Práctica 3:
Algoritmos Voraces

Santiago Carbó García y Carlos García Segura

Universidad de Granada, España

1. Ejercicio 1

1.1. Diseño de componentes

- Conjunto de candidatos: conjunto de nodos $(n_1, n_2, ..., n_n)$ por escoger. Inicialmente contendrá todos los nodos menos el nodo 1 (servidor central).
- Candidatos ya usados: inicialmente contendrá el nodo 1 (servidor central) y, posteriormente, todos los nodos.
- Función Solución: una lista de sensores intermedios y distancias entre el servidor central y los demás nodos.
- Condición de Factibilidad: siempre es factible insertar un nodo en la solución optimal.
- Función de Selección: escogemos el sensor que no haya sido marcado como utilizado, cuya distancia al origen, en términos de tiempo de envío, sea menor.
- Función Objetivo: encontrar el camino mínimo, en términos de tiempo de envío, desde el servidor central a todos los demás.

Análisis del problema:

- ¿En qué consiste el problema?: conectar el servidor central con cualquier nodo, de forma que el tiempo de envío sea mínimo.
- ¿El origen y el destino deberían ser el mismo?: en general, no debería ser el mismo.
- ¿Qué se debe obtener?: un grafo dirigido parcial, conexo y sin ciclos, tal que el tiempo de envío sea mínimo.
- ¿Qué tipo de problema es?: Problema del Camino Mínimo.
- ¿Algoritmos relacionados?: Algoritmo de Dijkstra.

1.2. Diseño del algoritmo

Nos encontramos ante un problema de Camino Mínimo. Hemos utilizado el algoritmo de Dijkstra. El algoritmo calcula el camino mínimo entre el nodo origen y el resto de nodos.

Variables principales utilizadas:

- visitados[x]: indica si el nodo x ha sido comprobado.
- D[x]: guarda las distancias, en términos de tiempo de envío, desde el servidor central hasta el resto de nodos.
- P[x]: guarda los elementos anteriores por los que hay que pasar en el camino mínimo entre el nodo inicial dado y el nodo x.

A continuación, se presenta al algoritmo resumido en pseudocódigo:

```
redSensoresGreedy (matrix L, Integer N, Integer nodo_inicial)
        Integer D[N]
        Integer P[N]
        Boolean visitados[N]

        Para i := 0 hasta N hacer
                D[i] := ∞
                visitados[i] := falso
                P[i] := nodo_inicial
        fin bucle
 D[nodo_inicial] := 0
 visitados[1] := true

 Para j := 0 hasta N hacer
        x = tiempo de envío min entre los no visitados
        visitado[x] = cierto;

        Para i := 0 hasta N hacer
                Si D[i] > D[x] + L[x, i] AND visitados[i] == falso entonces
                        D[i] = D[x] + L[x, i]
                        P[i] = x
                fin condicional
         fin bucle
 fin bucle
FIN redSensoresGreedy
```

A continuación, se presenta el algoritmo implementado en C++:

```cpp
int tiempoEnvioMin(int *tiempoEnvio, bool *visitados, int N)
{
    int nodo_min;
    int minimum = 999999;

    for(int i = 0; i < N; i++)
    {
        if(tiempoEnvio[i] <= minimum && visitados[i] == false)
        {
            minimum = tiempoEnvio[i];
            nodo_min = i;
        }
    }

    return nodo_min;
}
```

```cpp
void redSensoresGreedy(int **L, int N, int nodo_inicial)
{
    bool *visitados = new bool[N];
    int *tiempoEnvio = new int[N];
    vector<int> intermedios;
    intermedios.resize(N);

    for(int i = 0; i < N; i++)
    {
        tiempoEnvio[i] = 999999;
        visitados[i] = false;
        intermedios[i] = 1;
    }

    tiempoEnvio[nodo_inicial] = 0;

    for(int j = 0; j < N; j++)
    {
        int min = tiempoEnvioMin(tiempoEnvio, visitados, N);
        visitados[min] = true;

        for(int i = 0; i < N; i++)
        {
            if(tiempoEnvio[i] > tiempoEnvio[min] + L[min][i] && !visitados[i] && L[min][i] > 0)
            {
                tiempoEnvio[i] = tiempoEnvio[min] + L[min][i];
                intermedios[i] = min+1;
            }
        }
    }

    cout << "Sensor\t\t";
    cout << "Tiempo de envio" << "\t\t";
    cout << "Nodo anterior" << endl;

    for(int i = 0; i < N; i++)
        cout << i+1 << "\t\t\t" << tiempoEnvio[i] << "\t\t\t" << intermedios[i]<< endl;


}
```

