PRÁCTICA 3

ALGORITMOS GREEDY

UNIVERSIDAD
DE GRANADA

Marino Fernández Pérez,
Mehdi Iabouten,
Jorge López Molina,
Gabriele Ruggeri
y Pablo Vega Romero

EQUIPOS FORMADOS POR PAREJAS (problema 1)

```cpp
// Función para encontrar el emparejamiento óptimo
vector<pair<int, int>> HacerParejas(vector<vector<int>>& preferencias) {
    int num_estudiantes = preferencias.size();
    vector<bool> emparejados(num_estudiantes, false);
    vector<pair<int, int>> parejas_finales;

    for (int k = 0; k < num_estudiantes / 2; ++k) {
        int max_emparejamiento = -1;
        pair<int, int> mejor_pareja;

        // Buscar la mejor pareja no emparejada
        for (int i = 0; i < num_estudiantes; ++i) {
            if (!emparejados[i]) {
                for (int j = i + 1; j < num_estudiantes; ++j) {
                    if (!emparejados[j]) {
                        int valor = preferencias[i][j] * preferencias[j][i];
                        if (valor > max_emparejamiento) {
                            max_emparejamiento = valor;
                            mejor_pareja = {i, j};
                        }
                    }
                }
            }
        }

        // Emparejar y marcar como emparejados
        emparejados[mejor_pareja.first] = true;
        emparejados[mejor_pareja.second] = true;
        parejas_finales.push_back(mejor_pareja);
    }

    return parejas_finales;
}
```

Diseño de Componentes

→ **Lista de candidatos:** Número de estudiantes de una clase.
→ **Lista de candidatos usados:** Estudiantes que ya están emparejados.
→ **Criterio de selección:** Seleccionar aquellos estudiantes que maximicen la suma de los valores de los emparejamientos.
→ **Criterio de factibilidad:** Que un estudiante no esté emparejado.
→ **Función solución:** La disposición final de las parejas.
→ **Función objetivo:** La solución de este problema es obtener las mejores parejas tras asignar a todos un compañero.

Diseño del algoritmo en base a la plantilla Greedy

En la inicialización, se crean variables como num_estudiantes para determinar el total de estudiantes, emparejados para rastrear las asignaciones de parejas, y parejas_finales para almacenar los emparejamientos finales. El bucle principal itera hasta que se forman num_estudiantes / 2 parejas, ya que al formar parejas solo se deben recorrer la mitad de los estudiantes. En la selección de la mejor pareja, se exploran todas las combinaciones de estudiantes no emparejados, calculando el "valor" del emparejamiento como el producto de sus preferencias mutuas y seleccionando la pareja con el mayor valor. Una vez encontrada la mejor pareja, se marcan como emparejados y se agregan a la lista parejas_finales. Al finalizar el bucle, la función devuelve parejas_finales, que contiene todas las parejas formadas.

```
function HacerParejas(preferencias):
    "num_estudiantes" es el número de estudiantes
    emparejados <- {falso, ....., falso}  // array de tamaño num_estudiantes, inicializado con falsos
    "parejas_finales"  // array vacío

    para k desde 0 hasta num_estudiantes / 2:
        max_emparejamiento = -1
        mejor_pareja = par de enteros

        para i desde 0 hasta num_estudiantes:
            si no está emparejado[i]:
                para j desde i + 1 hasta num_estudiantes:
                    si no está emparejado[j]:
                        valor = preferencias[i][j] * preferencias[j][i]
                        si valor > max_emparejamiento:
                            max_emparejamiento = valor
                            mejor_pareja = {i, j}
                        end si
                    end si
                end for
            end si
        end for

        marcar emparejados[mejor_pareja.first] como verdadero
        marcar emparejados[mejor_pareja.second] como verdadero
        agregar mejor_pareja a parejas_finales

    end for

    devolver parejas_finales
end function
```

Estudio de optimalidad

Para demostrar si el algoritmo es óptimo para todos los casos o no vamos a utilizar un contraejemplo, es decir, buscar un ejemplo que demuestre que el algoritmo no es óptimo para todos los casos. Consideremos que tenemos 4 estudiantes con las siguientes preferencias dadas en la matriz preferencias:

```
int main() {
    int n = 4; // Número de estudiantes
    vector<vector<int>> preferencias = {
        {0,1,3,1},
        {5,0,10,7},
        {5,1,0,8},
        {12,3,4,0}
    };

    vector<pair<int, int>> pairs = HacerParejas(preferencias);

    for (auto &pair : pairs) {
        cout << "Estudiante " << pair.first << " emparejado con Estudiante " << pair.second << endl;
    }

    return 0;
}
```

Si ejecutamos el algoritmo en este ejemplo, escogerá la pareja con mayor valor de preferencia mutuo, que en este caso será la pareja (2,3) que tiene un valor de preferencia mutuo igual a 24, lo que nos deja la pareja (0,1), la cual tiene un valor de preferencia mutuo igual a 5, siendo la suma de preferencias totales mutuas igual a 29.

Sin embargo existen soluciones más óptimas en este caso, como por ejemplo, si se selecciona la pareja (1,3) la cual tiene un valor de preferencia mutuo de 21, menor que el de la pareja (2,3), pero al emparejar los estudiantes (0,2) tenemos que su valor de preferencia mutuo es de 15, por lo que la suma de preferencias totales mutuas es igual a 36, mayor que el caso que escogería el algoritmo.

Con este ejemplo se demuestra que el algoritmo no escoge la solución óptima en todos los casos.

Funcionamiento del algoritmo

A la hora de analizar el funcionamiento de nuestro algoritmo, hemos escogido un ejemplo de uso donde el número de estudiantes es 4 y los niveles de preferencia para la matriz son pequeños con el objetivo de poder observar su funcionamiento sencillamente.

```cpp
int main() {
    int n = 4; // Número de estudiantes
    vector<vector<int>> preferencias = {
        {0,1,3,1},
        {5,0,10,7},
        {5,1,0,8},
        {12,3,4,0}
    };

    vector<pair<int, int>> pairs = HacerParejas(preferencias);

    for (auto &pair : pairs) {
        cout << "Estudiante " << pair.first << " emparejado con Estudiante " << pair.second << endl;
    }

    return 0;
}
```

Primer paso:

- num_estudiantes = 4
- emparejados = {false, false, false, false}
- parejas_finales = {}

Segundo paso:

Por cada iteración, seleccionamos la pareja que maximice la suma de los valores de los emparejamientos. Para ello, deberemos iterar sobre un segundo bucle, en este caso, un bucle interno, que compruebe cuál es la mejor pareja que se puede formar.

Para la primera posición (k = 0), se busca la pareja que maximice la suma de los valores de los emparejamientos.

Estudiante 0 (i=0):

- Con el estudiante 1 (j=1): valor = p(0, 1) * p(1, 0) = 5
- Con el estudiante 2 (j=2): valor = p(0, 2) * p(2, 0) = 15
- Con el estudiante 3 (j=3): valor = p(0, 3)* p(3, 0) = 12

Estudiante 1 (i=1):

- Con el estudiante 2 (j=2): valor = p(1, 2) * p(2, 1) = 10
- Con el estudiante 3 (j=3): valor = p(1, 3)* p(3, 1) = 21

Estudiante 2 (i=2):

- Con el estudiante 3 (j=3): valor = p(2, 3) * p(3, 2) = 24

  La pareja {2,3} maximiza el valor de preferencias, por lo que se coloca en la primera posición y se marca como emparejado.

  emparejados[false, false, true, true]

Para la segunda posición y última (k = 1), se busca la pareja que maximice la suma de los valores de los emparejamientos con los estudiantes 2 y 3 ya emparejados, por lo tanto nos queda solo una pareja que sería:

Estudiante 0 (i=0):

- Con el estudiante 1 (j=1): valor = p(0, 1) * p(1, 0) = 5

  La pareja {0,1} maximiza el valor de preferencias, por lo que se coloca en la segunda posición y se marca como emparejado.

  emparejados[true, true, true, true]

La disposición final es: {2,3}, {0,1}.

