Universidad de Granada

ALGORÍTMICA

Práctica 3

Práctica 3 - Algoritmos Voraces

INTEGRANTES
GRUPO B3

Miguel Rodríguez Ayllón
Alberto Cámara Ortiz
Cristóbal Pérez Simón
José Vera Castillo
Pablo Rejón Camacho

ÍNDICE

Preámbulo

Nuestro proyecto cuenta con un makefile para que resulte más fácil a la hora de ejecutar los códigos proporcionados.

Cada ejercicio cuenta con una regla específica que muestra la ejecución del algoritmo y se muestra el resultado de la ejecución.

- make ej1 *(./Ej1.bin)*
- maje ej2 *(./Ej2.bin)*
- make ej3 *(./Ej3.bin 1000 20 150)*
- make ej4 *(./Ej4.bin)*
- make ej5 *(./Ej5.bin)*

Si es necesario probar con unos valores a elegir por el profesor, recomendamos modificar la regla de ese ejercicio, o cambiar el argumento de la ejecución a mano. De todas formas, también podría ejecutarse a mano con los argumentos necesarios ya que no eliminamos los archivos .bin.

Todos los ejercicios cuentan con 1 argumento excepto el tercer ejercicio, que recibe el kilometraje de la ruta, número de gasolineras del trayecto y los kilómetros disponibles que puede recorrer el autobús sin repostar.

Problema 1

1.1. Diseño de componentes

Detallamos los elementos necesarios para diseñar el algoritmo voraz.

- **Lista de candidatos:** Todos los estudiantes de la clase.
- **Lista de candidatos seleccionados:** Las parejas formadas por los estudiantes
- **Lista de candidatos descartados:** Aquellos cuyas parejas ya han sido previamente seleccionadas.
- **Función solución:** Suma de los valores de los emparejamientos.
- **Función de factibilidad:** Un estudiante es elegido dependiendo de si está o no emparejado y con el valor más alto entre los posibles candidatos.
- **Función de selección:** Valor más alto entre los estudiantes no emparejados aún.
- **Función objetivo:** Maximizar la conveniencia entre los estudiantes

1.2. Diseño del algoritmo

El diseño del algoritmo que resuelve el problema queda descrito en el siguiente pseudocódigo.

<u>Formalización</u>

- Sean **C** los elementos posibles (los estudiantes de la clase), **D** los estudiantes disponibles para hacer las parejas, **N** las parejas hechas y **P** la matriz con los valores de emparejamiento de los estudiantes.
- Se debe devolver **N**

```
function Mejor_emparejamiento( P)

      C ← {0..n-1} // Estudiantes de la clase
      D ← {true}   // Estudiantes disponibles para emparejar
      N ← {0}      // Parejas hechas

      while N < C do

            estudianteLibre ← Seleccionar(C) si está libre
            mult ← {0} // Variable donde acumular la multiplicación
            estudiante_pareja ← Estudiante != estudianteLibre que será
      su pareja

            for all i ∈ C y D[estudianteLibre] do
                  // Obviamos que un estudiante no puede ir consigo
      mismo

                  actual              =             P[estudianteLibre][i]
      *P[i][estudianteLibre];
```

```
            // Comprobamos que el  estudiante con el que queremos
      emparejar esté libre
            if actual > mult and D[i] then
                  mult = actual // Actualizamos el valor de mult
                  // actualizamos la pareja del estudiante
                  estudiante_pareja = i
            end if
      end for

      N += 2 // aumentamos el número de emparejados

      // Actualizamos los valores de los ya emparejados a false
      D[estudianteLibre] = false
      D[estudiante_pareja] = false
end while

end function
```

1.3 Estudio de la optimalidad (demostración/contraejemplo)

Para el estudio de la optimalidad vamos a buscar un contraejemplo en una matriz 4*4 donde se verá perfectamente que hay una manera más óptima de buscar:

{ 0, 7, 8, 5 }
{ 3, 0, 4, 1 }
{ 6, 3, 0, 2 }
{ 9, 4, 7, 0 }

La solución encontrada es:

Estudiante 1 con Estudiante 3 (8 * 6 = 48)
Estudiante 2 con Estudiante 4 (1 * 4 = 4)
Suma de los valores de la matriz: 52

Hemos podido encontrar una solución más óptima a la encontrada por el algoritmo greedy en este ejemplo:

Encontramos que si el 1 se empareja con el 4, daría 45 (que es menor que 48) el 2 se emparejaría con el 3 siendo así la suma = 12. Siendo la suma total 57.

Estudiante 1 -> Estudiante 4 (9 * 5 = 45)
Estudiante 2 -> Estudiante 3 (4 * 3 = 12)
Suma de los valores de la matriz: 57

Al haber encontrado una solución más óptima podemos llegar a la conclusión de que este algoritmo no es óptimo.

