Branch & Bound

PRÁCTICA 4

Problema del Viajante de Comercio

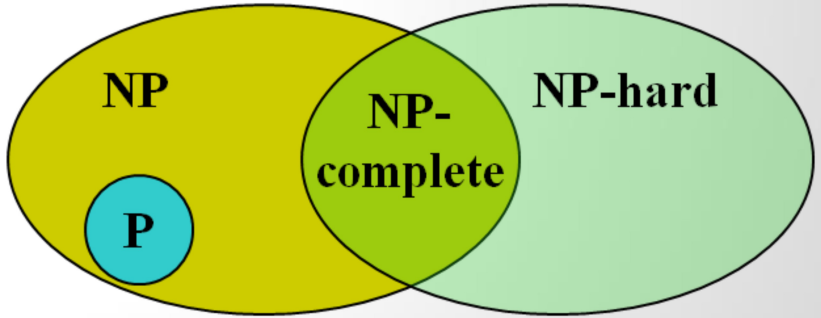
Enunciado del problema

En su formulación más sencilla, **el problema del viajante de comercio** (TSP, por Traveling Salesman Problem) se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Emplearemos en esta práctica el algoritmo de **ramificación y poda**.

Complejidad NP-Completo

El problema del viajante de comercio pertenecen a la clase de complejidad de los **NP-completos**.

La solución óptima no puede ser obtenida mediante técnicas Greedy, pero si mediante otras heurísticas válidas como la utilizada en está práctica (Branch &



Algoritmo Branch & Bound

Elementos

MATRIZ [][]

	0	1	2	3	4
0	0	2	8	3	6
1	2	0	4	8	8
2	8	4	0	7	3
3	3	8	7	0	3
4	6	8	3	3	0

Matriz cuadrada rellenada con las distancias de cada ciudad con el resto del conjunto.

SOLUCION_FINAL[][]

0	1	2	4	3

Vector con el recorrido final que será la solución óptima

Algoritmo Branch & Bound

Elementos

SOL []

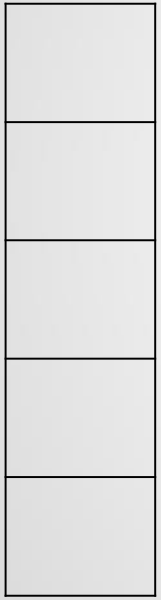
Vector que almacena la solución parcial hasta el momento (inicialmente con todas las ciudades disponibles). Lo utilizaremos para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda

AUX []

Vector auxiliar de pairs <nodo> (ciudad) y <cota_inf> (Suma de la estimación optimista del camino restante, más el coste del camino ya acumulado). Utilizaremos esa cota_inf para la poda.

Algoritmo Branch & Bound

Elementos



STACK_VIVOS

Pila de pairs <nodo><nivel>. Pila en la que introduciremos los nodos que cumplen con las restricciones (cota_inf <= Cota_Sup) y son por lo tanto buenos candidatos.

variable : NODO_VIVO

Variable que almacena el nodo que se está analizando en este instante y que sacamos del vector sol[].

Algoritmo Branch & Bound

Elementos

NIVEL = 0	N	CS
Variable que almacena el nivel del árbol en el que nos encontramos. Si el nivel es == n (tamaño) -1 nos encontramos en un nodo hoja.	Número total de ciudades en el circuito.	Cota superior. Almacena el costo de la mejor solución hasta ahora. La usaremos para comparar con la cota inferior de la solución parcial y podar o no. La precalculamos usando un algoritmo Greedy. (vecino mas cercano)

Algoritmo Branch & Bound

Datos general

- DATOS

matriz [ ][ ] // Matriz distancias
solucion_final[ ] // Vector con recorrido de la solución óptima
sol[ ] // Vector con las soluciones temporales
aux[ ] // Vector aux <nodo> <cota_inf>
stack_vivos // Pila pairs <nodo> <nivel>
CS // Cota superior
n // Número de ciudades
a visitar
nivel = 0 // Nivel del árbol
nodo_vivo // Nodo vivo que se está analizando

- FUNCIÓN

CS = greedy_practica4(sol[ ], n, matriz[ ][ ]);

Algoritmo Branch & Bound

Pseudocódigo

```
stack_vivos.push(sol[nivel], nivel)                    // Introducimos un primer elemento
nodo_vivo = sol[nivel]

while (! stack_vivos.isEmpty())                        // Bucle principal. Termina cuando no hay nodos vivos
        if (nivel == n-1)                                                  // Es 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-n-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 en orden 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)    //Introducimos en la pila el nodo
```