¿Qué hace el algoritmo?

- La función “tiempoEnvioMin” calcula cuál es el nodo con el tiempo de envío mínimo y que no haya sido visitado.
- La función “redSensoresGreedy” es la implementación del algoritmo de Dijkstra aplicado a este caso. En primer lugar rellena el vector “tiempoEnvio” con los tiempos de envío máximos posible, el vector visitados con todos a “false” y el vector de nodos intermedios con el valor del nodo inicial, ya que no hemos visitado ningún nodo. Tras ello, indicamos que el tiempo de envío al nodo inicial (servidor central) es 0. Dentro del primer bucle calculamos el mínimo entre los nodos no visitados y lo marcamos como visitado. Finalmente actualizamos los datos: si el tiempo de envío a cada nodo es mayor que que el tiempo de envío pasando por el nodo de tiempo mínimo, actualizamos su valor y guardamos su nodo intermedio.
- El orden es $O(n^2)$.

1.3. Estudio de optimalidad (demostración o contraejemplo)

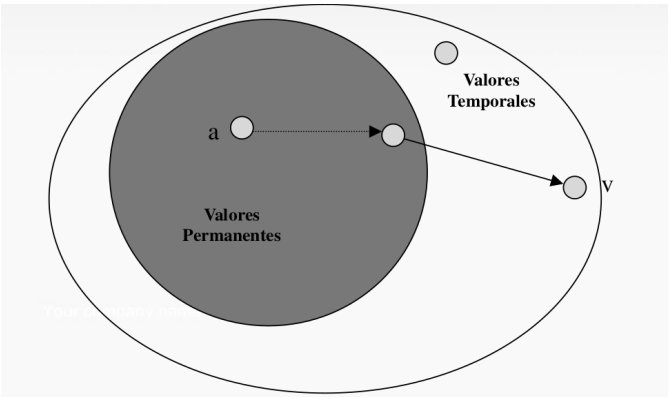
Para empezar, llamamos S al conjunto de nodos elegidos. Tenemos un vector D que contiene la longitud del camino especial (camino de un origen a otro nodo en el que todos los nodos intermedios están en S) más corto a cada nodo del grafo.

La demostración de la corrección va a ser realizada por inducción, de forma que es necesaria una base y una hipótesis de inducción:

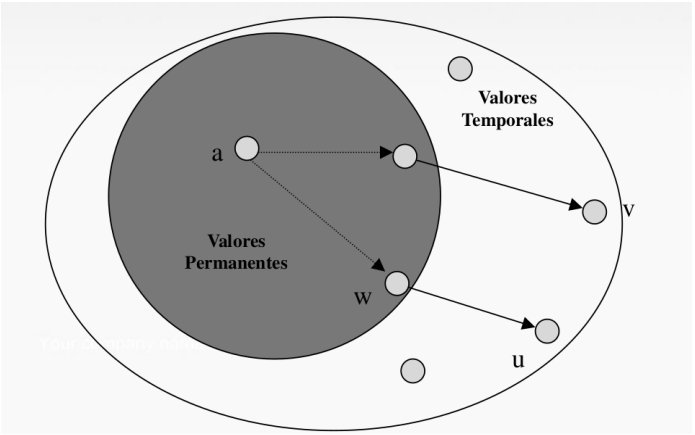
- **Base** (iteración 1):
  - Inicialmente $D[a] = 0$, y todos los demás $D's$ son mayores que cero, por tanto se elige el origen a.
  - $D[a]$ es el camino más corto desde a a a.
  - $T(1)$, por ende, es cierta.
- **Hipótesis de Inducción T(i)**:
  - En la i-ésima iteración del lazo del algoritmo, el valor $D[V]$ del vértice v se hace permanente.
  - $D[v]$ siempre es el valor mínimo entre los valores temporales.
  - El valor $D[v]$ da el camino mínimo desde el origen hasta el vértice v.

