PRÁCTICA 3:
ALGORITMOS VORACES

Integrantes del grupo:

Ana López Mohedano
Adrián Anguita Muñoz
Marina Jun Carranza Sánchez
Pedro Antonio Mayorgas Parejo
Rafael Jiménez Márquez

ÍNDICE

---

0. Aclaraciones previas

Para la resolución de problemas propuestos, se va a llevar a cabo el análisis de los siguientes algoritmos, cada cual se dividirá en tres apartados:

- **Diseño del algoritmo**: incluye la lista de componentes aplicada al problema en cuestión, así como el pseudocódigo basado en el esquema de Greedy.
- **Estudio de optimalidad**: argumenta si se trata de un algoritmo óptimo, dando un contraejemplo en caso contrario.
- **Ejemplo de uso**: una ejecución paso a paso de una instancia para explicar el funcionamiento del algoritmo.

Pautas para probar el código con makefile:

Se incluye una tarea “all” para compilar y probar todos los algoritmos, y también una orden de make de compilación y de ejecución para cada uno de ellos, por ejemplo: “make pX_compile” y “make pX_test”, siendo X el número del problema.

1. Problema de asignación de equipos

a) Diseño de algoritmo Greedy

Siendo n el número de estudiantes en la clase, el problema se modela con una matriz de entrada P de tamaño n×n, donde:

- Las filas y columnas representan los estudiantes desde 0 hasta n-1.
- P(i, j) indica el nivel de preferencia ([0,10]) del estudiante i para trabajar con j.
- El valor del emparejamiento (i, j) es el producto de P(i, j) × P(j, i).

El objetivo es encontrar un conjunto de n/2 emparejamientos tal que la suma de valores de emparejamientos sea máxima.

Componentes

- **Lista de candidatos**: todas las posibles parejas de estudiantes que pueden formarse, considerando que un par (i, j) equivale a (j, i), pero P(i, j) no tiene por qué ser igual que P(j, i).
- **Lista de candidatos usados**: pares que han sido considerados como solución, hayan sido factibles o no. Además, se llevará el recuento de los estudiantes que ya han sido asignados a un equipo mediante un vector de booleanos.
- **Criterio de selección**: el par de estudiantes con mayor valor de emparejamiento.
- **Criterio de factibilidad**: una pareja podrá insertarse en la solución siempre que esté compuesta por dos estudiantes distintos (P(i, i) es invalido) y ninguno de los dos esté en un equipo aún (no estén ya “usados”).
- **Función solución**: una subsolución será solución final cuando los *n* estudiantes ya estén en equipos; es decir, si el resultado contiene n/2 parejas.
- **Función objetivo**: maximizar la suma total de los valores de los emparejamientos que componen la solución.

Pseudocódigo Greedy

```
Función S = P1_Greedy(P<n,n>: matriz entrada)
   S = {∅}
   C = {todos los pares posibles}
   U = {n elementos a false}
   Ordenar(C)                              //matchValue decreciente
   Mientras C != ∅ y |S| < n/2, hacer:
      (i,j) = seleccionar(C)               //par con mayor matchValue
      C = C \ (i,j)
      Si (!U(i) y !U(j)), hacer:           //factibilidad
         S = S ∪ {(i,j)}
   Fin-Mientras
   Devolver S
Fin-Función
```

b) Estudio de optimalidad

Para refutar su optimalidad, se propone el siguiente contraejemplo:

Hay 6 estudiantes y se tiene la siguiente matriz de niveles de preferencia (P: 6×6):

```
   0  1  2  3  4  5
0 [-, 2, 3, 4, 5, 6 ]
1 [3, -, 5, 2, 9, 10]
2 [8, 2, -, 6, 7, 1 ]
3 [4, 6, 3, -, 5, 2 ]
4 [6, 5, 10, 8, -, 2]
5 [5, 3, 4, 3, 6, - ]
```

El algoritmo Greedy proporciona la solución que viene a continuación (en el siguiente apartado se explica paso por paso cómo se ha llegado a ella):

- ➢ Pareja 1: 2 y 4 → [P(2,4)=7] × [P(4,2)=10] = 70
- ➢ Pareja 2: 0 y 5 → [P(0,5)=6] ×[P(5,0)=5] = 30
- ➢ Pareja 3: 1 y 3 → [P(1,3)=2] × [P(3,1)=6] = 12

Por tanto el valor total de emparejamiento es 70+30+12=**112**

Sin embargo, la solución obtenida con este algoritmo voraz no es la óptima, ya que existen combinaciones de emparejamientos con las que se obtiene un mayor valor.

Como por ejemplo, la que tiene como valor total de emparejamiento **116**:

- ➢ Pareja 1: 1 y 5 → 10×3 = 30
- ➢ Pareja 2: 0 y 3 → 4×4 = 16
- ➢ Pareja 3: 2 y 4 → 7×10 = 70

Debido a la existencia de soluciones mejores y la no garantía del algoritmo de obtenerla para todos los casos, podemos afirmar que **no es óptimo**.

c) Ejemplo de uso

Se va a explicar una instancia del problema, usando la matriz **P** del apartado anterior.

```
    0  1  2  3  4  5
0 [-, 2, 3, 4, 5, 6 ]
1 [3, -, 5, 2, 9, 10]
2 [8, 2, -, 6, 7, 1 ]
3 [4, 6, 3, -, 5, 2 ]
4 [6, 5, 10, 8, -, 2]
5 [5, 3, 4, 3, 6, - ]
```

1) Se inicializan los vectores: solución (**S**), candidatos (**C**) y usados (**U**).