1.4 Funcionamiento del algoritmo para una instancia pequeña

Definimos la siguiente instancia para explicar el funcionamiento del algoritmo:

Pudiendo coger el ejemplo del anterior apartado:

{	**0,**	**7,**	**8,**	**5** **}**
{	**3,**	**0,**	**4,**	**1** **}**
{	**6,**	**3,**	**0,**	**2** **}**
{	**9,**	**4,**	**7,**	**0** **}**

Al principio , el algoritmo elige cuál es el estudiante libre , que sería el estudiante número 1.
Tras esto, empieza el for para saber con cuál de los otros compañeros de clase tiene un valor de emparejamiento mayor , en este caso con el estudiante 3 , ya que 8*6 = 48 , es mayor a 7*3 = 21 y a 9*5 = 45 , tras esto el número de emparejados aumenta a 2 ya que tenemos la pareja del 1 con el 3 y en el vector de booleanos pongo que el estudiante 1 y el tres ya no están disponibles.

Como el número de emparejados sigue siendo menor a los estudiantes totales (en este caso 4), el bucle hace otra iteración.
Al seleccionar el siguiente estudiante libre no elije el 1 ya que este ya ha sido previamente seleccionado y opta por el estudiante 2. Una vez hecho esto entra en el for y empieza a comprobar. Al comprobar sólo hace las comprobaciones del 2 con el 4 , debido a que el 1 y el 3 ya no están disponibles.
Después, aumenta el número de emparejados de dos en dos por lo que ahora serían 4, y los componentes del vector de bools correspondientes a los estudiantes 2 y 4 a false ya que ya están emparejados.

Como el número de emparejados y de estudiantes totales es igual, el while acaba y se acaba la función devolviendo las parejas creados en un vector pair y devueltas con un return.

1.5 Implementación

La función que nos permite llevar a cabo la solución del problema es la siguiente:

```
vector < pair < int,int > > Mejor_emparejamiento(vector<vector<int>> &preferencias) {

    vector < pair < int,int > > emparejamiento;

    int n = preferencias.size();
    vector<bool> disponible(n, true);
    int numEmparejados = 0;



    while (numEmparejados < n) {
      int estudianteLibre = 0;
      // Compruebo cuál es el siguiente almumno sin pareja

      while (estudianteLibre < n && !disponible[estudianteLibre]){
        estudianteLibre++;
      }


      int mult = 0;
      int estudiante_eleg;


      for (int primer_estudiante = 0;
           primer_estudiante < n && disponible[estudianteLibre];
           ++primer_estudiante) {
        //Un alumno no puede ir consigo mismo
        if (primer_estudiante == estudianteLibre) {
        }


        int actual = preferencias[primer_estudiante][estudianteLibre] *
                     preferencias[estudianteLibre][primer_estudiante];
        // Si la multiplicacion es mayor al del anterior y esta disponible , se actualiza la pareja
        if ( actual > mult && disponible[primer_estudiante]){
          mult = actual;
          estudiante_eleg = primer_estudiante;
        }
      }

      // Hago la pareja
      numEmparejados+=2;

      //Meto la pareja en el vector de emparejados
      emparejamiento.push_back(make_pair(estudianteLibre,estudiante_eleg ));

      // Hago que no esten disponibles los estudiantes ya emparejados
      disponible[estudiante_eleg] = false;
      disponible[estudianteLibre] = false;
    }
    return emparejamiento;
}
```

Problema 2

2.1. Diseño de componentes

Detallamos los elementos necesarios para diseñar el algoritmo voraz.

- **Lista de candidatos:** El número de invitados que se sentarán en la mesa circular.
- **Lista de candidatos seleccionados:** Los candidatos que ya han sido seleccionados para sentarse en un sitio específico.
- **Lista de candidatos descartados:** Aquellos que ya han obtenido un sitio en la mesa no podrán volver a ser elegidos en otro lugar.
- **Función solución:** Una solución en la que todos los comensales se encuentren sentados en la mesa
- **Función de factibilidad:** Un comensal es elegido para sentarlo a la derecha si el valor de conveniencia es máximo entre los candidatos sin sitio en la mesa.
- **Función de selección:** Valor de conveniencia más alto entre los candidatos sin asignar aún en la mesa.
- **Función objetivo:** Maximizar el valor de conveniencia de los invitados dependiendo del lugar en el que se sentarán los comensales.

2.2. Diseño del algoritmo

Formalización

- Sea **C** la lista de candidatos, y **L** la lista de invitados.

```
function asignarInvitados(n, conveniencia V[n][n]):
        Candidatos C ← ∅
        Invitados L ← ∅

        for i = 0 until n then
                C ← C ∪ i
                i ← i + 1
        end for
        for i = 0 until n && C ≠ ∅ then
                if L ← ∅ then
                        mejorPosicion ← Seleccionar(i, C, V)
                        L ← L ∪ mejorPosicion
                        C ← C \ mejorPosicion
                        L ← L ∪ i
                        C ← C \ i
                        mejorPosicion ← Seleccionar(i, C, V)
                        L ← L ∪ mejorPosicion
                        C ← C \ mejorPosicion
                        i ← i + 1
                else then
                        mejorPosicion ← Seleccionar(i, C, V)
                        L ← L ∪ mejorPosicion
                        C ← C \ mejorPosicion
                end if
                i ← i + 1
        end for
end function

function Seleccionar(i, C, V[n][n])
        valores ← ∅
        for C.begin (c) until |C| then
                if i ≠ c then
                        conveniencia total ← V[i][c] + V[c][i]
                        if conveniencia total > valores.conveniencia then
                                valores.posicion ← c
                                valores.conveniencia ← conveniencia total
                        end if
                end if
                c ← c + 1
        end for
end function
```