Tras haber definido esto:

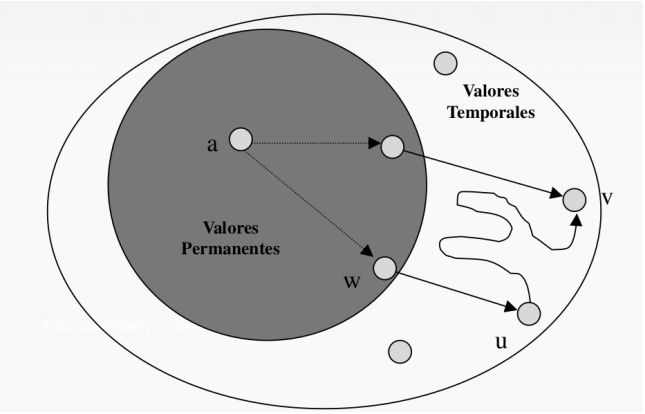
- Supongamos que $T(k)$ es cierto para todo $k < i$. En la iteración $k + 1$ seleccionamos el nodo v, y nos hacemos la siguiente pregunta: ¿Es $D(v)$ el camino más corto desde a a v?



- Supongamos que hay otro vértice u tal que el camino hasta v que pasa por u es más corto que el anterior (el que nos daba $D(v)$).



- Obligatoriamente el camino que pasa a través de u tiene que ser como el de la figura.



- Recordamos la preguntas que nos hemos hecho antes: “en la iteración $k + 1$ seleccionamos el nodo v: ¿Es $D(v)$ el camino más corto desde a a v?”
    - La respuesta es **sí**. Por tanto, $T(k + 1)$ es cierto y, como la base y la hipótesis de inducción son ciertas, $T(i)$ es cierta para todo i.

1.4. Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para una instancia pequeña

Matriz de adyacencias

	Central	**2**	**3**	**4**	**5**
Central	X	50	30	100	10
2	50	X	5	20	X
3	30	5	X	50	X
4	100	20	50	X	50
5	10	X	X	X	X

Nodo Origen = Nodo Central
Lista de candidatos inicial = 2, 3, 4, 5
Vector Distancias Inicial = 0, 50, 30, 100, 10
Vector Elementos Intermedios = “”

- Primera interacción:
    - Lo primero será escoger el elemento de la lista de candidatos que posea la menor distancia en el vector Distancias.
    - En nuestro caso el candidato elegido sería ‘5’ con una distancia de ‘10’.
    - Una vez elegido lo eliminamos de las lista de candidatos
    - Acto seguido comprobamos si alguna distancia del vector distancia se reduce al pasar primero por en nodo 5, si esto es así, se actualiza el vector distancias con esas nuevas distancias.
    - En nuestro caso:
        - Vector distancias original [0, 50, 30, 100, 10]
        - Vector distancia pasando por el nodo 5 [X, X, X, 20, X]
        - Como la distancia para ir al nodo 4 atravesando primero el nodo 5 es menor que yendo directamente desde el nodo Central, actualizamos.
        - Vector distancias actualizado [0, 50, 30, 20, 10]
        - Lista de candidatos = 2, 3, 4
    - Como para ir al nodo 4 es mejor pasar primero por el nodo 5 lo añadimos al vector de elementos intermedios.
    - Elementos Intermedios[4] = 5