MESA CIRCULAR Y CONVENIENCIAS (PROBLEMA 2)

```cpp
vector<int> disposicionMaxGreedy(const vector<vector<int>>& conveniencia, int num_invitados) {
    vector<int> mejor_disposicion(num_invitados, -1);
    vector<bool> sentado(num_invitados, false);

    mejor_disposicion[0] = 0;
    sentado[0] = true;

    for (int i = 1; i < num_invitados; i++) {
        int mejor_invitado = -1;
        int mejor_conveniencia = -1;

        for (int j = 0; j < num_invitados; j++) {
            if (!sentado[j]) {
                int izda = mejor_disposicion[(i - 1 + num_invitados) % num_invitados];
                int dcha = mejor_disposicion[(i + 1) % num_invitados];
                int conveniencia_actual;

                if (dcha == -1){
                    conveniencia_actual = conveniencia[izda][j];
                }
                else{
                    conveniencia_actual = conveniencia[izda][j] + conveniencia[j][dcha];
                }

                if (conveniencia_actual > mejor_conveniencia) {
                    mejor_invitado = j;
                    mejor_conveniencia = conveniencia_actual;
                }
            }
        }

        mejor_disposicion[i] = mejor_invitado;
        sentado[mejor_invitado] = true;
    }

    return mejor_disposicion;
}
```

```cpp
int calcularConvenienciaTotal(const vector<vector<int>>& conveniencia, const vector<int>&
disposicion){
    int nivel_total = 0;
    int num_invitados = disposicion.size();

    // suma de los niveles de conveniencia de cada invitado con los dos invitados sentados a su
    lado
    for (int i = 0; i < num_invitados; i++){
        // invitado 9, num_invitados = 10
        // el invitado de su derecha es el invitado 0
        // 9 + 1 % 10 = 0

        int actual = disposicion[i];
        int dcha = disposicion[(i + 1) % num_invitados];
        int izda = disposicion[(i - 1 + num_invitados) % num_invitados];

        nivel_total += conveniencia[actual][izda] + conveniencia[actual][dcha];
        cout << nivel_total << endl;
    }

    return nivel_total;
}
```

Diseño de Componentes

→ **Lista de candidatos:** Invitados que pueden ser asignados a los asientos de la mesa
→ **Lista de candidatos usados:** Invitados que ya han sido asignados a un asiento en la mesa
→ **Criterio de selección:** Seleccionar aquel invitado que maximice la suma de los niveles de conveniencia con los invitados ya sentados a su izquierda y derecha
→ **Criterio de factibilidad:** Que un invitado no sea asignado a un asiento ya ocupado
→ **Función solución:** La disposición final de los invitados en la mesa
→ **Función objetivo:** La solución de este problema es obtener el máximo nivel de conveniencia tras asignar a todos y cada uno de los invitados a un asiento de la mesa circular.

Diseño del algoritmo en base a la plantilla Greedy

Para resolver el problema, el algoritmo greedy implementado parte de una disposición inicial en la cual ya se encuentra asignado uno de los invitados (resto de la mesa vacía) y que posteriormente, para cada una de las posiciones vacías en la mesa, selecciona al invitado correcto basándose en el nivel de conveniencia que proporciona al estar sentado junto a otros invitados ya colocados en la mesa, en concreto , el de la izquierda y el de la derecha.

```
function encontrarDisposicionMaxima(conveniencia, num_invitados)
   "conveniencia" es una matriz de enteros donde cada posición (i, j) representa el nivel de conveniencia entre el invitado i y el invitado j

   "num_invitados" es el número de invitados

   mejor_disposicion <- {-1, ..... ,-1} // array de enteros con tamaño num_invitados

   sentado <- {falso, ..... ,falso} // array booleano con tamaño num_invitados

   colocamos al invitado x en mejor_disposicion
   almacenamos que el invitado x está sentado con verdadero

   Para cada una de las posiciones en la mesa que están libres (i):
      mejor_invitado = valor negativo
      mejor_conveniencia = valor negativo

      Para cada uno de los invitados (j):
         si el invitado actual no está sentado entonces:
            izda = invitado que se encuentra a la izquierda en el vector mejor_disposicion
            dcha = invitado que se encuentra a la derecha en el vector mejor_disposicion

            si no hay nadie sentado a la derecha del invitado actual entonces:
               conveniencia_actual = conveniencia entre el invitado actual y el invitado sentado a su izquierda
            si hay:
               conveniencia_actual = suma de conveniencia entre el invitado actual y los invitados que están a su izquierda y derecha
            end si

            si conveniencia_actual es mayor a mejor_conveniencia entonces:
               mejor_invitado = invitado actual (j)
               mejor_conveniencia = conveniencia_actual
            end si
         end si
      end for

      mejor_disposicion(i) = mejor_invitado
      sentado(mejor_invitado) = verdadero
   end for

   devolvemos mejor_disposicion
end function
```

```
function calcularConvenienciaTotal(conveniencia, disposicion)
   "conveniencia" es una matriz de enteros donde cada posición (i, j) representa el nivel de conveniencia entre el invitado i y el invitado j

   "disposicion" es un vector que representa la disposición final de invitados

   nivel total = 0
   numero de invitados = tamaño de la disposicion obtenida

   Para cada uno de los invitados (i) :
      actual = invitado actual
      dcha = invitado que se encuentra a la derecha en el vector mejor_disposicion
      izda = invitado que se encuentra a la izquierda en el vector mejor_disposicion

      nivel total += suma de conveniencia entre el invitado actual y los invitados que están a su izquierda y derecha
   end for

   devolvemos nivel total
end function
```

Estudio de optimalidad

A la hora de evaluar la optimalidad de nuestro algoritmo que resuelve el problema, podemos observar que no es un algoritmo óptimo. Para demostrarlo, evaluaremos un contraejemplo donde el algoritmo no devuelve la solución óptima.

Como sabemos, es muy importante tener en cuenta que la disposición final que obtenemos para nuestro algoritmo dependerá del primer invitado, que lo sentamos en el primer asiento de nuestra mesa. Dicho esto, pueden existir disposiciones para una matriz de conveniencia de mayor nivel de conveniencia, y por lo tanto, más óptimas, si colocamos un invitado distinto en la primera posición de la mesa o bien, si trabajamos con matrices de gran tamaño.

Supongamos que tenemos una situación con 10 invitados.

```
int num_invitados = 10;
vector<vector<int>> conveniencia = {
 {0, 50, 10, 30, 70, 20, 40, 90, 80, 60},
 {50, 0, 20, 40, 30, 60, 70, 80, 90, 10},
 {10, 20, 0, 50, 60, 70, 80, 90, 40, 30},
 {30, 40, 50, 0, 20, 80, 90, 70, 60, 10},
 {70, 30, 60, 20, 0, 10, 40, 50, 90, 80},
 {20, 60, 70, 80, 10, 0, 50, 30, 40, 90},
 {40, 70, 80, 90, 40, 50, 0, 60, 70, 20},
 {90, 80, 90, 70, 50, 30, 60, 0, 10, 40},
 {80, 90, 40, 60, 90, 40, 70, 10, 0, 50},
 {60, 10, 30, 10, 80, 90, 20, 40, 50, 0}
};
```

Utilizando nuestro algoritmo voraz que hemos implementado y asignando a la mesa como primer invitado al invitado 2, obtenemos la siguiente disposición

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

El nivel de conveniencia obtenido en este caso es 1640.