2.3 Estudio de la optimalidad (demostración/contraejemplo)

Para el estudio de la optimalidad, utilizando una matriz de datos 5x5, se ha encontrado un caso concreto que nos indica que nuestro algoritmo no es óptimo ya que hemos encontrado un ejemplo en el que el resultado de conveniencia máxima supera al valor de conveniencia indicado por el algoritmo y el orden de comensales es diferente.

El ejemplo de matriz escogido es el mismo que el explicado en el apartado posterior, por lo que recomendamos echar un vistazo al funcionamiento del algoritmo con la instancia utilizada en el apartado 2.4.

Tras la ejecución del algoritmo con los valores de datos indicados, el algoritmo devuelve una solución en la que los comensales serían sentados en las siguientes posiciones:

{3, 0, 1, 2, 4}

Relación (3,0)	108
Relación (0,1)	99
Relación (1,2)	106
Relación (2,4)	165
Relación (4,3)	95
Suma final	**572**

Tras probar, a mano, con diferentes posibles soluciones que mejorarían la conveniencia máxima que nos indica nuestro algoritmo, hemos encontrado finalmente una solución cuyo resultado de conveniencia máxima es mayor y vamos a realizar un contraejemplo con los siguiente valores:

{0, 3, 2, 4, 1}

Vamos a comprobar la conveniencia de esta distribución de comensales sumando las conveniencias:

Relación (0,3)	107
Relación (3,2)	133
Relación (2,4)	165
Relación (4,1)	83
Relación (1,0)	99
Suma final	**587**

Tras este contraejemplo, podemos concluir que nuestro algoritmo no es óptimo, ya que se puede llegar a una solución que mejora las solución dada como resultado de ejecutar el algoritmo.

2.4 Funcionamiento del algoritmo para una instancia pequeña

{	**0,**	**85,**	**12,**	**32,**	**18** **}**
{	**14,**	**0,**	**32,**	**54,**	**73** **}**
{	**42,**	**74,**	**0,**	**34,**	**84** **}**
{	**75,**	**21,**	**99,**	**0,**	**73** **}**
{	**32,**	**10,**	**81,**	**22,**	**0** **}**

Nuestro algoritmo tiene predeterminado el vector “invitados” vacío, por tanto entraría al primer if y buscaría los vecinos más convenientes, es decir, con mayor conveniencia mutua con respecto al invitado número 0. Para ello, hace uso de la función “mejorPosicion”, función que recorre todos los valores de la primera columna y, finalmente, guarda en el struct los valores de “mejor_conveniencia” y “mejor_posición”.
Tras la ejecución de la función, el struct queda con los siguientes valores:

- *mejor_conveniencia = 32 + 75 = 107 Posiciones (0,3) (3,0)*
- *mejor_posicion = 3.*

Insertamos el valor 3 y lo borramos del set que contiene la lista de comensales disponibles y reiniciamos a 0 los valores de la matriz que hemos utilizado, en este caso [0,3] y [3,0]. Ya tenemos el primer comensal sentado en la mesa, en este caso el 3, que ha sido eliminado de los comensales disponibles. Una vez situado en la mesa al comensal a la izquierda, vamos a sentar al comensal 0 y lo eliminaremos de los comensales disponibles, por lo que tendríamos el siguiente vector:

- *Comensales disponibles → {1, 2, 4}.*
- *Invitados → {3, 0}.*

A partir de ahora, vamos a ir sentando a los invitados a la derecha de los comensales ya sentados con una función de selección que será la mayor conveniencia mutua entre ambos.

El último valor añadido al vector invitados ha sido el 0, por lo que buscaremos en su fila al comensal con mayor conveniencia mutua, lo añadiremos a la lista de invitados y lo eliminaremos del set comensales disponibles.
En este caso:

- *mejor_conveniencia = 85 + 14 = 99. Posiciones (0,1) (1,0)*
- *mejor_posicion = 1.*
- *Comensales disponibles → {2, 4}.*
- *Invitados → {3, 0, 1}.*

A partir de ahora, entra en juego el else, que es repetir el método utilizado anteriormente y buscar el mejor comensal a su derecha (mayor conveniencia mutua), añadirlo a la lista de invitados y eliminarlo del set de comensales disponibles.