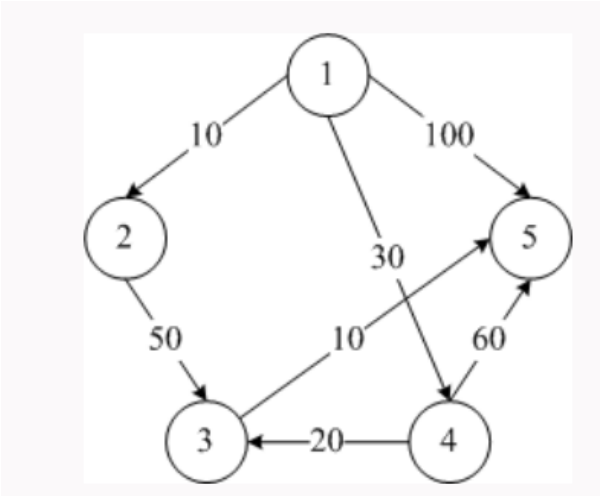
- Segunda Interacción:
    - Volvemos a escoger el elemento de la lista de candidatos que posea la menor distancia en el vector Distancias.
    - En nuestro caso el candidato elegido sería ‘4’ con una distancia de ‘20’.
    - Una vez elegido lo eliminamos de las lista de candidatos
    - Acto seguido comprobamos si alguna distancia del vector distancia se reduce al pasar primero por en nodo 4, si esto es así, se actualiza el vector distancias con esas nuevas distancias.
    - En nuestro caso:
        - Vector distancias original [0, 50, 30, 20, 10]
        - Vector distancia pasando por el nodo 4 [X, 40, 70, X, X]
        - Como la distancia para ir al nodo 2 atravesando primero el nodo 4 es menor que yendo directamente desde el nodo Central, actualizamos.
        - Vector distancias actualizado [0, 40, 30, 20, 10]
        - Lista de candidatos = 2, 3
    - Como para ir al nodo 2 es mejor pasar primero por el nodo 4 lo añadimos al vector de elementos intermedios.
    - Elementos Intermedios[2] = 4

- Tercera Interacción:

- Volvemos a escoger el elemento de la lista de candidatos que posea la menor distancia en el vector Distancias.
- En nuestro caso el candidato elegido sería ‘3’ con una distancia de ‘30’.
- Una vez elegido lo eliminamos de las lista de candidatos
- Acto seguido comprobamos si alguna distancia del vector distancia se reduce al pasar primero por en nodo 3, si esto es así, se actualiza el vector distancias con esas nuevas distancias.
- En nuestro caso:
    - Vector distancias original [0, 40, 30, 20, 10]
    - Vector distancia pasando por el nodo 3 [X, 35, X, X, X]
    - Como la distancia para ir al nodo 2 atravesando primero el nodo 3 es menor que yendo directamente desde el nodo Central, actualizamos.
    - Vector distancias actualizado [0, 35, 30, 20, 10]
    - Lista de candidatos = 2
- Como para ir al nodo 2 es mejor pasar primero por el nodo 3 lo añadimos al vector de elementos intermedios.
- Elementos Intermedios[2] = 3

- Cuarta Interacción:
    - Escoger el elemento de la lista de candidatos restante.
    - En nuestro caso el candidato elegido sería ‘2’ con una distancia de ‘35’.
    - Una vez elegido lo eliminamos de las lista de candidatos
    - Acto seguido comprobamos si alguna distancia del vector distancia se reduce al pasar primero por en nodo 2, si esto es así, se actualiza el vector distancias con esas nuevas distancias.
    - En nuestro caso:
        - Vector distancias original [0, 35, 30, 20, 10]
        - Vector distancia pasando por el nodo 3 [X, X, X, X, X]
        - No existe ninguna distancia menor.
        - Vector distancias actualizado [0, 35, 30, 20, 10]
        - Lista de candidatos = “”
    - Ya tenemos el vector de distancias mínimas desde el nodo central y el vector de elementos intermedios para llegar a cada uno de los nodos.

1.5. Correcto funcionamiento de la implementación



ITERACIÓN	S	w	D[2]	D[3]	D[4]	D[5]
INICIAL	{1}	—	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

```
carlosgs@pssito:/mnt/c/Users/carlo/Desktop$ ./a.out
Sensor          Tiempo de envío         Nodo anterior
1                       0                       1
2                       10                      1
3                       50                      4
4                       30                      1
5                       60                      3
```

Hemos aplicado este caso de ejemplo para comprobar que el algoritmo se está aplicando correctamente. Como podemos observar la solución otorgada por el código es la correcta.