Para la comprobación de la mejor disposición que podríamos obtener y que, por lo tanto, sería la óptima, hemos hecho uso de un algoritmo basado en realizar todas y cada una de las permutaciones posibles para encontrar disposición con el nivel de conveniencia máximo:

```
vector<int> disposicionMaxPermutaciones(const vector<vector<int>>& conveniencia) {
    int num_invitados = conveniencia.size();
    vector<int> disposicion(num_invitados, -1);

    for (int i = 0; i < num_invitados; ++i) {
        disposicion[i] = i;
    }

    int mejor_nivel_conveniencia_total = 0;
    vector<int> mejor_disposicion;

    do {
        int nivel_conveniencia_total = calcularNivelConvenienciaTotal(conveniencia, disposicion);
        if (nivel_conveniencia_total > mejor_nivel_conveniencia_total) {
            mejor_nivel_conveniencia_total = nivel_conveniencia_total;
            mejor_disposicion = disposicion;
        }
    } while (next_permutation(disposicion.begin(), disposicion.end()));

    return mejor_disposicion;
}
```

Luego, utilizando el algoritmo no voraz (fuerza bruta), obtenemos la siguiente disposición

$$[0, 1, 8, 4, 9, 5, 3, 6, 2, 7]$$

El nivel de conveniencia obtenido en este caso es 1660, que tras verificar las operaciones, es el valor correcto.

Cabe destacar que, para esta matriz contraejemplo, seleccionar al invitado 0 como primer invitado sentado en la mesa también nos permite obtener, en este caso, la mejor disposición posible sin tener que hacer uso de un algoritmo de Fuerza Bruta.

Finalmente, podemos concluir que este contraejemplo muestra que el algoritmo greedy no siempre produce la solución óptima, ya que toma decisiones basadas en la información local en cada paso y puede perder la oportunidad de alcanzar una solución óptima. En este caso, demostramos que la elección del primer invitado sentado en la mesa puede influir en la calidad de la solución encontrada por el algoritmo voraz. Al fijar el primer invitado como el invitado 2, el algoritmo voraz puede perder la oportunidad de encontrar una solución óptima en ciertos casos.

Funcionamiento del algoritmo

A la hora de analizar el funcionamiento de nuestro algoritmo, hemos escogido un ejemplo de uso donde el número de invitados es 4, el primer invitado seleccionado es el 0 y los niveles de conveniencia para la matriz son pequeños con el objetivo de poder observar su funcionamiento sencillamente.

```
118  int main() {
119
120      // matriz ejemplo
121      int num_invitados = 4;
122      vector<vector<int>> conveniencia = {
123          {0, 5, 1, 10},
124          {5, 0, 1, 1},
125          {1, 1, 0, 5},
126          {10, 1, 5, 0}
127      };
128
129      // matriz contrajemplo
130      // int num_invitados = 10;
131      // vector<vector<int>> conveniencia = {
132      //  {0, 50, 10, 30, 70, 20, 40, 90, 80, 60},
133      //  {50, 0, 20, 40, 30, 60, 70, 80, 90, 10},
134      //  {10, 20, 0, 50, 60, 70, 80, 90, 40, 30},
135      //  {30, 40, 50, 0, 20, 80, 90, 70, 60, 10},
136      //  {70, 30, 60, 20, 0, 10, 40, 50, 90, 80},
137      //  {20, 60, 70, 80, 10, 0, 50, 30, 40, 90},
138      //  {40, 70, 80, 90, 40, 50, 0, 60, 70, 20},
139      //  {90, 80, 90, 70, 50, 30, 60, 0, 10, 40},
140      //  {80, 90, 40, 60, 90, 40, 70, 10, 0, 50},
141      //  {60, 10, 30, 10, 80, 90, 20, 40, 50, 0}
142      // };
143
144
145      vector<int> disposicion = disposicionMaxGreedy(conveniencia, num_invitados);
146      int nivel_conveniencia_total = calcularConvenienciaTotal(conveniencia, disposicion);
147
```

Primer paso:
Seleccionamos al invitado 0 como primer candidato para ser sentado en la mesa y lo colocamos en ella.

- Invitado inicial = 0
- mejor_disposicion[0, -1, -1, -1]
- sentados[true, false, false, false]

Segundo paso:
Para cada posición siguiente en la mesa, seleccionamos al invitado que maximice la suma de los niveles de conveniencia con los invitados ya sentados a su izquierda y derecha. Para ello, deberemos iterar sobre un segundo bucle, en este caso, un bucle interno, que compruebe la conveniencia con todos y cada uno de los invitados sin asignar.

2. Para la segunda posición (i = 1), se busca al invitado que maximice la suma de los niveles de conveniencia con los invitados ya sentados a su izquierda y derecha.
    - Para el invitado 1: conveniencia_actual = conveniencia[0][1] = 5
    - Para el invitado 2: conveniencia_actual = conveniencia[0][2] = 1
    - Para el invitado 3: conveniencia_actual = conveniencia[0][3] = 10
      El invitado 3 maximiza la conveniencia, por lo que se coloca en la segunda posición y se marca como sentado.
    - sentados[true, false, true, false]
    - mejor_disposicion[0, 3, -1, -1]
3. Para la tercera posición (i = 2), se busca al invitado que maximice la suma de los niveles de conveniencia con los invitados ya sentados a su izquierda (invitado 3).
    - Para el invitado 1: conveniencia_actual = conveniencia[3][1] = 1
    - Para el invitado 2: conveniencia_actual = conveniencia[3][2] = 5
      El invitado 2 maximiza la conveniencia, por lo que se coloca en la segunda posición y se marca como sentado.
    - sentados[true, true, true, false]
    - mejor_disposicion[0, 3, 2, -1]
4. Para la cuarta y última posición (i = 3), solo queda el invitado 1 sin sentar. Se coloca al invitado 1 en la cuarta posición.
    - Para el invitado restante: conveniencia_actual = conveniencia[2][1] + conveniencia[1][0] = 1 + 5 = 6
    - sentados[true, true, true, true]
    - mejor_disposicion[0, 3, 2, 1]
5. La disposición final es: 0 3 2 1.

Tercer paso:

Finalmente , llamamos en el main a la función “calcularConvenienciaTotal”, que va a recibir como parámetro tanto la matriz de conveniencia como el vector obtenido tras llamar a la implementación mencionada anteriormente y va a obtener como resultado el valor 42.

El funcionamiento interno de esta función es el siguiente paso a paso:

- Para el invitado 0: conveniencia con el invitado a su derecha (3) y a su izquierda (1) = conveniencia[0][1] + conveniencia[0][3] = 5 + 10 = 15
- Para el invitado 3: conveniencia con el invitado a su derecha (2) y a su izquierda (0) = conveniencia[3][0] + conveniencia[3][2] = 10 + 5 = 15
- Para el invitado 2: conveniencia con el invitado a su derecha (1) y a su izquierda (3) = conveniencia[2][3] + conveniencia[2][1] = 5 + 1 = 6
- Para el invitado 1: conveniencia con el invitado a su derecha (0) y a su izquierda (2) = conveniencia[1][2] + conveniencia[1][0] = 1 + 5 = 6