- *mejor_conveniencia = 32 + 74 = 106. Posiciones (1,2) (2,1)*
- *mejor_posicion = 2.*
- *Comensales disponibles → {4}.*
- *Invitados → {3, 0, 1, 2}*

Finalmente, sólo nos queda un candidato como comensal disponible, por lo tanto se añadirá al vector invitados independientemente del valor ya que no lo comprobará con ninguno más:

- *Invitados → {3, 0, 1, 2, 4}*
- *Suma total de la conveniencia = 107+99+106+165+95 = **572***

2.5 Implementación

La función que nos permite llevar a cabo la solución del problema es la siguiente:

```cpp
struct MejorPosicion {
    int mejor_conveniencia = -1;
    int mejor_posicion = -1;
};

MejorPosicion mejorPosicion(int i, set<int> &comensales_disponibles, int conveniencia[5][5]) {
    MejorPosicion valores;
    for(auto comensal : comensales_disponibles) {
        if(i != comensal) {
            int conveniencia_total = conveniencia[i][comensal] + conveniencia[comensal][i];
            if(conveniencia_total > valores.mejor_conveniencia) {
                valores.mejor_posicion = comensal;
                valores.mejor_conveniencia = conveniencia_total;
            }
        }
    }

    return valores;
}
```

```cpp
void asignarInvitados(int n, int conveniencia[5][5]) {
    set<int> comensales_disponibles;
    vector<int> invitados;

    for(int i = 0; i < n; i++)
        comensales_disponibles.insert(i);

    for(int i = 0; i < n && !comensales_disponibles.empty(); i++) {
        if(invitados.empty()) {
            // Colocamos al comensal izquierda:
            MejorPosicion valores = mejorPosicion(i, comensales_disponibles, conveniencia);

            conveniencia[i][valores.mejor_posicion] = 0;
            conveniencia[valores.mejor_posicion][i] = 0;
            invitados.push_back(valores.mejor_posicion);
            comensales_disponibles.erase(valores.mejor_posicion);

            invitados.push_back(i);
            comensales_disponibles.erase(i);

            // Colocamos al comensal derecha:
            valores = mejorPosicion(i,comensales_disponibles, conveniencia);
            conveniencia[i][valores.mejor_posicion] = 0;
            conveniencia[valores.mejor_posicion][i] = 0;
            invitados.push_back(valores.mejor_posicion);
            comensales_disponibles.erase(valores.mejor_posicion);
            i++;
        } else {
            // Colocamos al mejor comensal a su derecha
            MejorPosicion valores = mejorPosicion(invitados[i], comensales_disponibles, conveniencia);
            conveniencia[invitados[i]][valores.mejor_posicion] = 0;
            conveniencia[valores.mejor_posicion][invitados[i]] = 0;
            invitados.push_back(valores.mejor_posicion);
            comensales_disponibles.erase(valores.mejor_posicion);
        }
    }
    for(auto invitado : invitados)
        cout << invitado << " ";
}
```

Problema 3

3.1. Diseño de componentes

Detallamos los elementos necesarios para diseñar el algoritmo voraz.

- **Lista de candidatos:** Las gasolineras en las que se puede repostar
- **Lista de candidatos seleccionados:** Las gasolineras que han sido seleccionadas para que el número de paradas sea mínimo.
- **Lista de candidatos descartados:** Aquellas gasolineras que no computan para el cálculo de paradas.
- **Función solución:** El número de gasolineras exacto y el punto kilométrico en el que se encuentra cada.
- **Función de factibilidad:** Una gasolinera es elegida si se puede repostar en ella antes de que el autobús se quede sin gasolina.
- **Función de selección:** La gasolinera que se encuentra en el punto kilométrico más alejado de manera que se pueda repostar en ella sin quedarse sin combustible antes.
- **Función objetivo:** Minimizar el número de paradas que componen la solución del problema.

3.2. Diseño del algoritmo

El diseño del algoritmo que resuelve el problema queda descrito en el siguiente pseudocódigo.

Formalización

- Sea **C** las gasolineras del trayecto, **S** las gasolineras en las que el autobús va a parar, **s** el punto kilométrico en el que se encuentran, y **k** los kilómetros restantes.
- Se debe devolver **S**.

```
function trazarRuta(k, C)
     S ← ∅

     recorremos C
          x ← Seleccionar(C) si (C[i+1].s - C[i].s) > k
          C ← C \ {x}
          if EsFactible(S ∪ {x}) then
               S ← S ∪ {x}
          end if
     end while
     if EsSolution(S) then
          return S
     else
          return «No hay solución»
     end if
end function
```

3.3. Estudio de optimalidad (demostración/contraejemplo)

Consideraciones
Un conjunto de gasolineras es prometedor si se puede extender para obtener una solución óptima.
El conjunto vacío es prometedor.
Una gasolinera “i” está antes que una gasolinera “j” si el punto kilométrico de “i” es menor que el punto kilométrico de “j” y estar después es lo contrario.