2) Se generan los candidatos, que serán todas las parejas no repetidas que pueden formarse, junto con sus respectivos valores de emparejamiento (calculados mediante productos de los elementos de P).
   a) 0 y 1 (6), 0 y 2 (24), 0 y 3 (16), 0 y 4 (30), 0 y 5 (30)
   b) 1 y 2 (10), 1 y 3 (12), 1 y 4 (45), 1 y 5 (30)
   c) 2 y 3 (18), 2 y 4 (70), 2 y 5 (4)
   d) 3 y 4 (40), 3 y 5 (6)
   e) 4 y 5 (12)

3) Se ordena C de forma decreciente con respecto al valor de emparejamiento.
   a) (2,4) 70 > (1,4) 45 > (3,4) 40 > (0,4) 30 > (0,5) 30 > (1,5) 30 > (0,2) 24 > (2,3) 18 > (0,3) 16 > (1,3) 12 > (4,5) 12 > (1,2) 10 > (0,1) 6 > (3,5) 6 > (2,5) 4

4) Mientras queden elementos en C y el vector S tenga menos de 3 parejas:
   a) Se selecciona el primer par (i, j) de la lista C, que como está ordenada se corresponde con la del mayor valor de emparejamiento.
   b) Si ninguno de los dos seleccionados está en un equipo, entonces el par puede añadirse a S (es factible). Después se marcan como “usados” en el vector U y se suprimen de C.
      i) Pareja 1: **2 y 4** (**70**) → es factible, se añade a S.
         2 y 4 se marcan como “usados” y se eliminan de C.
      ii) Pareja 2: 1 y 4 (45) → 4 ya tiene pareja, no es factible.
         1 y 4 se eliminan de C.
      iii) Pareja 2: 3 y 4 (40), 0 y 4 (30) → 4 usado, no factibles, quitar de C.
      iv) Pareja 2: **0 y 5** (**30**) → factible, se añade a S.
         Se marcan 0 y 5 como usados y se eliminan de C.
      v) Pareja 3: 1 y 5 (30), 0 y 2 (24), 2 y 3 (18), 0 y 3 (16) → no factibles
         0, 2 y 5 ya usados, se eliminan estas parejas de C.
      vi) Pareja 3: **1 y 3** (**12**) → factible, se añade a S.

Por tanto, al final la solución es: **S={(2,4), (0,5), (1,3)}**, con valor de **112**.

2. Problema de invitados de una cena de gala

a) Diseño de algoritmo Greedy

Componentes

- **Lista de candidatos**: matriz de adyacencia con pesos. Donde cada C[i,j] se evalúa con un peso de conveniencia para sentarse a un lado.
- **Lista de candidatos usados**: se identifican los comensales que se van a sentar en un lado concreto como posibilidad.
- **Criterio de selección**: la arista con el mayor valor posible de cada comensal.
- **Criterio de factibilidad**: un comensal se pondrá a un lado siempre que su valor sea el mayor posible, no sea el mismo y no esté al lado de otro ya seleccionado.
- **Función solución**: la solución se alcanzará cuando se haya terminado de evaluar toda la matriz de adyacencia máxima para cada comensal.
- **Función objetivo**: maximizar la suma total de conveniencia de la mesa, seleccionando los comensales que deben sentarse cada uno a un lado.

Pseudocódigo Greedy

```
Function P2_Greedy(C = [][]: Adyacency Matrix }
    // Comensales ya sentados
    S = {size(C): 0}

   // Comensales visitados o candidatos
   CV = {size(C): Boolean/Bit}
   max = 1
   c = 0

   // Comensal inicial
   CV[c] = 1

   Mientras i < size(C) y j < size(C[i]) hacer:
      Si i ⩾ Size(c):
        Salir

      Si i ≠ y (CV[j] == 0) y C[i][j] > max entonces:
        max = C[i][j]
        // Marcar comensal como posible solucion al actual
        S[c] = j
      Fin-Si

      j+= 1

      Si j == size(C[i]) entonces:
        max = 1
        j = 0
        // Marcamos que ese comensal se ha tenido en cuenta
        // para la solucion
        CV[sentados[c]] = 1
        // Seleccionamos otro comensal a evaluar.
        i = sentados[c]  - 1
        c += 1
      Fin-Si
   Fin-Mientras
   Devolver S
Fin-Función
```

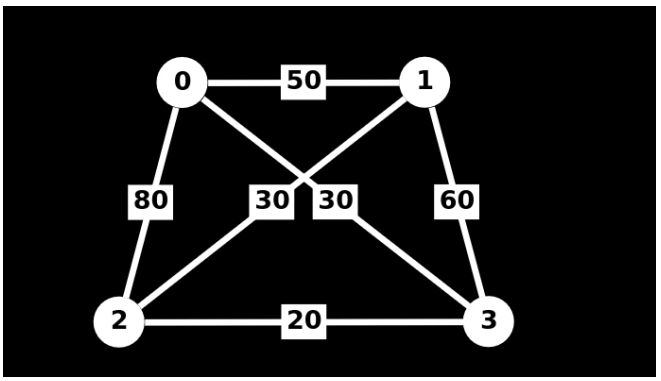
b) Estudio de optimalidad

La base del algoritmo de la cena de gala, consiste en una versión maximal del algoritmo del viajante. Donde cada comensal tiene una matriz de adyacencia con todos con un peso o nivel de conveniencia entre 1 <= Conveniencia <= 100. Por diseño de la matriz de adyacencia para representar el grafo, se consideran los ceros como la conveniencia nula para evitar los ciclos contra el vértice mismo.

Conocemos como C: La matriz de adyacencia que representa el grafo cuyas aristas son la conveniencia de sentarse uno al lado del otro.
Conocemos como S: Como el vector solución que marca quién se sienta a un lado formando, cada uno de los cuales sin repetirse va teniendo a un comensal a un lado cuyo nivel de conveniencia sea el máximo.
Conocemos como CV: Como vector auxiliar de visitados que impide la repetición y formación de ciclos.

Para los candidatos de ejemplo tenemos la siguiente matriz (Grafo en la siguiente imagen).

```
# Input Matrix
C = [   [0, 50, 80, 30],
        [50, 0, 30, 60],
        [80, 30, 0, 20],
        [30, 60, 20, 0]]
```



La solución saldría como la siguiente:

```
0 -> 2 VALUE: 80
1 -> 3 VALUE: 60
2 -> 1 VALUE: 30
3 -> 0 VALUE: 30
Maximal Convenience is : 200
```

El funcionamiento consiste en:

1. Partiendo de cualquier candidato arbitrario (en el ejemplo el 0) buscamos el comensal cuyo nivel de conveniencia sea el máximo posible.
2. Una vez obtenido dicho comensal iterando sobre todos los posibles candidatos. Se almacena.
3. Cuando se almacena, se coge el siguiente candidato que es el candidatos elegido restando uno para poder evaluar toda la matriz.
4. Repetir hasta terminar.

Es óptimo y rápido seleccionando los candidatos. Pero tiene un problema, evalúa sólo una posibilidad desde el punto de partida elegido, no tiene en cuenta otros.