Ruta de Autobús (Problema 3)

```
vector<int> gasolinerasGreedy(int kmTotalRuta, vector<int> gasolineras , int autonomiaMax){
    vector<int> solucion;
    int autonomiaAct = autonomiaMax;
    int gasCounter = 0;
    bool quedoSinAutonomia = false;

    for (int i = 0 ; i < kmTotalRuta && !quedoSinAutonomia ; i++){ //cada uno de los km de la ruta

        if (i == gasolineras[gasCounter]){ //es decir esta pasando por delante de una gasolinera

            if (gasCounter == gasolineras.size()-1){ //es es la ultima
                if (i+autonomiaAct < kmTotalRuta){ //entonces que entre
                    solucion.push_back(gasolineras[gasolineras.size()-1]);
                    autonomiaAct = autonomiaMax;

                    autonomiaAct--;
                }
            }
            else{ //en caso de que no sea la ultima
                if (i+autonomiaAct < gasolineras[gasCounter+1]){ //entonces que entre
                    solucion.push_back(gasolineras[gasCounter]);
                    autonomiaAct = autonomiaMax;

                    autonomiaAct--;
                }
            }


            gasCounter++;
        }
        else{
            autonomiaAct--;
        }

        if (autonomiaAct == 0 && i != kmTotalRuta){
            quedoSinAutonomia = true;
            cout << "quedo sin autonomia" << endl;
            solucion.clear();
        }
     }

    return solucion;
}
```

```
int main(){
    int kmTotal = 50;
    vector<int> gasolineras = {5,10,15,19,25,31,35,38,45};
    int autonomia = 13;

    cout << "las gasolineras estan en: ";

    for (int i = 0 ; i < gasolineras.size() ; i++){
        cout << gasolineras[i] << " ";
    }

    cout << endl;

    cout << "la autonomia maxima es: " << autonomia << endl;

    vector<int> dondeParara = gasolinerasGreedy(kmTotal,gasolineras,autonomia);

    if (dondeParara.empty()){
        cout << "no tiene solucion o llego sin tener que parar a repostar" << endl;
    }
    else
    {
        cout << "parara en las siguientes gasolineras: " << endl;

        for (int i = 0 ; i < dondeParara.size() ;i++){
            cout << "gasolinera :" << dondeParara[i] << endl;
        }
    }
}
```

Diseño de Componentes

→ **Lista de candidatos:** Gasolineras por las que pasa un trayecto de autobús.
→ **Lista de candidatos usados:** Gasolineras en las que es indispensable repostar.
→ **Criterio de selección:** Seleccionar aquella secuencia de gasolineras que minimicen el número de paradas de un recorrido teniendo en cuenta la autonomía del autobús.
→ **Criterio de factibilidad:** Que no se pare a repostar si se podría alcanzar la próxima gasolinera con la autonomía del depósito restante.
→ **Función solución:** Una secuencia de enteros que representa los kilómetros del trayecto en los que hay una gasolinera que deberemos visitar para repostar el depósito.
→ **Función objetivo:** La solución de este problema debe dar con la lista de gasolineras que minimiza el número de paradas de un trayecto.

Diseño del algoritmo en base a la plantilla Greedy

Para abordar este problema, el algoritmo implementado parte del kilómetro 0 de una ruta en el que se asume que tiene el depósito lleno. Además cuenta con un vector de gasolineras en las que puede parar y se asume que se podrá repostar en todas ellas si se necesita. Seleccionará una gasolinera si observa que no podríamos llegar a la próxima con la autonomía restante, de esta forma se maximizan los kilómetros por depósito y se minimizan las paradas.

```
function gasolinerasGreedy(kmTotalRuta,gasolineras,autonomiaMax)
    "kmTotalRuta" es un entero que simboliza los kilometros totales de una ruta.
    "gasolineras" es una lista de enteros que guarda el kilometro del trayecto en el que nos encontramos con una gasolinera.
    "autonomiaMax" es un entero que guarda la autonomía del autocar que realiza la ruta.

    solucion <- es un array que guardar las gasolineras en las que paremos (tiene que estar ordenado de menor a mayor)

    autonomiaAct <- es un entero que guarda la autonomía que nos queda en el kilómetro en el que nos encontramos

    gasCounter <- entero que funciona como contador de las gasolineras
                  por las que ya hemos pasado, se usa para recorrer el vector gasolineras

    quedoSinAutonomia <- es un booleano que solo será true si queda sin autonomía,
                         es decir ya no puede moverse más y por tanto dicha combinacion
                         de datos carcía de solución.

    recorremos cada uno de los kilometros de la ruta y en el que exista una gasolinera
    e interpretemos que hay que parar a respotar sera almacenado en el vector solución

    Para cada uno de los kilómetros de la ruta mientras no se quede sin autonomía (quedoSinAutonomia = true):

        Si en el kilómetro actual hay una gasolinera:

            Si se trata de la ultima gasolinera:

                Si no podemos llegar al final de la ruta con la autonomía restante:

                    Añadimos gasolinera[gasCounter] al vector solucion
                    autonomiaAct = autonomiaMax  recargamos la autonomia al maximo

                    autonomiaAct-- porque restamos la correspondiente al kilómetro donde se encuentra la gasolinera

                end
            Si no se trata de la ultima gasolinera:

                Si no podemos llegar a la siguiente con lo que nos queda de autonomia:

                    Añadimos gasolinera[gasCounter] al vector solución
                    autonomiaAct = autonomiaMax  recargamos la autonomia al maximo

                    autonomiaAct-- porque restamos la correspondiente al kilómetro donde se encuentra la gasolinera
                end

            end

            gasCounter++ porque haya repostado o no ya no volveremos a esta

        Si no se trataba de una gasolinera:
            autonomiaAct-- se resta la autonomía correspondiente a recorrer ese kilómetro y continuamos
        end

        Si nos hemos quedado sin autonomía y no hemos terminado la ruta:
            quedoSinAutonomía = true
            se informa por pantalla con un mensaje
            se vacía el vector solucion (solucion.clear())
        end
    end

    devolvemos solucion
```

Estudio de optimalidad

A la hora de evaluar la optimalidad de nuestro algoritmo que resuelve el problema, podemos observar que se trata de un algoritmo óptimo.

Como sabemos, la cualidad que determina si una solución es óptima es el número de repostajes que se van a realizar en el trayecto. Si vamos más allá llegaremos a la conclusión de que la solución óptima es aquella en la que se aprovechan más los kilómetros de los que dispone un depósito.

Nuestra implementación greedy usa esa misma condición para obtener la solución óptima. Mediante una comprobación averigua si con los kilómetros restantes de su depósito podría llegar a la próxima gasolinera sin importar lo arriesgado que pueda ser.

De esta forma se están aumentando los kilómetros realizados por depósito al máximo y es por ello que devuelven la solución óptima siempre para cualquier caso. Sin importarle el viaje completo, localmente observa hasta donde le podría llevar su depósito actual. Es por esto que se demuestra que la solución es siempre óptima. A mayor utilización de los depósitos menos paradas se van a producir.

Funcionamiento del algoritmo

Para estudiar el funcionamiento de este algoritmo hemos optado por un trayecto simple con muchas gasolineras, es decir, muchas combinaciones de paradas posibles para llegar hasta el final.

Se trata de un trayecto de 20 kilómetros y un autocar con 13 kilómetros de autonomía.
El trayecto tiene gasolineras en las posiciones {5,10,15,19}.

```
int main(){
    int kmTotal = 20;
    vector<int> gasolineras = {5,10,15,19};
    int autonomia = 13;

    cout << "las gasolineras estan en: ";

    for (int i = 0 ; i < gasolineras.size() ; i++){
        cout << gasolineras[i] << " ";
    }

    cout << endl;

    cout << "la autonomia maxima es: " << autonomia << endl;

    vector<int> dondeParara = gasolinerasGreedy(kmTotal,gasolineras,autonomia);

    if (dondeParara.empty()){
        cout << "no tiene solucion o llego sin tener que parar a repostar" << endl;
    }
```

Primer paso:

En el instante inicial tenemos la autonomía al máximo y nos disponemos a comenzar el trayecto.

kmActual = 0
autonomiaMax = 13
autonomiaAct = autonomiaMax

Segundo paso:

Comenzamos a recorrer la ruta hasta dar con una gasolinera, vamos iterando un bucle for desde 0 hasta los kilómetros de la ruta. De momento el vector solución está vacío y vamos a comprobar si en esta gasolinera debemos repostar o no.