Lema

Dicho eso, sea B un subconjunto de gasolineras, Siendo T un conjunto de gasolineras prometedoras, tal que todas las gasolineras de T están antes que las gasolineras de B. La unión del conjunto T con la gasolinera más lejana de B a la que el depósito permite llegar es un conjunto prometedor.

Inducción
Sea A el conjunto de gasolineras por las que el autobús ha pasado, T un conjunto prometedor inmediatamente antes de añadir una gasolinera “g”.
A divide el conjunto de gasolineras de la ruta en dos, si partimos de una gasolinera “u”, esta se encontrará en una parte y “g” en otra.

Sea B el conjunto que contiene a “g”, el conjunto B no contiene a “u”.
Todas las gasolineras de T están antes que las de B y “g” es la más lejana a la que puede llegar el autobús sin repostar (las anteriores se han descartado o incluido).
Entonces se cumple el lema y T ∪ “g” es prometedor,

3.4. Funcionamiento del algoritmo para una instancia pequeña

Definimos la siguiente instancia para explicar el funcionamiento del algoritmo:

- La ruta del viaje será de 500 km
- Existen 3 gasolineras
    - La primera en el punto kilométrico 100
    - La segunda en el punto kilométrico 250
    - La tercera en el punto kilométrico 400
- El autobús puede recorrer 250 km sin parar a repostar

Solucion:

El autobús comenzará desde el kilómetro 0 y no parará a repostar en la primera gasolinera puesto que esa se encuentra en los primeros 100km y por tanto, puede seguir circulando sin tener que parar.
Una vez pase los 250km, el autobús se habrá quedado sin combustible, por tanto, parará en la segunda gasolinera y al repostar podrá recorrer otra vez otros 250km.
Finalmente, no necesitará parar en la tercera gasolinera porque la distancia entre estas dos últimas es menor que 250 y habrá hecho el viaje realizando solo una parada.

3.5. Implementación

La función que nos permite llevar a cabo la solución del problema es la siguiente:

```
pair<set<int>,bool> trazarRuta(int kms_restantes, set<int> gasolineras){

    set<int> paradas; // Creamos el vector de paradas a realizar
    bool haySolucion = true;

    // Guardamos los kilómetros que podemos recorrer sin repostar
    int distancia_posible = kms_restantes;
    set<int>::iterator it = gasolineras.begin();


    // Reducimos el kilometraje restante (kms desde el inicio hasta la primera gasolinera)
    kms_restantes -= (*it);

    for(;it != prev(gasolineras.end())&& haySolucion; it++){
        //Si no podemos realizar el siguiente tramo sin parar a repostar
        if(*(next(it)) - (*it) > kms_restantes){

            // Repostamos y añadimos la gasolinera actual al vector de paradas
            paradas.insert(*it);

            // Reiniciamos el kilometraje
            kms_restantes = distancia_posible;

            if(kms_restantes < *(next(it)) - (*it)){
                haySolucion = false;
            }
        }

        // Restamos al kilometraje restante la cantidad de kilómetros recorridos
        kms_restantes -= *(next(it)) - (*it);
    }

    pair<set<int>,bool> resultado(paradas, haySolucion);
    return resultado; // Devolvemos las pradas realizadas
```

Donde “Gasolinera” es un set en el que se almacena el punto kilométrico de cada gasolinera. A la función se le pasan los kilómetros que puede recorrer el autobús sin parar a repostar y las gasolineras encontradas en la ruta.

Esta itera sobre todas las gasolineras del viaje de modo que si la diferencia entre el punto kilométrico de la siguiente gasolinera y la gasolinera actual es mayor que la distancia que puede recorrer el autobús, este para a repostar en la gasolinera actual y se restablece la distancia a recorrer (por eso se almacena el kilometraje actual en una variable auxiliar).

Si no se puede llegar a la siguiente gasolinera, se establece la variable boolean a falso indicando que no hay solución.

Restamos el kilometraje ya recorrido los puntos kilométricos de las dos gasolineras anteriores y finalmente, la función devuelve el número de paradas realizadas y el valor de la variable booleana que servirá para indicar en el main si hay solución o no.

Problema 4

4.1. Diseño de componentes

Detallamos los elementos necesarios para diseñar el algoritmo voraz.

- **Lista de candidatos:** Todos los sensores disponibles para realizar la transferencia
- **Lista de candidatos seleccionados:** Los sensores que se consideren para realizar la transferencia.
- **Lista de candidatos descartados:** Sensores que no se consideran porque otros candidatos tienen un valor menor.
- **Función solución:** Una sub-solución será la solución final al problema cuando se haya llegado al destino
- **Función de factibilidad:** El nodo seleccionado no se ha seleccionado anteriormente y no es un nodo con coste de transferencia máximo
- **Función de selección:** Seleccionar el sensor candidato con el menor tiempo de entre todos los que se consideran en ese momento.
- **Función objetivo:** Minimizar el tiempo de transferencia de los sensores del sistema independientemente de la longitud del camino.