Para solucionar esto, quizás sea mejor usar Programación Dinámica, pero tiene un coste mucho más elevado en memoria y CPU. Ya que tiene que almacenar todas las posibilidades.

c) Ejemplo de uso

1. Partiendo del candidato 0 se evalúan todos los demás candidatos. Se escoge el máximo candidato que tenga un puntaje mayor del criterio. Una vez valorados todos, se almacena cuál es el que se ha seleccionado y se actualiza el vector de los seleccionados CV.
2. Luego para escoger el siguiente a evaluar, se coge a partir del candidato elegido actual se escoge el anterior, es decir si en el caso anterior se escoge el 2 como el ideal. El siguiente a evaluar sería 2 - 1 = 1.
3. Luego se repite el paso 1 hasta encontrar que para el comensal 1, se selecciona el 3 y el siguiente candidato a evaluar es el 2 para escoger su lado izquierdo.

3. Problema de las gasolineras

a) Diseño de algoritmo Greedy

Componentes

- **Lista de candidatos:** lista de las gasolineras que se encuentran en la ruta
- **Lista de candidatos usados:** las gasolineras en las cuales paramos a repostar
- **Criterio de selección:** elegimos la gasolinera más lejana a la que podemos ir con el combustible que tenemos
- **Criterio de factibilidad:** pararemos en una gasolinera a repostar si con el combustible que tenemos no llegamos a la siguiente gasolinera o al objetivo.
- **Función solución:** un conjunto de gasolineras será solución cuando podamos llegar al objetivo sin tener que repostar en la siguiente gasolinera o cuando la siguiente parada sea en el objetivo.
- **Función objetivo:** realizar el mínimo número de paradas posibles antes de llegar al destino.

Pseudocódigo Greedy

```
Función S = P3_Greedy (gasolineras: V entrada, depósitoCombustible)
      S = {∅}
      C = lista de gasolineras
      U = {depositoCombustibleAux}
      Mientras (C != ∅) hacer:
            gasolineraMásLejana = 0;
            U = depositoCombustible
            terminar = Falso
            Para cada elemento gasolinera en C y no terminar:
                  U -= gasolinera
                  gasolineraMásLejana += gasolinera
                  Si(U < 0) entonces
                        U += gasolinera
                        gasolineraMásLejana -= gasolinera
                        terminar = verdadero
                  Fin-Si
                  Si(C != ∅)
                  C = C \ gasolinera
                  Fin-Si
                  S = S ∪ gasolineraMásLejana
      Fin-Mientras
      Devolver S
Fin-Función
```

b) Estudio de optimalidad

Demostraremos la optimalidad de este algoritmo por el método de inducción.

Se demostrara que:

1. vi pertenece S (vi != vo) => D[i] es el corset del camino óptimo desde vo a vi.
2. vi no pertenece S => D[i] es el coste del camino especial óptimo desde vo a vi.

Para empezar llamaremos S al conjunto de nodos elegidos. Tenemos un vector D que contiene la longitud del camino especial (camino de un nodo a otro en el cual todos los nodos intermedios están en S) más corto a cada nodo del grafo.

Necesitaremos una Base y una Hipotesis de induccion:

- ➢ Base:
  - Inicialmente D[a] = 0 y todos los demás D son mayores que cero, por tanto se elige el origen a.
  - D[a] es el camino más corto desde *a* hasta *a*.
  - T(1) por ende es cierta.
- ➢ Hipótesis de Inducción T(i):
  - El valor de D[v] da el camino mínimo desde el origen hasta el vértice *v*.

Tras haber definido lo anterior:
Suponemos que T(k) es cierto para todo k < i. En la iteración k + 1 seleccionamos el nodo v y nos hacemos la siguiente pregunta es D(v) el camino más corto desde *a* a *v.*

Supongamos que hay otro vértice *u* tal que el camino hasta *v* pasa por *u* es más corto que el anterior(el de D(v))

Si esto es así obligatoriamente el camino a través u tiene que ser mucho más largo y complicado.

Recordemos la pregunta anterior en la iteración k + 1 seleccionamos *v* ¿es D[v] el camino más corto desde *a* hasta *v*?

La respuesta a esto es que si, Por tanto T(k + 1) es cierto, como la base y la hipótesis de inducción son ciertas T(i) es cierta para todo i.

c) Ejemplo de uso

Ejemplo paso a paso del funcionamiento del algoritmo para una instancia pequeña.

Existen 5 gasolineras y tenemos 20 km de depósito.

Cada gasolinera se representa en función de la distancia a la que estamos:
g1 = 5
g2 = 20
g3 = 10
g4 = 15
g5 = 17

Empezamos en el punto 0 esto quiere decir que:
gasolineraMasLejana = 0
gasolinerasCandidatas [g1,g2,g3,g4,g5]
gasolinerasSolucion []

Cogemos la primera gasolinera y la actualizamos.
gasolineraMasLejana += 5 (g1) (la gasolinera más lejana tiene un valor de 5)
depositoCombustible -= 5 (g1) (valor del depósito = 15)

Como ya hemos usado g1, la eliminamos de las gasolineras candidatas.
gasolinerasCandidatas [g2,g3,g4,g5}

Como el depósito no se ha vaciado comprobamos la siguiente gasolinera.
gasolineraMasLejana += 20 (g2) (valor de gasolinera más lejana = 25)
depositoCombustible -= 20 (g2) (valor del depósito = -5)

Como ya hemos usado g2, la eliminamos de las gasolineras candidatas.
gasolinerasCandidatas [g3,g4,g5]

Como el depósito se ha vaciado (depósito < 0) añadimos a gasolinerasSolucion la gasolinera anterior.
gasolinerasSolucion [g1]

Repostamos en g1.
valor del depósito = 20
valor de la gasolinera más lejana = 0

Comprobamos la siguiente Gasolinera.
gasolineraMasLejana += 10 (g3) (valor de gasolinera más lejana = 10)
depositoCombustible -= 10 (g3= (valor de depositoCombustible = 10)

Como ya hemos usado g3, la eliminamos de las gasolineras candidatas.
gasolinerasCnadidatas [g4,g5]

Como el depósito no se ha vaciado comprobamos la siguiente gasolinera.
gasolineraMasLejana += 15 (g4) (valor de gasolinera más lejana = 25)
depositoCombustible -= 15 (g4) (valor del depósito = -5)

Como ya hemos usado g4, la eliminamos de las gasolineras candidatas.
gasolinerasCandidatas [g5]

Como el depósito se ha vaciado añadimos a gasolinerasSolucion la gasolinera anterior.
gasolinerasSolucion [g1,g3]