1. para el instante en el que i == gasolinera
    - comprobamos si es la última gasolinera del trayecto (la cual debe recibir una trato especial)
2. si no es la última gasolinera
    - comprobamos si puede llegar a la siguiente con el depósito restante
    - en el caso de que no añadimos esta gasolinera a la solución y reiniciamos la autonomía

      En este caso cuando llegue a 5 verá que puede avanzar hasta 10 sin agotar su autonomía luego no respostará en 5.

      Cuando llegue a 10 verá que no puede llegar a la siguiente gasolinera que está en 15 sin repostar luego respostará en 10 y será añadido a la solución. Recargará la autonomía al máximo y se le restará uno correspondiente al kilómetro que representa la gasolinera. (suponemos que esta se encuentra al principio del kilómetro que la reprensenta)
3. si es la última gasolinera
    - comprobamos si puede llegar al final con el combustible restante
    - en el caso de que no pueda añadimos la gasolinera a la solución y reiniciamos la autonomía

      Cuando pase por la gasolinera que está en el kilómetro 19 qué es la última comprobará que puede finalizar el trayecto con el combustible restante sin problema luego continuará hasta el final sin repostar

Tercer paso:

Hay una comprobación auxiliar que consiste en verificar si la autonomía es igual a 0 y aún no ha terminado su trayecto. Esto quiere decir que se ha quedado sin gasolina y no le ha sido posible terminar luego no había solución para esos datos. En el ejemplo planteado si hay solución y es la siguiente.

```
jorge@smart-refrigerator:~/Escritorio/Algoritmica/ALGP3_2023$ ./problema3
las gasolineras estan en: 5 10 15 19
la autonomia maxima es: 13
parara en las siguientes gasolineras:
gasolinera :10
```

Red de sensores (problema 4)

El problema que queremos resolver, nos plantea obtener la máxima velocidad posible al enviar datos entre cualquier nodo de la red hasta el servidor de datos central, de modo que tengamos que obtener el camino con el menor coste de tiempo posible entre el servidor y el resto de nodos sensores.

```
49  void red_sensores(vector<Nodo> &grafo_red, int servidor, vector<caminos> &solucion){
50      int seleccionados = 1;
51      int peso;
52      int padre = servidor;
53
54      vector<Nodo> elegidos;
55
56      elegidos.push_back(grafo_red.at(servidor));
57
58      while(seleccionados < grafo_red.size()){
59          int menor = 100000;
60
61          vector<Nodo> candidatos;
62
63          for(int i = 0; i < elegidos.size(); i++){
64
65              for(int j = 0; j < (elegidos.at(i)).conexiones.size(); j++){
66
67                  if(!busca(elegidos, elegidos.at(i).conexiones.at(j))){
68
69                      int pesoServ = elegidos.at(i).dist_serv;
70                      Nodo x = grafo_red.at(elegidos.at(i).conexiones.at(j));
71                      x.dist_serv += (pesoServ + elegidos.at(i).tiempo_conex.at(j));
72
73                      if(x.dist_serv < menor){
74                          padre = elegidos.at(i).identificador;
75
76                          menor = x.dist_serv;
77                      }
78
79                      candidatos.push_back(x);
80
81                  }
82              }
83          }
84
```

```
85          int seleccionado = menorTiempo(candidatos);
86
87          seleccionados++;
88
89          grafo_red.at(candidatos.at(seleccionado).identificador).dist_serv = candidatos.at(seleccionado).dist_serv;
90
91          elegidos.push_back(grafo_red.at(candidatos.at(seleccionado).identificador));
92
93          solucion.at(candidatos.at(seleccionado).identificador) = solucion.at(padre);
94
95          (solucion.at(candidatos.at(seleccionado).identificador)).camino.push_back(candidatos.at(seleccionado).identificador);
96      }
97  }
98
```

Funciones y estructuras auxiliares utilizadas:

```
11 struct Nodo{
12     int identificador;  //Identificador del nodo
13     int dist_serv;  //Distancia total al servidor
14     vector<int> conexiones; //Nodos con los que se conecta
15     vector<int> tiempo_conex; //Tiempo que hay entre los nodos con los que se conecta.
16 };
```

Struct Nodo: estructura en la que se almacenarán los datos de cada nodo sensor (y el servidor) en la red. Cuenta con un identificador para diferenciarlos, una variable que nos indica la distancia con respecto al servidor(en este caso también nos sirve para representar el tiempo con respecto al nodo que vaya a conectarse), un vector con las conexiones que tiene (a qué nodo se puede conectar) y otro vector que nos muestra el tiempo que tarda en realizar dichas conexiones.

```
18 struct caminos{
19     vector<int> camino; //nodos por los que tiene que pasar para llegar al servidor.
20 };
```

Struct caminos: se trata de una estructura que nos sirve para representar los nodos por los que tiene que pasar la conexión de manera que tarde el menor tiempo posible.
Más adelante veremos que creamos un vector<caminos> donde se nos mostrarán todas las soluciones para todos los nodos sensores.

```
22 int menorTiempo(vector<Nodo> nodos){
23     int menor = 100000;
24     int sensor = -1;
25
26     for(int i = 0; i < nodos.size(); i++){
27         if(nodos[i].dist_serv < menor){
28             menor = nodos[i].dist_serv;
29             sensor = i;
30         }
31     }
32
33     return sensor;
34 }
```

Función menorTiempo: se trata de una función que dado un vector de nodos, nos devuelve el identificador del nodo sensor con menor tiempo dentro de dicho vector.

```
36  bool busca(vector<Nodo> nodos, int id){
37      bool encontrado = false;
38
39      for(int i = 0; i < nodos.size(); i++){
40          if(id == nodos[i].identificador){
41              encontrado = true;
42              break;
43          }
44      }
45
46      return encontrado;
47  }
```

Función busca: se trata de buscar un nodo, pasando su identificador, dentro de un vector, si lo encuentra devuelve "true" y en caso contrario devuelve "false".

Diseño de Componentes:

→ **Lista de candidatos:** Nodos sensores que son válidos para ser escogidos.

→ **Lista de candidatos usados:** Nodos sensores que ya han sido escogidos y forman parte de la solución.

→ **Criterio de selección:** Seleccionar aquel camino con el menor coste de tiempo posible hasta el servidor.

→ **Criterio de factibilidad:** El camino seleccionado debe ser el mínimo y no deben generarse ciclos.

→ **Función solución:** Recorrido que une todos los nodos (cada uno de ellos) con el servidor.

→ **Función objetivo:** Recorrido que une todos los nodos sensores con el servidor central usando el menor tiempo posible.

Diseño del algoritmo:

→En cada iteración se comprueba cuáles son los nodos que están unidos a los ya usados, calculando para cada uno de ellos el coste de llegar hasta el nodo servidor.

→Se comprueba que no se encuentran en los ya usados, de esta manera evitamos la aparición de ciclos.