4.2. Diseño del algoritmo

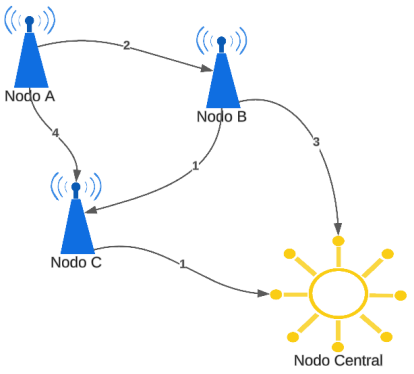
El diseño del algoritmo que resuelve el problema queda descrito en el siguiente pseudocódigo:
Formalización

- C = matriz de adyacencia con costes de transferencia
- D = sensor destino
- O = sensor origen
- S = vector camino(seleccionados)
- A = nodo actual
- coste = coste acumulado del camino
- haySolucion = booleano que indica si se ha escogido un camino por el que no se puede llegar al nodo destino.
- MAX = valor máximo posible para un double
- seleccion = nodo seleccionado de entre todos los que se consideran a partir del actual
- mínimos= vector de valores mínimos

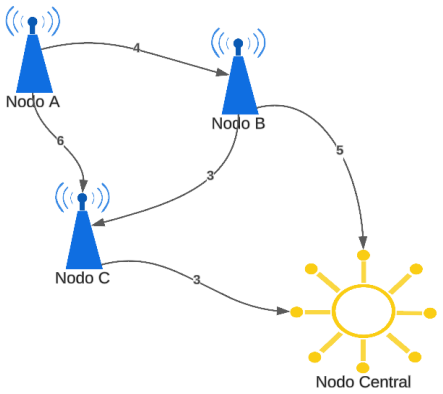
```
Algoritmo Greedy (C[n][n], O, D, &coste)
	S ← ∅
	A ← O
	minimos ← ∅

	while A != D  y haySolucion:
		minimo ← MAX
		seleccion ← 0

		for i in C[actual]
			if valor < minimo
				minimo ← C[actual][i]
				minimos ← ∅
				minimos[] ← C[actual][i]
				seleccion ← i

			if valor = minimo
				minimos[] ← C[actual][i]

			end if
		+end for

		//Considerar cuando hay más de un valor mínimo
		media_min ← MAX

		for i in mínimos
			suma ← 0
			contador ← 0
			for j in C[i]
				suma += C[i][j]
				contador++
			end for
			media ← suma/contador

			if (media <= media_min)
				media_min ← media
				seleccion ← i
		end for


		//Asegurar que no se repitan nodos
		C[actual][seleccion] ← MAX
		C[seleccion][actual] ← MAX
		S ← S ∪ seleccion
		coste += minimo
		A ← seleccion

		if minimo == MAX
		//De todas las posibilidades se ha escogido un nodo por donde no se
		podía pasar por lo que no hay solución para esa matriz con Greedy

			haySolucion = false
			vaciar camino
			coste acumulado = 0
		end if
	end while
return C
```

4.3 Estudio de la optimalidad (demostración/contraejemplo)

Para estudiar la optimalidad de este algoritmo lo haremos por reducción al absurdo, es decir, supondremos en primer lugar que el algoritmo si es óptimo y trataremos de llegar a una contradicción. Vamos a analizar lo que ocurriría en este caso:



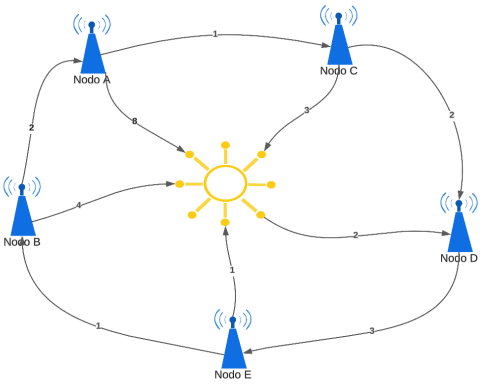
Si partimos del nodo A, el algoritmo Greedy escogería en primer lugar el nodo B y tendría un coste acumulado de 2. Después, el algoritmo pudiendo escoger el camino que le lleva directamente al nodo Central escoge el de menor coste de los dos que es el que le lleva al nodo C. Una vez en el nodo C y con un coste acumulado de 3 no le queda más remedio que escoger el camino que le lleva al nodo central con un coste de 1 por lo que el coste total es de 4. Si miramos los costes de los caminos que el algoritmo ha escogido es fácil ver que aunque son más cortos, son más costosos. Sin embargo, ¿qué pasa si sumamos un coste de dos a todos los caminos?



En este caso, el algoritmo haría el mismo recorrido pero esta vez, coger un camino más largo le está penalizando más y hay otros caminos menos costosos como el que va de A -> B -> Central o A-> C -> Central. Por tanto, hemos llegado a una solución en la que hay otra solución óptima que no es la escogida por el algoritmo por lo que podemos determinar que no se trata de un algoritmo óptimo.