Repostamos en g3.
(valor del depósito = 20)
(valor de gasolinera más lejana = 0)

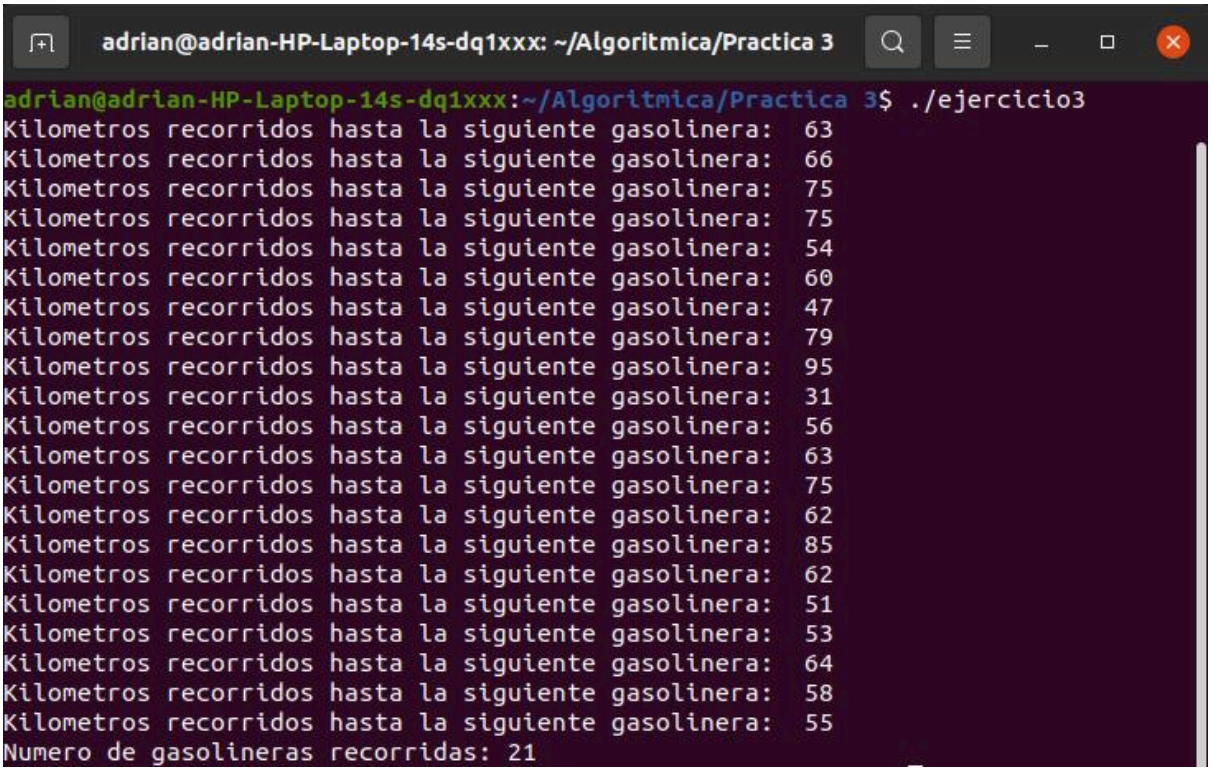
Comprobamos la siguiente gasolinera.
gasolineraMasLejana += 17 (g5) (valor de gasolinera más lejana = 17)
depositoCombustible -= 17 (g5) (valor del depósito = 3)

Como ya hemos usado g5, la eliminamos de las gasolineras candidatas
gasolinerasCandidatas []

Como el depósito no se ha vaciado comprobamos la siguiente gasolinera. Como no quedan más gasolineras ya tenemos nuestro vector solución.
gasolinerasSolucion [g1,g3].

Funcionamiento de la implementación:

```
vector<int> gasolinerasGreedy(vector<int> gasolineras, const int depositoCombustible){
    vector<int> plan_viaje;


    for (int i = 0; !gasolineras.empty(); i++) {
        int gasolineraMasLejana = 0;
        int depositoCombustibleaux = depositoCombustible;
    bool terminar = false;

        for(auto it = gasolineras.begin(); it != gasolineras.end() && !terminar; it++){
            depositoCombustibleaux -= (*it);
            gasolineraMasLejana += (*it);

            if(depositoCombustibleaux < 0){
                depositoCombustibleaux += (*it);
                gasolineraMasLejana -= (*it);
                terminar = true;
            }

            if(!gasolineras.empty()){
                it = gasolineras.erase(it);
            }
        }
        plan_viaje.push_back(gasolineraMasLejana);
    }

    return plan_viaje;
}
```



Hemos usado un vector con 50 gasolineras donde el tamaño del depósito es 100. Como podemos observar usamos 21 gasolineras para realizar todo el recorrido y en ningún momento superamos los 100 km de depósito.

4. Problema de red de sensores

a) Diseño de algoritmo Greedy

Componentes

- **Lista de candidatos**: los nodos sensores de la red inalámbrica.
- **Lista de candidatos usados**: los nodos sensores ya seleccionados.
- **Criterio de selección**: se selecciona un nodo sensor inicial al azar. Los siguientes nodos a seleccionar tienen que cumplir que la arista que lo une con el último nodo sensor ya seleccionado tenga el coste mínimo.
- **Criterio de factibilidad**: el nodo sensor seleccionado no puede estar seleccionado previamente, evitando de esta manera ciclos.
- **Función solución**: lista de nodos sensores desde el nodo sensor inicial al nodo servidor central.
- **Función objetivo**: la distancia de un nodo sensor inicial al nodo servidor central de la red (grafo) tiene que tener el coste mínimo.

