

# Análisis de algoritmos: Practica 3b.

# **Algoritmos de Ramificación y Poda.**

Daniel Lois Nuevo

Adrián García Oller

1. **Técnica de vuelta atrás.**
2. **Técnica de ramificación y poda.**
3. **Definición de la función de cota:** Es el mínimo coste a partir de las elecciones de sedes ya realizadas según el siguiente algoritmo heurístico:

public static int sedesC(int[] c0, int[] c1, int f,int solFij,int sede){

        int coste = solFij;

        int anterior = sede;

        for (int i = 0; i < c0.length;i++){

            if ((anterior == 0)){

                if (c0[i] < (c1[i] + f)){

                    coste = coste + c0[i];

                    anterior = 0;

                } else{

                    coste = coste + c1[i] + f;

                    anterior = 1;

                }

            } else if (anterior == 1) {

                if ((c0[i]+ f) < c1[i]){

                    coste = coste + c0[i] + f;

                    anterior = 0;

                } else{

                    coste = coste + c1[i];

                    anterior = 1;

                }

            }

        }

        return coste;

    }

Dicho algoritmo heurístico usa una función de selección que elige en función del coste de la sede mas el coste de traslado si se cambia de sede.

**Valor inicial:** será el menor coste según el algoritmo anterior sin haber elegido ninguna sede aún.

**Actualización en cada nodo del árbol de búsqueda:** se llamará de nuevo al algoritmo anterior para obtener el mínimo coste a partir de las decisiones ya tomadas.

b)

Hemos usado dos funciones, una principal que recoge los datos de entrada y transforma los dos arrays en una matriz, inicializa la solución optima como infinito, pero como se puede hemos elegido el valor más grande posible, inicializamos la solAux , anterior (la sede donde ha estado el anterior mes) e i (i=mes) a cero , inicializamos la cota al valor explicado anteriormente y llamamos a la función auxiliar que es el algoritmo de ramifica y poda.

**public** **static** **int** sedesP (**int**[] c0, **int**[] c1, **int** f) {

**int** i = 0;

**int** aux;

**int** cambio = 0;

**int** sol = 2147483647;

**int** solAux = 0;

**int**[][] cAux = **new** **int**[2][c0.length];

**int** cota=*sedesC*(c0,c1, f,0,0);

**for** (**int** k = 0; k < c0.length; k++){

cAux[0][k] = c0[k];

}

**for** (**int** k = 0; k < c0.length; k++){

cAux[1][k] = c1[k];

}

aux = *sedesAuxP*(i,sol,solAux , cAux ,f, cambio,cota);

**return** aux;

}

En este algoritmo por cada una de las sedes decidimos si es el primer mes no usaremos el coste de traslado al escoger sede, y si es el único mes a tener en cuenta comprobara si la solución óptima es peor que solAux y si es el caso asignara a sol(solución óptima) el valor de solAux y devolverá la solución, si no, llamará de nuevo a esta función con el siguiente mes (i++) , la solAux más el coste de la sede k en el mes i, la misma sol , la misma matriz de costes y el mismo coste de traslado y la sede k, y recogerá la sol devuelta en el caso de que solAux sea menor que la cota.

Si no es el primer mes comprueba si es el último mes, si lo es comprobara si la solución óptima es peor que solAux y si es el caso asignara a sol (solución óptima) el valor de solAux y devolverá la solución.

Si es un mes intermedio comprobara si la anterior sede y la actual son iguales, si lo son hará lo mismo que para el primer mes, si no, además de lo que hace para el primer mes añadirá a solAux el coste de traslado.

**private** **static** **int** sedesAuxP(**int** i,**int** sol, **int** solAux, **int**[][] cAux, **int** f,**int** anterior,**int** cota){

**for** (**int** k = 0; k < 2; k++){

**if** (i==0){

**if** (i == (cAux[k].length)-1){

solAux=cAux[k][i];

**if**(sol>solAux) {

sol=solAux;

}

**return** sol;

}

**else**{

anterior =k;

solAux += cAux[k][i];

i++;

**if**(solAux<=cota) {

**int** c0[]=**new** **int**[cAux[k].length];

**int** c1[]=**new** **int**[cAux[k].length];

**int** l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c0[l]=cAux[0][c];

}

l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c1[l]=cAux[1][c];

}

**int** aux=cota;

cota =*sedesC*(c0, c1, f,solAux,anterior);

sol=*sedesAuxP*(i, sol, solAux, cAux, f, anterior,cota);

cota=aux;

}

i--;

solAux -= cAux[k][i];

}

}

**else**{

**if** (i == (cAux[k].length)){

**if**(sol>solAux){

sol=solAux;

}

**return** sol;

}

**else**{

**if**(anterior==k){

solAux+=cAux[k][i];

i++;

**if**(solAux<=cota) {

**int** c0[]=**new** **int**[cAux[k].length];

**int** c1[]=**new** **int**[cAux[k].length];

**int** l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c0[l]=cAux[0][c];

}

l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c1[l]=cAux[1][c];

}

**int** aux=cota;

cota =*sedesC*(c0, c1, f,solAux,k);

sol=*sedesAuxP*(i, sol, solAux, cAux, f, k,cota);

cota=aux;

}

i--;

solAux-=cAux[k][i];

}

**else**{

solAux+=cAux[k][i]+f;

i++;

**if**(solAux<=cota) {

**int** c0[]=**new** **int**[cAux[k].length];

**int** c1[]=**new** **int**[cAux[k].length];

**int** l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c0[l]=cAux[0][c];

}

l=0;

**for**(**int** c=i;c<cAux[k].length;c++,l++) {

c1[l]=cAux[1][c];

}

**int** aux=cota;

cota =*sedesC*(c0, c1, f,solAux,k);

sol=*sedesAuxP*(i, sol, solAux, cAux, f, k,cota);

cota=aux;

}

i--;

solAux-=cAux[k][i]+f;

}

}

}

}

**return** sol;

}

1. Comparación de optimalidad**.**
2. **Conclusiones.**