4.4 Funcionamiento del algoritmo para una instancia pequeña

Definimos la siguiente instancia para explicar el funcionamiento del algoritmo:

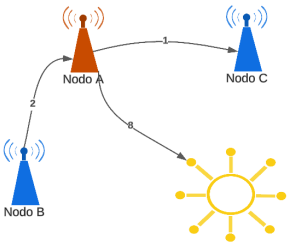


Esta instancia del problema presenta la siguiente matriz de adyacencias:

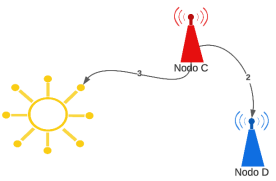
	A	B	C	D	E	F
A		2	1			8
B	2		3		1	4
C	1	3		2		3
D			2		3	2
E		1		3		1
F	8	4	3	2	1	

Partimos del nodo A y el objetivo es llegar al nodo central, en este caso, el nodo F minimizando el tiempo de transferencia entre nodos.

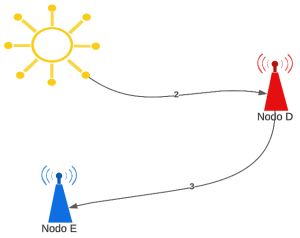
En el primer paso, el algoritmo tiene tres nodos entre los que elegir:



Y de estos tres, escoge ir al nodo C ya que es el camino de menor coste. Una vez en C, se le plantean dos opciones: el nodo D o ir directamente al nodo central.



De nuevo, escoge el de menor coste de los dos que en este caso es el D.



Aquí escoge el de menor coste que es que le lleva directamente al nodo central por lo que habría terminado con un coste acumulado de 5, habiendo otros caminos con menor coste como el de A->C->F con un coste de 4.

4.5 Implementación

Este problema lo modelamos como una matriz de adyacencia en la que las filas y columnas representan a los nodos y cada valor dentro de la matriz representa el tiempo de transferencia entre los nodos de esa fila y columna. Para representar que no sea posible transferir datos de un nodo a otro se han utilizado valores máximos. Para la resolución del problema se asume que siempre habrá un camino viable desde el nodo origen al nodo destino independientemente de su coste y que no se puede pasar por un nodo más de una vez. La función que nos permite llevar a cabo la solución del problema es la siguiente:

```
vector<int> Greedy(vector<vector<double>>& valores, int source, int destino, double& coste_acumulado) {
   int actual = source;
   vector<int> camino;
   coste_acumulado = 0;
   bool haySolucion = true;

   while (actual != destino && haySolucion) {
      double minimo = M;
      vector<int> minimos;
      int seleccion = 0;

      // Encuentra el mínimo y los índices iguales al mínimo
      for (int i = 0; i < valores[actual].size(); i++) {
         if (valores[actual][i] < minimo) {
            minimo = valores[actual][i];
            minimos.clear();
            minimos.push_back(i);
         }else if (valores[actual][i] == minimo) {
            minimos.push_back(i);
         }
      }


      double media_min = M;
      for (int i : minimos) {
         double suma = 0;
         int contador = 0;

         for (int j = 0; j < valores[i].size(); j++) {
            if (valores[i][j] != M) {
               suma += valores[i][j];
               contador++;
            }
         }
         double media = suma / contador;
         if (media <= media_min) {
            media_min = media;
            seleccion = i;
         }
      }

      // Aseguramos que no haya repetidos
      valores[actual][seleccion] = M;
      valores[seleccion][actual] = M;
      camino.push_back(seleccion);
      coste_acumulado += minimo;
      actual = seleccion;

      // Si el mínimo es M, no hay solución
      if (minimo == M) {
         haySolucion = false;
         cout << "\nNo existe una solución Greedy para la matriz de adyacencia\n";
         camino.clear();
         coste_acumulado = 0;
      }
   }

   return camino;
}
```

Problema 5

5.1. Diseño de componentes

Detallamos los elementos necesarios para diseñar el algoritmo voraz.

- **Lista de candidatos:** las calles que componen la ciudad.
- **Lista de candidatos seleccionados:** las calles que se han seleccionado para asfaltar.
- **Función solución:** las calles seleccionadas unen todas las plazas, se asfaltan n-1 calles (siendo n el número de plazas).
- **Función de factibilidad:** el conjunto de calles seleccionadas no forma ciclos
- **Función de selección:** se selecciona la calle con menos coste de asfaltado.
- **Función objetivo:** minimizar la suma del coste de asfaltar las calles que forman la solución.

5.2. Diseño del algoritmo

El diseño del algoritmo que resuelve el problema queda descrito en el siguiente pseudocódigo.