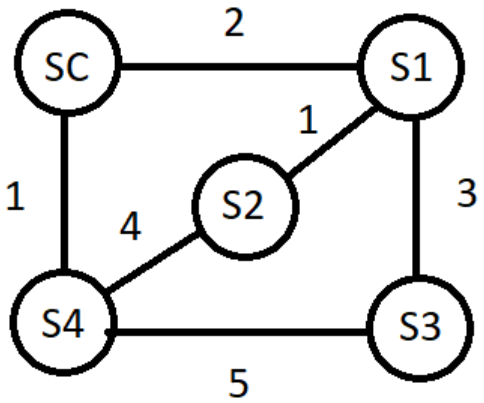
Pseudocódigo Greedy

```
Función P4_Greedy(grafo[FIL][COL], n, nodo_inicio, nodo_servidor):
     Vector "visitados" (tamaño n) inicializado a false
     Vector “solución” para almacenar el camino óptimo

     Insertar nodo_inicio como primer nodo en “solución”
     Marcar nodo_inicio a true en “visitados"
     Asignar el nodo_inicio como el nodo_actual

     Mientras nodo_actual sea distinto al nodo_servidor:
        Encontrar el nodo más cercano no visitado al nodo actual
     con una función auxiliar encontrarNodoMasCercano
        Si no se encuentra un nodo más cercano, salir de la función
        Fin-si
        Agregar nodo_mas_cercano a la “solución”
        Marcar nodo_mas_cercano a true en “visitados"
        Actualizar nodo_actual al nodo_mas_cercano
     Fin-Mientras
    Devolver la solución
Fin-Función
```

El pseudocódigo de la función auxiliar que utilizamos para encontrar el nodo más cercano no visitado es el siguiente:

```
Función encontrarNodoMasCercano(grafo[FIL][COL], visitados[n], n,
nodo_actual, nodo_servidor):
   nodo_mas_cercano = -1
   distancia_minima = INFINITO

   Para cada i en rango(0, n):
      Si i=nodo_sevidor y no visitados[i] y grafo[nodo_actual][i] <
      distancia_minima
            nodo_mas_cercano = i
            Devolver nodo_mas_cercano
      Fin-Si
     Si no visitados[i] y grafo[nodo_actual][i] < distancia_minima:
            distancia_minima = grafo[nodo_actual][i]
            nodo_mas_cercano = i
      Fin-Si

     Devolver nodo_mas_cercano
Fin-función
```

b) Estudio de optimalidad y ejemplo de uso



Si tenemos este grafo en el que SC es el servidor central y S1, S2, S3 y S4 los nodos sensores de esta red, podemos estudiar la optimalidad del algoritmo propuesto para la resolución de este problema.

SC tendrá como identificador el 0 y los demás nodos 1, 2, 3 y 4. Establecemos, por tanto, el nodo_servidor = 0 y, por ejemplo, el nodo_inicio = 2. Es claro que el algoritmo busca como nodo más cercano al primer nodo(2) el nodo 1 ya que su coste es el menor. Posteriormente se elegirá el nodo 0 y acaba el algoritmo. Podemos ver como en este caso el algoritmo ha

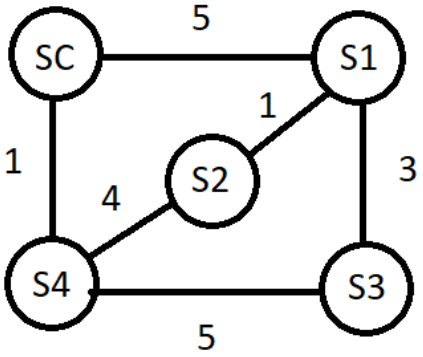
encontrado una solución que coincide con la solución óptima para estos nodos elegidos. Como una de las características de los algoritmos greedy es que toman decisiones localmente óptimas en cada paso, sin considerar el panorama completo del problema, puede llevar a soluciones que son son la óptima en algunos casos, donde una solución que parezca buena en un paso no conduzca a la mejor solución global.

Si miramos nuestro grafo de ejemplo, esto podría ocurrir si el nodo_inicio lo establecemos al nodo 3. En este caso, se elegiría el nodo 1 como el más cercano pero el siguiente en ser elegido sería el 2 ya que tiene menor coste que llegar al nodo 0, aunque es el nodo objetivo. Lo mismo ocurriría si elegimos como nodo_inicio = 1. El nodo más cercano elegido sería también el 2 pero volvemos a conseguir una solución que no es óptima.

Este problema se ha podido solucionar optimizando un poco el algoritmo estableciendo una comprobación para ver si el nodo siguiente al último elegido coincide con el nodo_servidor. Si es así, se devuelve el nodo_servidor.

Esta optimización no soluciona la desventaja del algoritmo greedy al no asergurar poder obtener siempre la solución óptima pero es la mejor aproximación que hemos podido realizar.

Da la casualidad que en este grafo ejemplo, obtenemos la solución óptima con el algoritmo propuesto sea cual sea el nodo_inicio y manteniendo el nodo_servidor = 0. Podemos cambiar el grafo para dar un contraejemplo en el que podamos ver como no obtenemos la solución óptima. Bastaría con cambiar el peso de la arista que une los nodos 0-1 a 5 en lugar de 2. Con este cambio de pesos, tendríamos este grafo:



Si establecemos estos pesos podemos ver algunos contraejemplos. Si nodo_inicio = 2, el camino elegido sería 2, 1, 0. Podemos observar que la solución óptima es el camino 2, 4, 0 ya que presenta un coste menor. Otro contraejemplo es si ponemos nodo_inicio = 3. En este caso, el camino elegido es 3, 1, 0 mientras que con el camino 3, 4, 0 obtenemos la solución óptima.

Terminando, podemos concluir que el algoritmo es eficiente pero **no es óptimo** aunque hemos establecido un condición que mejora ligeramente y aumenta las veces que el algoritmo puede darnos el mejor resultado posible.

5. Problema de embaldosado de un pueblo

a) Diseño de algoritmo Greedy

Componentes

- **Lista de candidatos:** todas las conexiones entre plazas en la ciudad. Esta lista se inicializa como C = A, donde A es el conjunto de todas las aristas del grafo de entrada G.
- **Lista de candidatos usados:** esta lista se refiere a las aristas que ya han sido seleccionadas y añadidas al conjunto solución S.
- **Criterio de selección:** es la función que selecciona la arista más corta en la lista de candidatos C. Esto se hace en la línea “x= Seleccionar arista más corta en C”.
- **Criterio de factibilidad:** es la verificación de que la adición de la arista seleccionada x al conjunto solución S no forma ciclos con el grafo V. Esto se evalúa con “Si S U {x} no forma ciclos en V”.
- **Función solución:** se refiere al proceso de añadir la arista seleccionada x al conjunto solución S si cumple con el criterio de factibilidad. Esto se realiza con la línea “S= S U {x}”.
- **Función objetivo:** minimizar el número de calles que se compran para conectar todas las plazas, y que estas sean lo más baratas posibles.