2. Ejercicio 2

2.1. Diseño de componentes

- Conjunto de candidatos: todas las gasolineras que se encuentran entre el origen y el destino del autobús. Al final del algoritmo este conjunto estará vacío.
- Candidatos ya usados: las gasolineras que vamos evaluando.
- Función Solución: una lista de gasolineras, en términos de sus distancias.
- Condición de Factibilidad: no puede acabarse la gasolina del autobús.
- Función de Selección: elegimos aquella gasolinera más lejana.
- Función Objetivo: realizar el mínimo número de paradas posibles antes de llegar al destino.

Análisis del problema:

- ¿En qué consiste el problema?: tenemos una ruta que seguir desde un origen a un destino y queremos hacer el mínimo número de paradas posibles, siempre que no se nos acabe la gasolina antes de llegar a una parada.
- ¿El origen y el destino deberían ser el mismo?: en general, no debería ser el mismo.
- ¿Qué tipo de problema es?: variante del Problema del Camino Mínimo. Aquí ya tenemos el camino preestablecido, pero queremos hacer el mínimo número de paradas posibles.
- ¿Algoritmos relacionados?: variante del Algoritmo de Dijkstra.

2.2. Diseño del algoritmo

Nos encontramos ante una variante del problema del Camino Mínimo. Tenemos un camino marcado pero queremos recorrerlo haciendo el número mínimo de paradas posible, de forma que puede utilizar un algoritmo derivado del Algoritmo de Dijkstra con una condición de factibilidad nueva, que es que la gasolina no se acabe. Variables importantes del algoritmo:

- lista_gasolineras(1:N) contiene todas las gasolineras que son candidatas a ser elegidas.
- depósitoCombustible es la cantidad de gasolina en kilómetros de la que disponemos.
- gasolineraMasLejana indica la gasolinera de mayor distancia a la que podemos llegar, para minimizar el número de paradas.
- itinerario_paradas son las gasolineras que conforman el itinerario final.

A continuación, se presenta al algoritmo escrito en pseudocódigo:

```
autobusGreedy(lista_gasolineras(1:N), Integer depositoCombustible)
        itinerario_paradas(1:N);

        Mientras lista_gasolineras no esté vacía, hacer
                gasolineraMasLejana := 0;
                depositoCombustibleAux := depositoCombustible;

                Para i:= 0 hasta N-1 AND continuar == true hacer
                        depositoCombustibleAux := depositoCombustibleAux - lista_gasolineras [i]:
                        gasolineraMasLejana := gasolineraMasLejana + lista_gasolineras [i];

                        Si depositoCombustibleAux < 0
                          depositoCombustibleAux := depositoCombustibleAux + lista_gasolineras [i]:
                          gasolineraMasLejana := gasolineraMasLejana - lista_gasolineras [i];
                          continuar := false;
                        fin condicional

                        Si lista_gasolineras no está vacío, entonces
                                lista_gasolineras.delete(i);
                        fin condicional
                fin bucle
                itinerario_paradas.add(gasolineraMasLejana);
        fin bucle
Devolver itinerario_paradas.
FIN autobusGreedy
```

A continuación, se presenta el algoritmo implementado en C++:

```cpp
vector<int> autobusGreedy(vector<int> lista_gasolineras, const int depositoCombustible){
    vector<int> itinerario_paradas;

    while(!lista_gasolineras.empty()) {
        int gasolineraMasLejana = 0;
        int depositoCombustibleaux = depositoCombustible;
        bool continuar = true;

        for(vector<int>::iterator it = lista_gasolineras.begin(); it != lista_gasolineras.end() && continuar; it++){
            depositoCombustibleaux -= (*it);
            gasolineraMasLejana += (*it);

            if(depositoCombustibleaux < 0){
                depositoCombustibleaux += (*it);
                gasolineraMasLejana -= (*it);
                continuar = false;
            }

            if(!lista_gasolineras.empty()){
                it = lista_gasolineras.erase(it);
            }
        }
        itinerario_paradas.push_back(gasolineraMasLejana);
    }

    return itinerario_paradas;
}
```

¿Qué hace el algoritmo?