→Una vez que calculamos todos los candidatos seleccionamos entre ellos el que tenga menor coste de llegar, añadiendolo a los usados (aquellos nodos sensores que ya han sido seleccionados) y añadiendo además su solución de la siguiente forma: La solución es el camino de llegar al padre mas el mismo.

```
Procedimiento red_sensores(grafo_red: vector<Nodo>, servidor: entero, solucion: vector<caminos>)
    seleccionados = 1
    padre = servidor
    elegidos = [grafo_red[servidor]]  // Inicializar la lista de nodos elegidos con el nodo servidor

    Mientras seleccionados sea menor que el tamaño de grafo_red hacer
        menor = ∞  // Inicializar el valor del peso mínimo como infinito
        candidatos = []  // Inicializar la lista de nodos candidatos

        Para cada nodo en elegidos hacer
            Para cada conexión del nodo hacer
                Si la conexión no está en la lista de nodos elegidos entonces
                    Calcular el peso sumando la distancia al servidor del nodo actual, el tiempo de conexión y la distancia al servidor del nodo seleccionado anteriormente
                    Si el peso calculado es menor que el menor entonces
                        Actualizar el padre con el identificador del nodo actual
                        Actualizar el valor del peso mínimo con el peso calculado
                    Fin Si
                    Agregar el nodo a la lista de candidatos
                Fin Si
            Fin Para
        Fin Para

        Seleccionar el nodo con el menor peso en la lista de candidatos

        Aumentar el contador de seleccionados en 1

        Actualizar la distancia al servidor del nodo seleccionado con el valor del peso mínimo

        Agregar el nodo seleccionado a la lista de nodos elegidos

        Actualizar la solución del nodo seleccionado con la solución del nodo padre

        Agregar el identificador del nodo seleccionado al camino de la solución del nodo seleccionado
    Fin Mientras
Fin Procedimiento
```

Estudio de optimalidad:

Aunque vayamos a demostrar que es óptimo, en principio ya sabemos que lo es, ya que nos hemos basado en el algoritmo de Dijskstra para realizarlo. Recordemos que con este algoritmo siempre nos aseguramos de que el peso sea mínimo y no haya ciclos. Además en la práctica siempre se llega a una solución óptima.

Como vemos el algoritmo siempre elige el camino más rápido para acceder desde los nodos sensores al servidor central, esto garantiza la optimalidad con tiempos que no sean negativos, ya que Dijkstra encuentra una solución óptima cuando los tiempos son positivos. Esto es debido a que siempre se comprueban los nodos más cercanos primero, y una vez se encuentra el camino más pequeño, avanza.

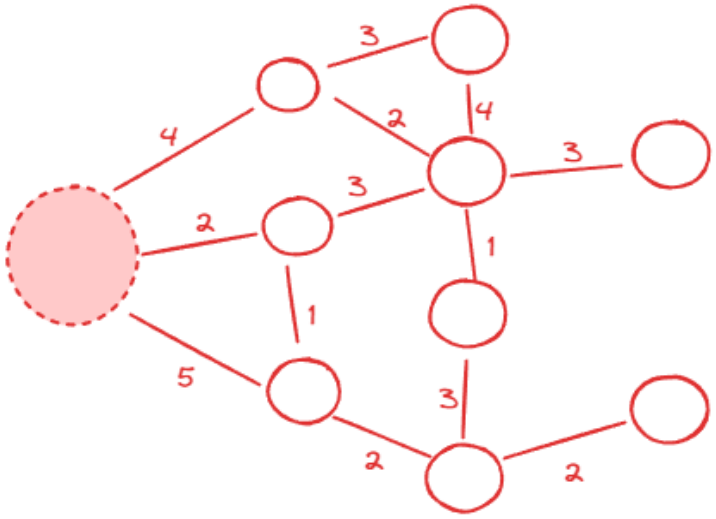
Supongamos que tenemos un conjunto de 5 nodos sensor y un nodo servidor, conectados entre sí de la siguiente manera:
Nodo sensor 0: [(1, 7), (2, 3), (3, 8), (4, 8)]
Nodo sensor 1: [(0, 7), (2, 2), (3, 8), (4, 1)]
Nodo sensor 2: [(0, 3), (1, 2), (3, 9), (4, 8)]
Nodo sensor 3: [(0, 8), (1, 8), (2, 9)]
Nodo sensor 4: [(0, 8), (1, 1), (2, 8)]

Supongamos además que el nodo servidor es el nodo sensor 0.
Los resultados serían los siguientes:
Nodo sensor 0, recorre: [0]
Nodo sensor 1, recorre: [0 1]
Nodo sensor 2, recorre: [0 2]
Nodo sensor 3, recorre: [0 2 3]
Nodo sensor 4, recorre: [0 1 4]

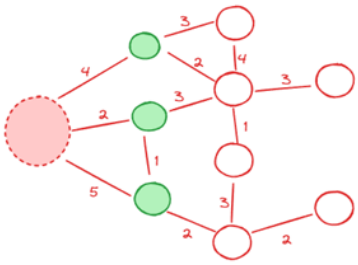
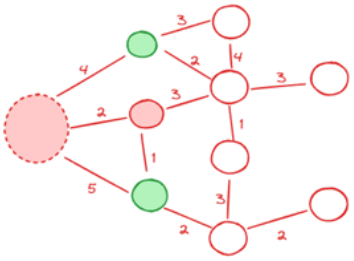
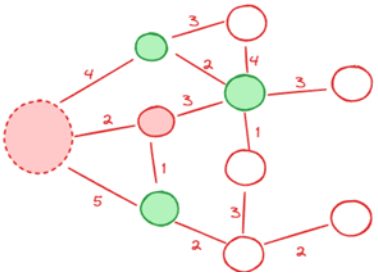
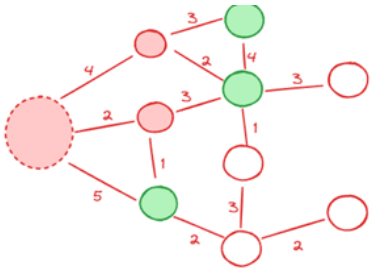
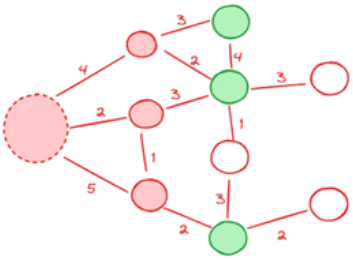
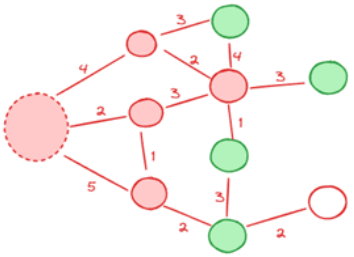
Podemos verificar manualmente que estos son los caminos más cortos desde el nodo servidor hasta cada nodo sensor. Si comparamos estos caminos con cualquier otro posible conjunto de caminos, encontraremos que estos son los óptimos en términos de distancia total recorrida.

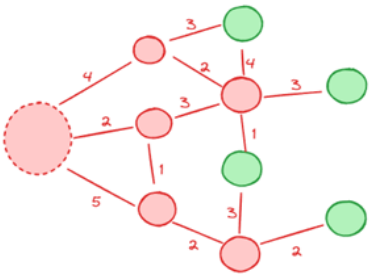
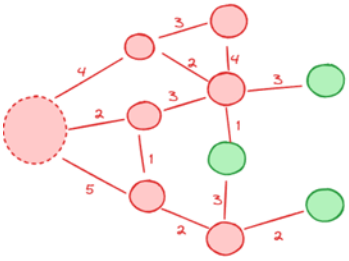
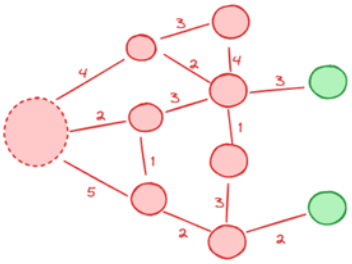
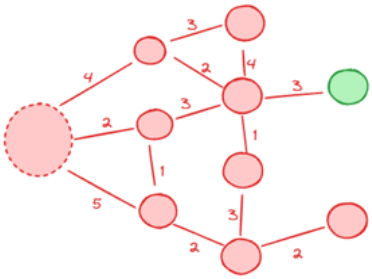
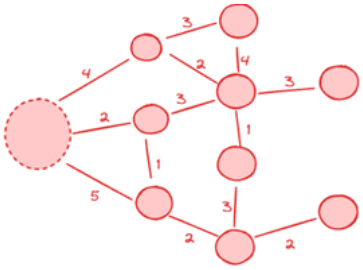
Funcionamiento del algoritmo:

A continuación mostraré cómo actúa el algoritmo utilizando un esquema para representar una red, en la que tenemos un servidor central y una serie de nodos sensores.