Pseudocódigo Greedy

```
Algoritmo S=Greedy(G=<V, A>: Grafo de entrada)
  S = {} # Conjunto vacío
  C = A # Candidatos
  Mientras |S| < (|V| - 1), hacer:
    x = Seleccionar arista más corta en C
    C = C \ {x} # Se elimina de candidatos
    Si S U {x} no forma ciclos en V, hacer:
      S = S U {x} # Se añade a la solución
  Devolver S
```

El código utilizado para la práctica es una plantilla de Árbol generador minimal de Kruskal, que podremos usar para el ejemplo más adelante metiendo el grafo que queramos.

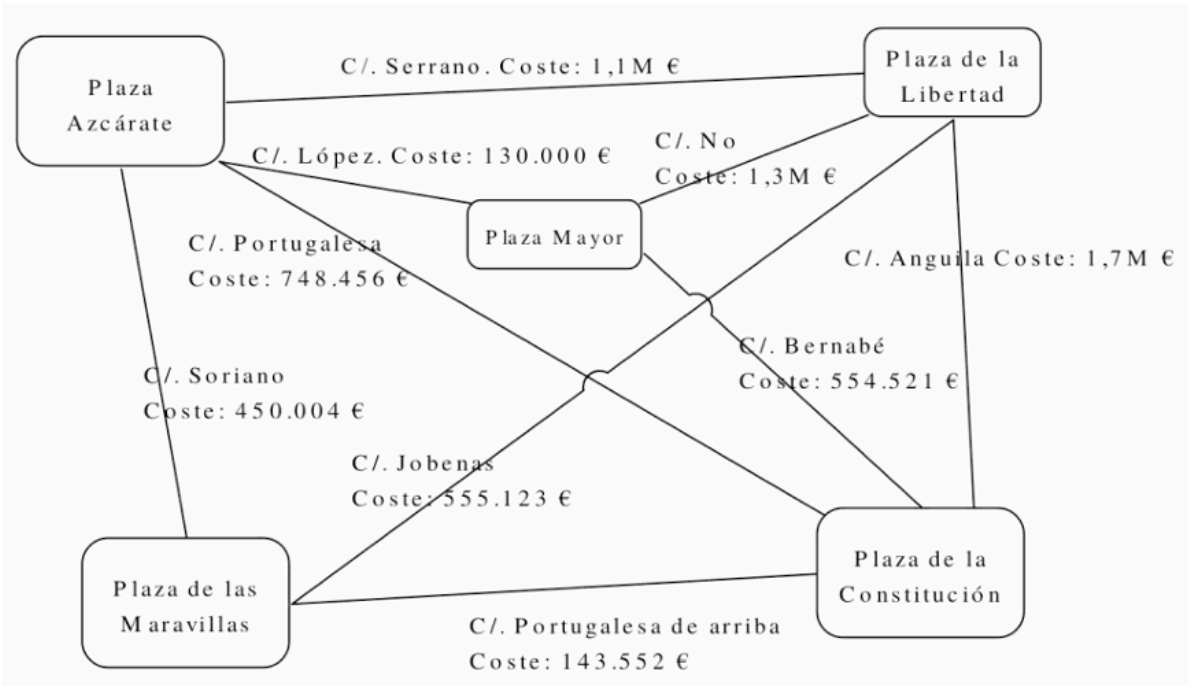
b) Estudio de optimalidad

Este algoritmo greedy obtiene siempre el óptimo, ya que es un árbol de recubrimiento de coste mínimo, en este caso de Kruskal. Es un MST Kruskal porque este método selecciona cualquier arista del grafo, sin recorrer vértices como en Prim. La eficacia de este enfoque se debe a la propiedad de subestructura óptima del problema del MST, que establece que la solución óptima global a un problema puede ser construida a partir de las soluciones óptimas de sus subproblemas.

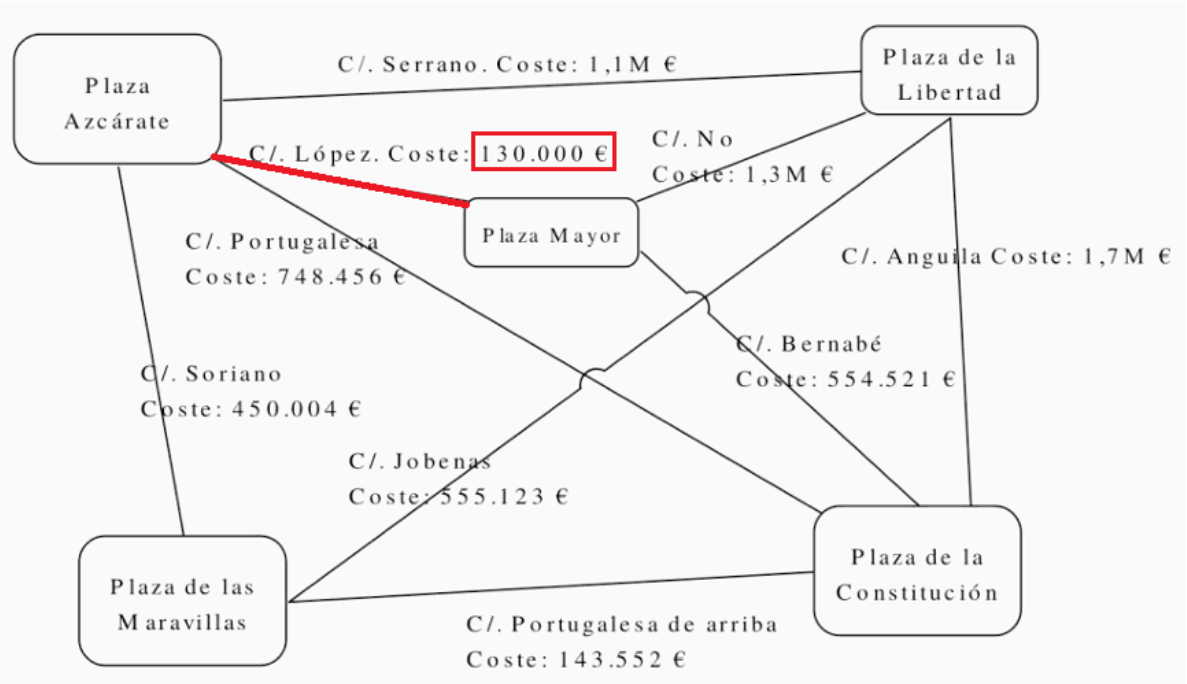
En el caso del MST, la solución óptima global (el MST) puede ser construida a partir de las soluciones óptimas locales (las aristas seleccionadas en cada paso). En este contexto, donde se busca minimizar el costo total de las calles que conectan todas las plazas sin repetir ninguna, el algoritmo greedy es efectivo y garantiza encontrar la solución óptima.

