

# 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** coste = 0;

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

**if** (c0[i] < c1[i]){

coste += c0[i];

} **else** {

coste += c1[i];

}

}

**return** coste;

}

Dicho algoritmo heurístico usa una función de selección que elige en función del coste menor de entre las dos sedes.

**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** cambio = 0;

**int** sol =2147483647;

**int** solAux = 0;

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

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

**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];

}

**int** 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,recalculará la cota y 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 además de comparar si la sol que se haya escogido es mayor que la cota hará lo mismo que para el primer mes,en caso contrario poda la rama, si cambia de sede, además de lo que hace en caso de que sea la misma sede 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;

**int** aux=cota;

solAux += cAux[k][i];

i++;

**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];

}

cota =solAux+*sedesC*(c0, c1);

**if**(sol>=cota) {

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){

**int** aux=cota;

solAux+=cAux[k][i];

i++;

**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];

}

cota =solAux+*sedesC*(c0, c1);

**if**(sol>=cota) {

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

}

cota=aux;

i--;

solAux-=cAux[k][i];

}

**else**{

**int** aux=cota;

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

i++;

**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];

}

cota =solAux+*sedesC*(c0, c1);

**if**(sol>=cota) {

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

}

cota=aux;

i--;

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

}

}

}

}

**return** sol;

}

1. **Comparación de optimalidad.**
2. El método sedes es el vuelta atrás.

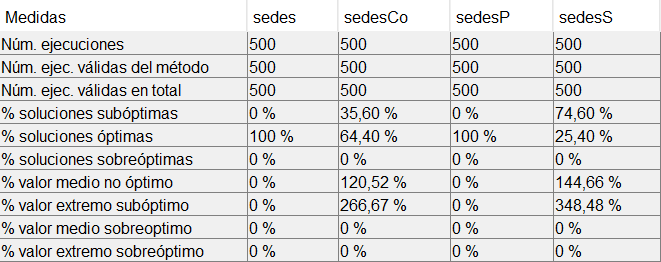
El método sedesCo es un heurísticos que compara los costes de cada mes más el coste de traslado si cambiara de sede y elige el menor.

El método sedesS es un heurísticos que compara los costes de cada mes y elige el menor.

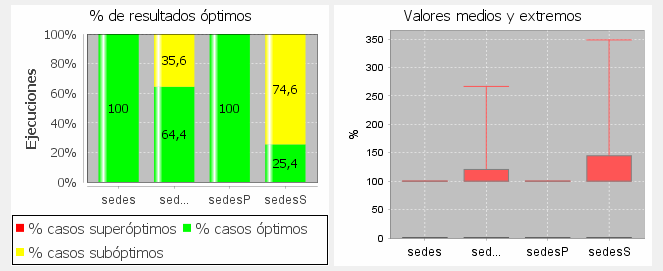
El método sedesP es el ramifica y poda.

1. Los algoritmos sedes y sedesP son exactos ya que siempre devuelven la solución óptima, por el contrario, sedesCo y sedesS no lo son ya que devuelven soluciones validas, pero no óptimas.
2. Para comprobar la optimidad del algoritmo hemos realizado dos baterias de pruebas.La primera con 500 ejecuciones y la segunda con 1000.

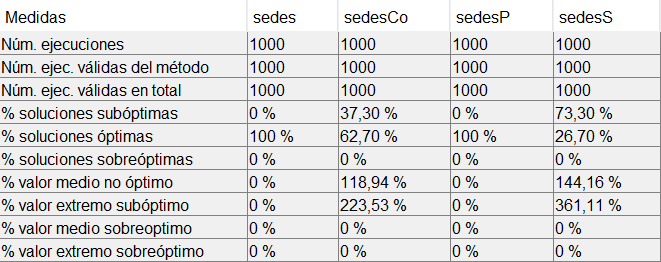
Resumen numérico para 500 ejecuciones.



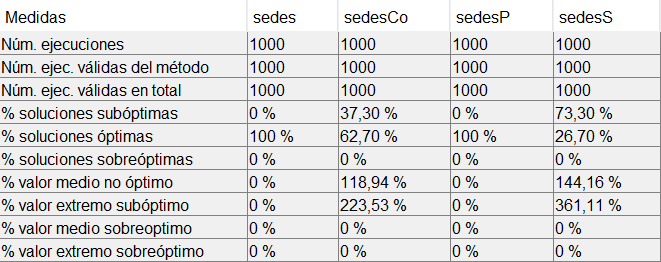
Resumen grafico para 500 ejecuciones.



Resumen numérico para 1000 ejecuciones.



Resumen grafico para 1000 ejecuciones.



Como era de esperar nuestro algoritmo de backtracking obtenía el mismo número de resultados óptimos que nuestro algoritmo de ramifica y poda, siendo un 100% de los casos el resultado óptimo. Pero el algoritmo de ramifica y poda, al no explorar todas las ramas del árbol de búsqueda, resulta más eficiente a pesar de obtener el mismo resultado que el backtracking en todos los casos.

1. No han sido necesarios cambios en ninguno de los algoritmos comparados ya que se hicieron las modificaciones pertinentes en la anterior práctica.
2. **Conclusiones.**

Esta práctica ayuda a comprender los algoritmos de ramifica y poda y a practicar con herramientas de estudio de algoritmos. En cuanto a la dificultad de la práctica creemos que tiene una complejidad notablemente mayor a la anterior por la complejidad en la decisión del tipo de función de cota a emplear. Gracias a optimex hemos podido comprobar que tanto el backtracking como el ramifica y poda son exactos. Nos costó entender el funcionamiento de la función de cota para minimización.