Aquí tenemos una demostración de cómo podría ser el grafo, a continuación veremos los pasos para realizar el algoritmo.

1. 	2. 	3. 
4. 	5. 	6. 

7. 	8. 	9. 
10. 	11. 	

A continuación también mostraremos volcados de pantalla y veremos cómo se llega a las soluciones que mostramos en dichos volcados:

```
aulas@aulas-VirtualBox:~/Escritorio/ALGP3$ ./red_sensores
0 6 2 9 8
6 0 0 7 7
2 0 0 9 8
9 7 9 0 3
8 7 8 3 0
Nodo sensor 0 :[(1,6),(2,2),(3,9),(4,8),]
Nodo sensor 1 :[(0,6),(3,7),(4,7),]
Nodo sensor 2 :[(0,2),(3,9),(4,8),]
Nodo sensor 3 :[(0,9),(1,7),(2,9),(4,3),]
Nodo sensor 4 :[(0,8),(1,7),(2,8),(3,3),]
El nodo servidor es: 1

MOSTRANDO RESULTADOS:
Nodo sensor 0, recorre: [0 1 ]
Nodo sensor 1, recorre: [1 ]
Nodo sensor 2, recorre: [2 0 1 ]
Nodo sensor 3, recorre: [3 1 ]
Nodo sensor 4, recorre: [4 1 ]
```

En este caso y como vemos, el nodo servidor es el **1**, ahora nos fijamos en la matriz y sus resultados.
La matriz, según el volcado de pantalla, quedaría de esta manera:

	0	1	2	3	4
0	0	6	2	9	8
1	6	0	0	7	7
2	2	0	0	9	8
3	9	7	9	0	3
4	8	7	8	3	0

A la hora de ver las conexiones y los tiempos de los nodos el volcado nos muestra cómo quedarían las conexiones junto a sus tiempos de conexión con respecto a los demás.

Nodo 0:

Conecta con:

-Nodo 1 y tiene peso 6.

-Nodo 2 y tiene peso 2.

-Nodo 3 y tiene peso 9.

-Nodo 4 y tiene peso 8.

Nodo 1(Servidor):

Conecta con:

-Nodo 0 y tiene peso 6.

-Nodo 3 y tiene peso 7.

-Nodo 4 y tiene peso 7.

Nodo 2:

Conecta con:

-Nodo 0 y tiene peso 6.

-Nodo 3 y tiene peso 7.

-Nodo 4 y tiene peso 7.

Nodo 3:

Conecta con:

-Nodo 0 y tiene peso 9.

-Nodo 1 y tiene peso 7.

-Nodo 2 y tiene peso 9.

-Nodo 4 y tiene peso 3.

Nodo 4:

Conecta con:

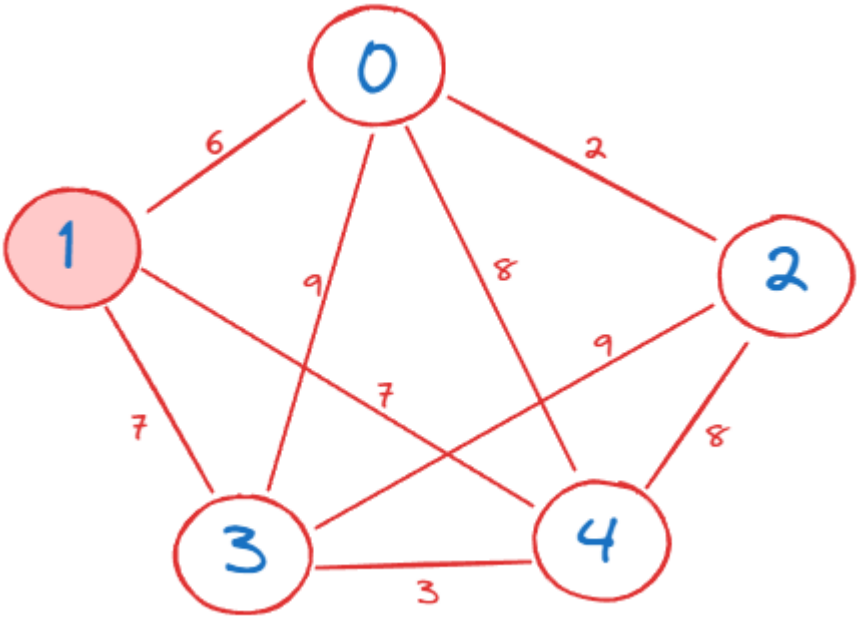
-Nodo 0 y tiene peso 8.

-Nodo 1 y tiene peso 7.

-Nodo 2 y tiene peso 8.

-Nodo 3 y tiene peso 3.

Nuestro grafo representado como dibujo sería el siguiente:



Una vez visto cómo sería el grafo representado en el dibujo podemos de ver de forma más clara cómo serían los recorridos que necesita hacer cada nodo sensor para que el tiempo de conexión sea el más pequeño posible.

```
MOSTRANDO RESULTADOS:
Nodo sensor 0, recorre: [0 1 ]
Nodo sensor 1, recorre: [1 ]
Nodo sensor 2, recorre: [2 0 1 ]
Nodo sensor 3, recorre: [3 1 ]
Nodo sensor 4, recorre: [4 1 ]
```

Como vemos, los resultados son correctos, ya que en todo momento se encuentra el camino cuyo valor temporal sea el más pequeño de entre todos los posibles.

Un claro ejemplo para verlo es el nodo 2, que no conecta directamente con el servidor central. Este avanza hacia el nodo 0, de entre todas las opciones disponibles(0,3,4) porque es el que menor peso tiene. Y a continuación también lo hace hacia el nodo 1 porque es el que menor peso tiene.

Por tanto viendo este ejemplo podemos concluir que el algoritmo greedy funciona correctamente.

Para concluir, haremos un breve repaso de cómo actúa el algoritmo:

1. Inicialización de variables: Se inicializan las variables necesarias.
2. Selección del nodo servidor inicial: Se añade el nodo inicial al vector elegidos.
3. Bucle principal: Se ejecuta un bucle mientras no se hayan seleccionado todos los nodos del grafo.

4. Selección de candidatos: Para cada nodo del vector elegidos, se buscan sus conexiones que no se han seleccionado aún. Estos nodos no seleccionados se consideran candidatos.
5. Cálculo del peso mínimo: Se calcula el peso total de cada candidato sumando el tiempo de conexión con su nodo padre y el peso acumulado del nodo padre. Se busca aquel candidato que tenga el menor peso.
6. Actualización de la solución: Se actualiza la solución con el camino más corto para llegar a cada nodo. Se añade el nodo seleccionado al camino del nodo candidato.
7. Actualización de seleccionados: Se incrementa el contador de los nodos seleccionados y se añade el nuevo nodo seleccionado al vector.

Cuando acaba el bucle el vector de la solución contendrá todos los nodos seleccionados con los caminos de menor tiempo posible.

Asfaltar Calles (Problema 5)

Introducción.