c) Ejemplo de uso

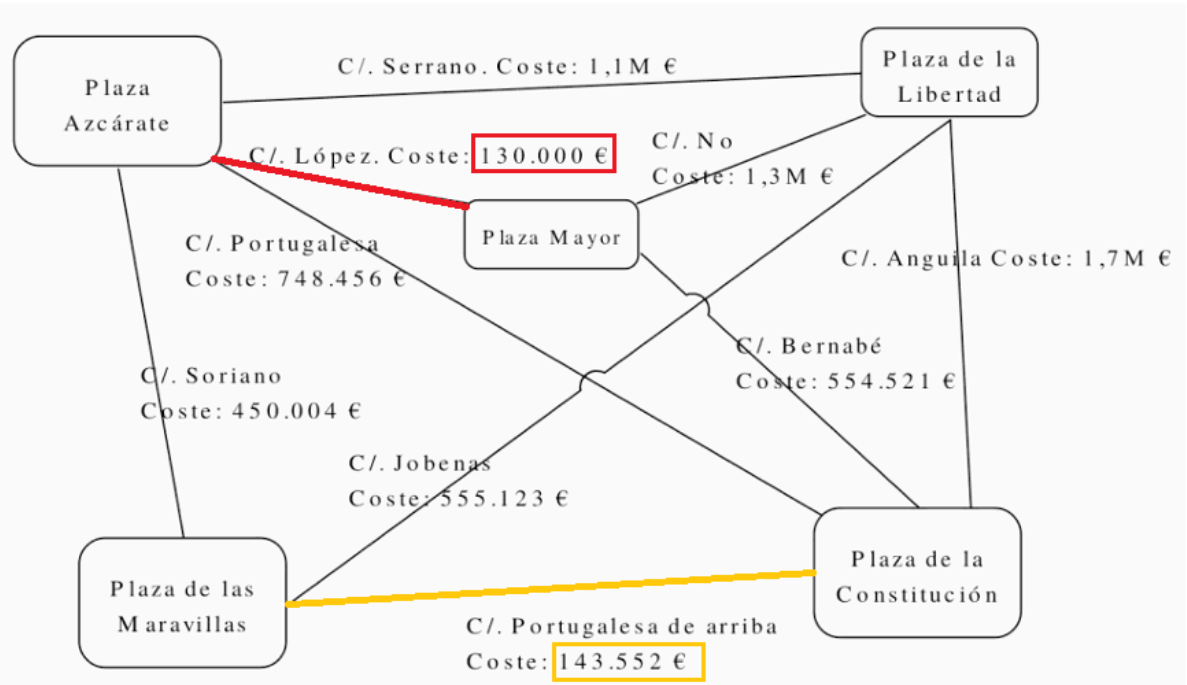
Para el ejemplo de Algovilla del Tuerto, tenemos 5 plazas, cada una con sus conexiones a otras plazas del pueblo. Cada calle que conecta 2 plazas tiene su propio precio. La lista de candidatos sería todas las calles que hay en el mapa, las cuales iremos seleccionando según el criterio de selección, que será seleccionar la más corta.



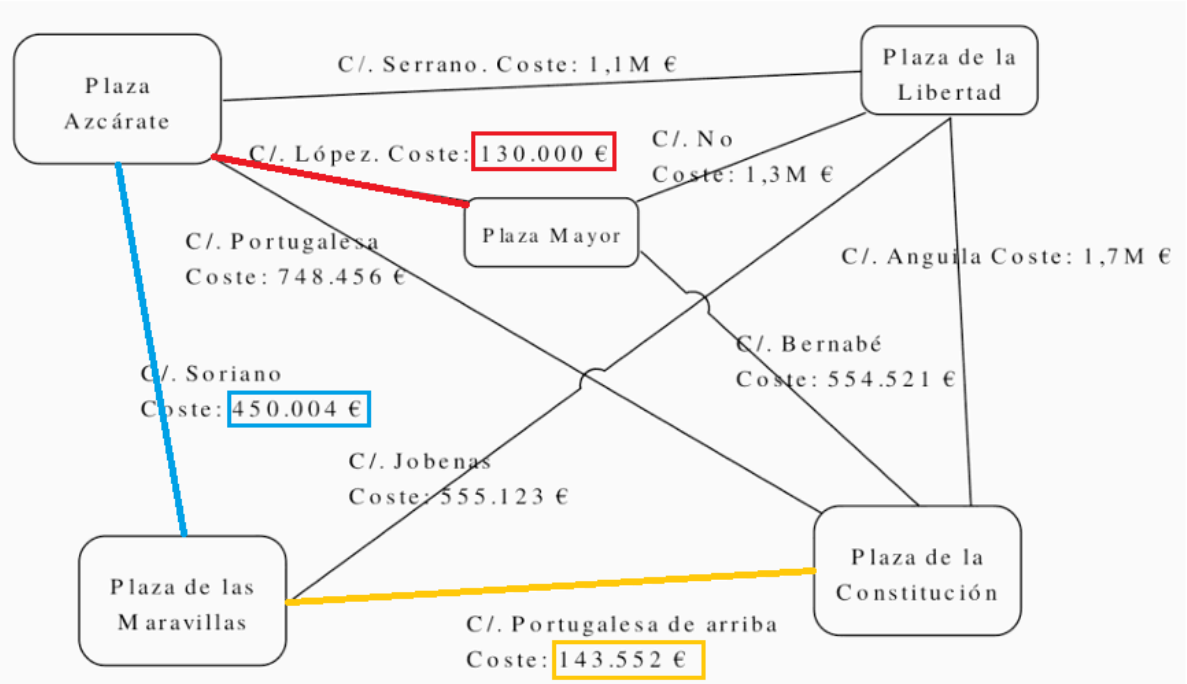
En la primera iteración, seleccionaremos la calle López, con un valor de 130.000€, que es la calle más barata. Luego comprobará con el criterio de factibilidad si se puede incluir en el conjunto solución, que al ser la primera calle, no hará ciclos, así que se mete en el conjunto solución.