- Recibimos como parámetros iniciales una lista de gasolineras representadas como su distancia entre ella y su anterior (en el caso de la primera gasolinera, entre ella y el origen) y el depósito de combustible representado en los kilómetros que podemos hacer antes de que se acabe.
- El primer bucle se recorre siempre que la lista de gasolineras no esté vacía (se van eliminando las gasolineras una vez la hemos comprobado). La variable “gasolineraMasLejana” irá acumulando la distancia que podemos recorrer antes de que se nos acabe la gasolina y la variable “depositoCombustibleAux” irá indicando la gasolina que no queda. En el momento en el que nos quedamos sin gasolina, es decir, “depositoCombustibleAux < 0”, rectificamos los valores de la variable y salimos de este segundo bucle para añadir la gasolinera más lejana que hemos encontrado a “itinerario_paradas”.
- El orden es $O(n)$.

2.3. Estudio de optimalidad (demostración o contraejemplo)

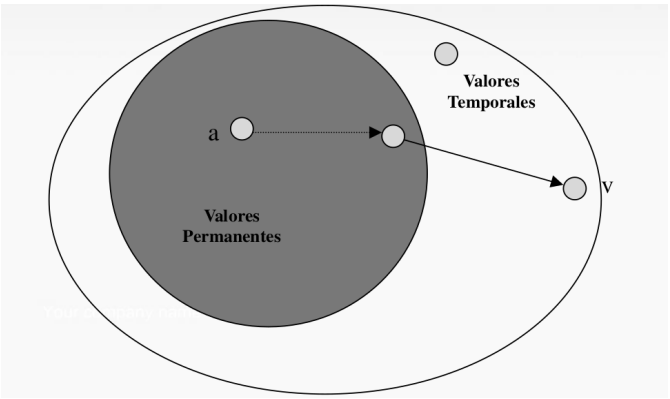
Para empezar, llamamos S al conjunto de nodos elegidos. Tenemos un vector D que contiene la longitud del camino especial (camino de un origen a otro nodo en el que todos los nodos intermedios están en S) más corto a cada nodo del grafo.

La demostración de la corrección va a ser realizada por inducción, de forma que es necesaria una base y una hipótesis de inducción:

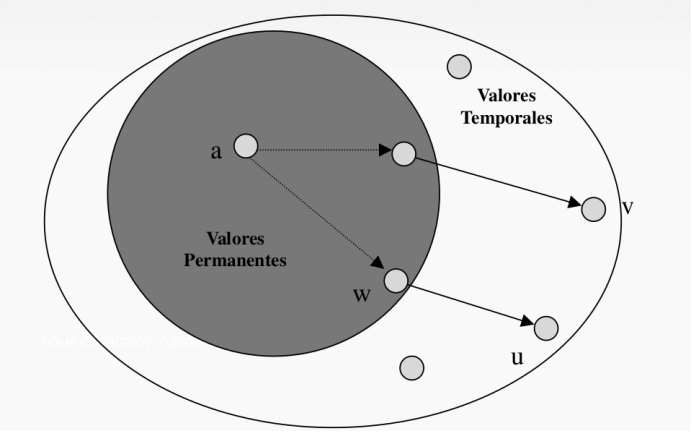
- **Base** (iteración 1):
  - Inicialmente $D[a] = 0$, y todos los demás $D's$ son mayores que cero, por tanto se elige el origen a.
  - $D[a]$ es el camino más corto desde a a a.
  - $T(1)$, por ende, es cierta.
- **Hipótesis de Inducción T(i)**:
  - En la i-ésima iteración del lazo del algoritmo, el valor $D[V]$ del vértice v se hace permanente.
  - $D[v]$ siempre es el valor mínimo entre los valores temporales.
  - El valor $D[v]$ da el camino mínimo desde el origen hasta el vértice v.