Para resolver el problema, el algoritmo greedy implementado parte de una plaza aleatoria, y busca la plaza con menor coste para asfaltar la calle a la que poder ir.
En este problema he supuesto que la plaza en la que empezamos es aleatoria, lo que no siempre devuelve el camino más económico, para resolver este problema tendríamos que empezar en la plaza Azcárate o en la plaza de la libertad.

```cpp
int buscar_menor(vector<vector<int>> matriz, int ciudad_actual, int ciudades_visitadas[5]){
    int menor = -1;
    for (int i = 0; i < 5; i++){
        if (ciudades_visitadas[i] == 0 && ciudad_actual != i && matriz[ciudad_actual][i] != -1){
            if (menor == -1 ){
                menor = i;
            } else if (matriz[ciudad_actual][i] < matriz[ciudad_actual][menor]){
                menor = i;
            }
        }
    }
    return menor;
}

int main() {
    // Semilla para generar números aleatorios
    std::srand(std::time(0));

    // Crear una matriz 5x5
    vector<vector<int>> matriz(5);
    int ciudades_visitadas[5] = {0, 0, 0, 0, 0};

    int ciudad_inicial = std::rand() % 5;

    //Plaza Azcárate = 0
    //Plaza de las maravillas = 1
    //Plaza de la constitución = 2
    //Plaza de la libertad = 3
    //Plaza mayor = 4

    matriz[0] = {0, 450004, 748456, 1100000, 130000};
    matriz[1] = {450004, 0, 143552, 555123, -1};
    matriz[2] = {748456, 143552, 0, 1700000, 554521};
    matriz[3] = {1100000, 555123, 1700000, 0, 1300000};
    matriz[4] = {13000, -1, 554521, 1300000, 0};

    int ciudad_actual = ciudad_inicial;
    int ciudad_siguiente = -1;
    int distancia_total = 0;

    for (int i = 0; i < 5; i++){
        cout << "Ciudad actual: " << ciudad_actual << endl;
        ciudades_visitadas[ciudad_actual] = 1;
        ciudad_siguiente = buscar_menor(matriz, ciudad_actual, ciudades_visitadas);
        if (ciudad_siguiente == -1) {
            break;
        }
        cout << "Ciudad siguiente: " << ciudad_siguiente << endl;
        distancia_total += matriz[ciudad_actual][ciudad_siguiente];
        ciudad_actual = ciudad_siguiente;
    }

    cout << "Distancia total: " << distancia_total << endl;
}
```

Diseño de Componentes

→ **Lista de candidatos:** Las 5 plazas del pueblo.
→ **Lista de candidatos usados:** Las plazas que ya han sido visitadas.
→ **Criterio de selección:** Plaza que tienen un menor coste para asfaltar la calle que las comunica.
→ **Criterio de factibilidad:** La plaza no debe haber sido visitada antes.
→ **Función solución:** Recorrido óptimo para visitar todas las ciudades a partir de una aleatoria.
→ **Función objetivo:** La solución de este problema es obtener el recorrido más económico para visitar todas las ciudades a partir de otra aleatoria.

Diseño del algoritmo en base a la plantilla Greedy

```
Función buscar_menor(matriz, ciudad_actual, ciudades_visitadas):
    Definir menor como -1
    Para i desde 0 hasta 5:
        Si ciudades_visitadas[i] es 0 y ciudad_actual no es igual a i y matriz[ciudad_actual][i] no es igual a -1:
            Si menor es -1:
                Definir menor como i
            Sino si matriz[ciudad_actual][i] es menor que matriz[ciudad_actual][menor]:
                Definir menor como i
    Devolver menor

Función principal:
    Generar una semilla para números aleatorios basada en el tiempo actual

    Crear una matriz 5x5
    Definir ciudades_visitadas como un arreglo de 5 elementos, todos inicializados a 0

    Generar una ciudad_inicial aleatoria entre 0 y 4

    Definir la matriz con los valores de las distancias entre ciudades

    Definir ciudad_actual como ciudad_inicial
    Definir ciudad_siguiente como -1
    Definir distancia_total como 0

    Para i desde 0 hasta 5:
        Imprimir "Ciudad actual: " y ciudad_actual
        Marcar ciudad_actual como visitada en ciudades_visitadas
        Definir ciudad_siguiente como el resultado de buscar_menor(matriz, ciudad_actual, ciudades_visitadas)
        Si ciudad_siguiente es -1, romper el bucle
        Imprimir "Ciudad siguiente: " y ciudad_siguiente
        Aumentar distancia_total por matriz[ciudad_actual][ciudad_siguiente]
        Definir ciudad_actual como ciudad_siguiente

    Imprimir "Distancia total: " y distancia_total

    Generar una nueva ciudad_inicial aleatoria entre 0 y 4

    Reiniciar el arreglo de ciudades_visitadas a 0

    Definir ciudad_actual como ciudad_inicial
    Definir ciudad_siguiente como -1
    Definir distancia_total como 0
```

```
Para i desde 0 hasta 5:
    Imprimir "Ciudad actual: " y ciudad_actual
    Marcar ciudad_actual como visitada en ciudades_visitadas
    Definir ciudad_siguiente como el resultado de buscar_menor(matriz, ciudad_actual, ciudades_visitadas)
    Si ciudad_siguiente es -1, romper el bucle
    Imprimir "Ciudad siguiente: " y ciudad_siguiente
    Aumentar distancia_total por matriz[ciudad_actual][ciudad_siguiente]
    Definir ciudad_actual como ciudad_siguiente

Imprimir "Distancia total: " y distancia_total
```

Estudio de optimalidad

A la hora de evaluar la optimalidad de nuestro algoritmo que resuelve el problema, podemos observar que no es un algoritmo óptimo, si no empieza en determinadas ciudades (pero se podría hacer óptimo guardando todos los recorridos y luego elegir el más corto).

Como observamos en el código, buscamos siempre la ciudad con menor coste para asfaltar, pero como no podemos volver para atrás en una calle ya asfaltada, dependiendo de donde empecemos el recorrido va a ser óptimo o no. Si empezamos en la plaza Azcárate o en la plaza de la libertad, vamos a tener un recorrido óptimo, sino no.

Para crear un algoritmo que encuentre el recorrido óptimo tendríamos que guardar las distancia de todos los recorridos empezados por cada ciudad y elegir el menor, (si empezamos en plaza Azcárate o en la plaza de la libertad el recorrido va a ser igual ya que son el principio y fin del mejor recorrido).

Contraejemplo, siempre elegimos el camino de menor coste, Empezamos en plaza Mayor, plaza Azcárate, plaza de la Constitución, Plaza de las Maravillas, Plaza de la libertad. El coste total es 2306556.
El recorrido óptimo es empezamos en plaza Azcárate, plaza Mayor, plaza de la Constitución, Plaza de las Maravillas, Plaza de la libertad (también en sentido contrario) cuyo coste es: 1383196.

Funcionamiento del algoritmo

El algoritmo consiste en buscar la plaza con el menor coste para asfaltar la calle y guardarla en un vector de plazas visitadas y hacer así con todas hasta que se hayan visitado las 5 plazas.
Cada vez que vayamos a una plaza la guardamos en el vector de ciudades visitadas, y sumamos la distancia para visitar la plaza con menor coste (la distancia sería el coste).
Para encontrar la plaza con el menor coste para asfaltar la calle usamos una función auxiliar, “buscar_menor” que busca la plaza con el menor coste entre las posibles (las que todavía no han sido visitadas, y las que están unidas).
Una vez que hayamos visitado las 5 plazas imprimimos el coste de visitar todas las plazas.

Ejemplo óptimo:
Empezamos en plaza Azcárate (0), buscamos la plaza con menor coste, tenemos entre las opciones:
plaza Mayor (4) coste 130000
plaza de la Constitución(2) coste 748456
Plaza de las Maravillas (1) coste 450004
Plaza de la libertad (3) coste 1100000
vamos a plaza Mayor (4) porque tiene el menor coste.
Ahora nos desplazamos a plaza mayor y repetimos lo mismo con las otras plazas y el orden que nos sale es: plaza Azcárate (0), plaza Mayor (4), plaza de la Constitución(2), Plaza de las Maravillas (1), Plaza de la libertad (3). Con un coste es: 1383196.