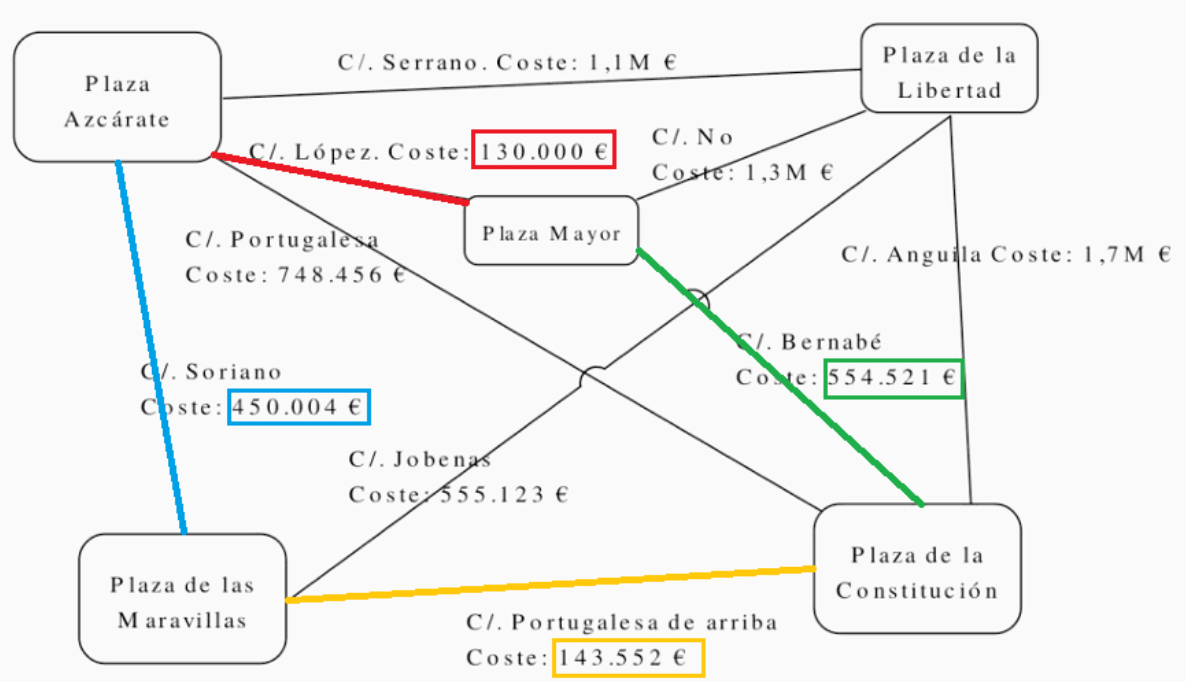
En la siguiente iteración, buscará la arista con un precio más barato, que será la calle portugalesa de arriba con 143.552€. Comprobará nuevamente si hace ciclos, y como no hace, lo añade al conjunto solución.



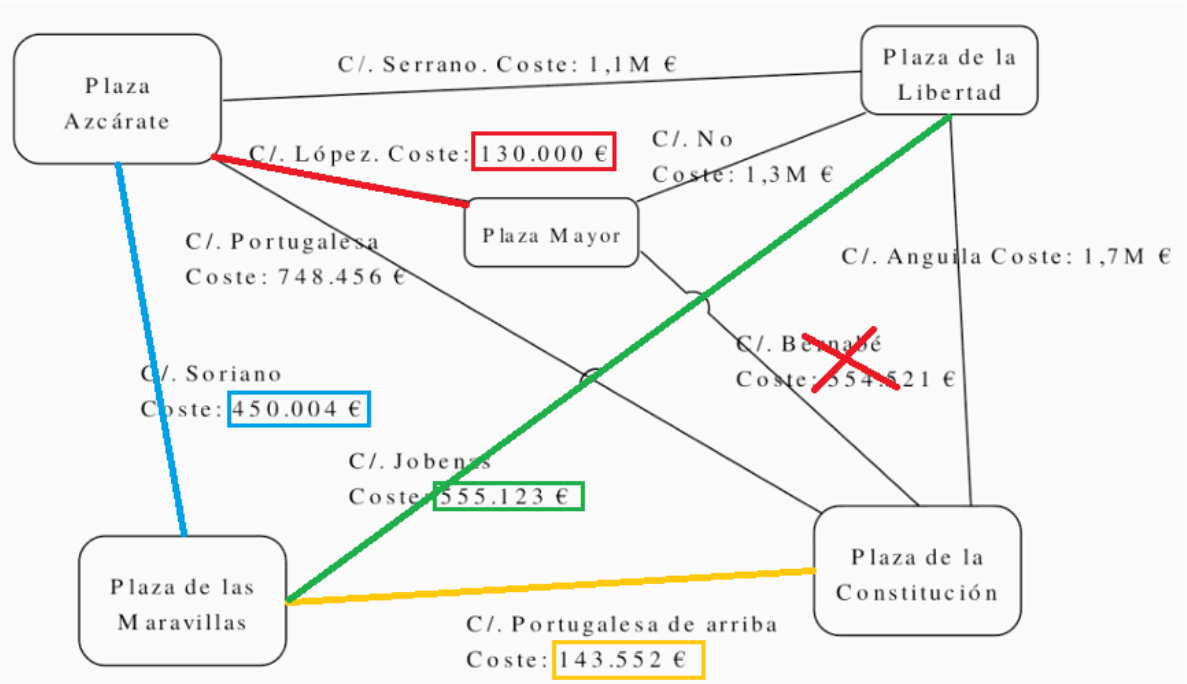
Luego seleccionará la calle Soriano, de 450.004€, que tampoco hace ciclos y añadirá a la solución.



La siguiente elección será la calle Bernabé, de 554.521€, y luego al pasar a la función de factibilidad, no podrá añadirse al conjunto solución, porque crearía un ciclo.



Entonces se sigue con la siguiente iteración que sería la calle Jobenas, de 555.123€, y al llegar a la función de factibilidad podemos aceptar la arista en el conjunto solución.



Aquí llegamos a la condición de parada, que es que el conjunto solución tenga n-1 calles, siendo n el número de plazas del pueblo. Eso es porque solo necesitamos n-1 calles para conectar todas las plazas, y en nuestro caso, necesitamos 4 para conectar 5 plazas. El precio total de todas las calles sería de 1.278.679€, que en este caso es la solución óptima porque son los precios más baratos disponibles entre todas las opciones, y con las calles suficientes para conectar todas las plazas.