Formalización
resultado = conjunto de calles que se asfaltan
G = mapa de la ciudad
V = conjunto de plazas de la ciudad
A = coste de asfaltar las calles
compu = conjunto al que pertenece u
compv = conjunto al que pertenece v

```
function Asfaltar(G = (V , A))
      ordenar A de forma creciente
      n = n° de plazas
      resultado = ∅
      while |resultado| < n - 1 hacer
            calle = A[0]
            A = A \ {calle}
            u, v = calle.vertices
            compu = Buscar(u)
            compv = Buscar(v)
            if compu != compv then
                  Unir(compu, compu)
                  resultado U {calle}
            end if
      end while
      return resultado
end function
```

5.3 Estudio de la optimalidad (demostración/contraejemplo)

Consideraciones
Un conjunto de calles es prometedor si se puede extender para obtener una solución óptima.
El conjunto vacío es prometedor.
Si un conjunto ya es solución la extensión es innecesaria.
Una calle sale de un conjunto de plazas si tiene un extremo en ese conjunto.

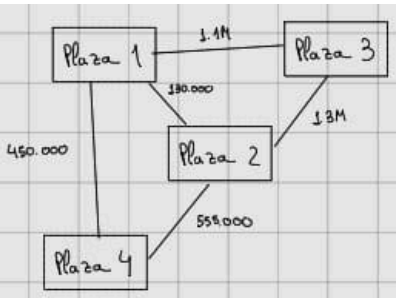
Lema
Dicho eso, sea B un subconjunto de las plazas de la ciudad, Siendo T un conjunto de calles prometedor, tal que ninguna calle de T sale de B. La unión del conjunto T con la calle que menos cuesta asfaltar que sale de B es un conjunto prometedor.

Inducción
Supongamos que T es prometedor antes de añadir una nueva calle c. Las calles de T dividen la ciudad en dos o más componentes. La plaza u está en una de esas componentes y la v está en otra. Llamamos B al conjunto que contiene a u.

El conjunto B no incluye a v
T es un conjunto prometedor, tal que ninguna calle de T sale de B.
La calle c es la de coste mínimo (el resto ya se ha descartado o incluido)

5.4 Funcionamiento del algoritmo para una instancia pequeña

Definimos la siguiente instancia para explicar el funcionamiento del algoritmo:



Paso	Calle	¿Sirve?	Componentes	Costes	Resultado
Ini			{1}, {2}, {3}, {4}	130k, 450k, 555k, 1.1M, 1.3M	

Tomamos el coste más barato, que une las calles 1 y 2. Como están en componentes diferentes es una calle valida, así que las juntamos y guardamos el coste

1	(1,2)	Si	{1,2}, {3}, {4}	450k, 555k, 1.1M, 1.3M	130k

Tomamos el coste más barato, que une las calles 1 y 4. Como están en componentes diferentes es una calle valida así que las juntamos y guardamos el coste

2	(1,4)	Si	{1,2,4}, {3}	555k, 1.1M, 1.3M	130k, 450k

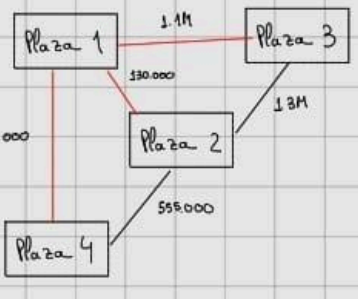
Tomamos el coste más barato, que une las calles 4 y 2. Como están en la misma componente, es innecesario asfaltar.

3	(4,2)	NO	{1,2,4}, {3}	1.1M, 1.3M	130k, 450k

Tomamos el coste más barato, que une las calles 1 y 3. Como están en componentes diferentes es una calle valida así que las juntamos y guardamos el coste

4	(1,3)	Si	{1,2,3,4}	1.3M	130k, 450k, 1.1M

Ya hemos asfaltado n-1 calles así que podemos parar



5.5 Implementación

La función que nos permite llevar a cabo la solución del problema es la siguiente:

```
Codiumate: Options | Test this function
vector<int> asfaltar(const Pueblo &p,  map<string,int> &presupuestos) {
  int n = p.getNumPlazas();
  vector<int> resultado;
  set<set<int>> componentes;

  //Generar las componentes conexas
  for (int i = 0; i < n; i++){···

  while (resultado.size() < n - 1) {
    int coste_calle = presupuestos.begin()->second;
    string nombre_calle = presupuestos.begin()->first;
    presupuestos.erase(nombre_calle);

    //Sacar las plazas conectadas por esa calle
    vector<int> conectadas  = p.getPlazasNombreCalle(nombre_calle);
    int u = conectadas[0], v = conectadas[1];

    //Sacar las componentes de las plazas
    set<int> compu;
    set<int> compv;
    for (auto it = componentes.begin(); it != componentes.end(); ++it)
      if (it->find(u) != it->end()) compu = *it;


    for (auto it = componentes.begin(); it != componentes.end(); ++it)
      if (it->find(v) != it->end()) compv = *it;

    //Comprobar si son distintas
    if (compu != compv){
      //Unir esas componentes
      componentes.erase(compu);
      componentes.erase(compv);
      for (int ele : compv)
        compu.insert(ele);
      componentes.insert(compu);

      //Añadir el precio al resultado
      resultado.push_back(coste_calle);
    }
  }

  return resultado;
}
```