector auxiliar de pairs <nodo><cota_inf>
stack_vivos             //pila de pairs <nodo><nivel>
CS                      //Cota superior
n                       //número de ciudades a visitar
nivel = 0               //nivel del arbol.
nodo_vivo               //nodo vivo que se está analizando
------fin Datos usados

------Algoritmo Greedy
CS=greedy_practica3(sol[], n, matriz[][]);
------fin Algoritmo Greedy
```

Una vez descritos los datos, tendremos que hacer uso de ellos en el siguiente pseudocódigo que implementa la solución al problema planteado:

```
--------Branch and Bound
stack_vivos.push(sol[nivel],nivel)          //introducimos un primer elemento
nodo_vivo=sol[nivel]
while ( ! stack_vivos.isEmpty())    //bucle principal que termina cuando no quedan nodos vivos
        if (nivel == n-1)                   //estamos en un nodo hoja
                c=cota_inferior(sol[],nivel,nodo_vivo, matriz[][])
                if (c <= CS)
                        CS=c
                        copy (sol[], solucion_final[], n)
        else
                for(i=0 ; i< n-nivel-1)     //creamos todos los hijos
                        aux[i].first = sol[n-i-1]       //de atrás a adelante en el vector
                        aux[i].second = cota_inferior(sol[],nivel,aux[i].first, matriz[][])
                aux[].ordenaCI()        //ordenamos el vector según su CI, decreciente
                for (i=0; i < aux[].size(); i++)    //para cada hijo
                        if (aux[i].second <= CS)        //Si el nodo es un buen candidato
                                stack_vivos.push(aux[i].first, nivel+1)  //introducir en la pila
                nodo_vivo=stack_vivos.top().first
                nivel = stack_vivos.top().second
                stack_vivos,pop()
                swap (sol[nivel],sol[sol.indexOf(nodo_vivo)])
--------fin Branch and Bound
```

4.- Visualización de las soluciones