Tras haber definido esto:

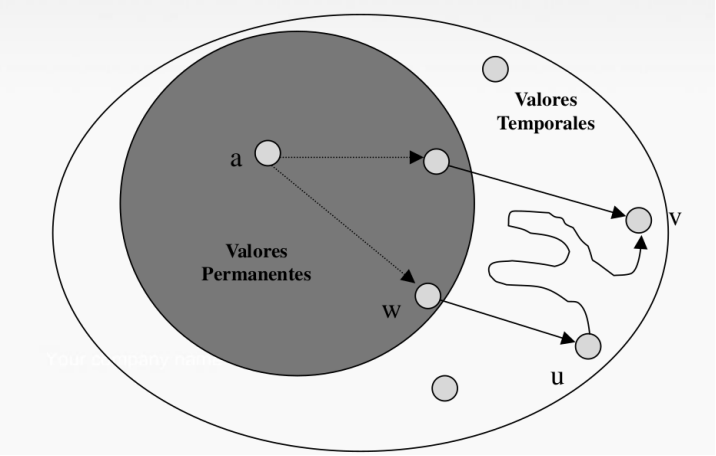
- Supongamos que $T(k)$ es cierto para todo $k < i$. En la iteración $k + 1$ seleccionamos el nodo v, y nos hacemos la siguiente pregunta: ¿Es $D(v)$ el camino más corto desde a a v?



- Supongamos que hay otro vértice u tal que el camino hasta v que pasa por u es más corto que el anterior (el que nos daba $D(v)$).



- Obligatoriamente el camino que pasa a través de u tiene que ser como el de la figura.



- Recordamos la preguntas que nos hemos hecho antes: “en la iteración $k + 1$ seleccionamos el nodo v: ¿Es $D(v)$ el camino más corto desde a a v?”
    - La respuesta es **sí**. Por tanto, $T(k + 1)$ es cierto y, como la base y la hipótesis de inducción son ciertas, $T(i)$ es cierta para todo i.

2.4. Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para una instancia pequeña

- Existen 5 gasolineras y tenemos 20 km de depósito.
- Cada una de las gasolineras se representan en función de su distancia a donde estamos:
    - g1 = 5
    - g2 = 20
    - g3 = 10
    - g4 = 15
    - g5 = 17

- Empezamos en el punto 0.
    - gasolineraMasLejana = 0
    - gasolinerasCandidatas [g1,g2,g3,g4,g5]
    - gasolinerasSolucion []

- Cogemos la primera gasolinera y la actualizamos.
    - gasolineraMasLejana += 5 (g1) (Valor de gasolineraMasLejana = 5)
    - despositoCombustible -= 5 (g1) (Valor de depositoCombustible = 15)

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

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

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

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

- Repostamos en g1.
    - (Valor de depositoCombustible = 20)
    - (Valor de gasolineraMasLejana = 0)

- Comprobamos la siguiente gasolinera.
    - gasolineraMasLejana += 10 (g3) (Valor de gasolineraMasLejana = 10)
    - despositoCombustible -= 10 (g3) (Valor de depositoCombustible = 10)

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

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

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

- Como el depósito se ha vaciado (depositoCombustible < 0) añadimos a gasolinerasSolucion la gasolinera anterior.
    - gasolinerasSolucion [g1, g3];

- Repostamos en g3.
    - (Valor de depositoCombustible = 20)
    - (Valor de gasolineraMasLejana = 0)

- Comprobamos la siguiente gasolinera.
    - gasolineraMasLejana += 17 (g5) (Valor de gasolineraMasLejana = 17)
    - despositoCombustible -= 17 (g5) (Valor de depositoCombustible = 3)

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

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

2.5. Correcto funcionamiento de la implementación

```cpp
vector<int> autobusGreedy(vector<int> lista_gasolineras, const int depositoCombustible){
    vector<int> itinerario_paradas;

    while(!lista_gasolineras.empty()) {
        int gasolineraMasLejana = 0;
        int depositoCombustibleaux = depositoCombustible;
        bool continuar = true;

        for(vector<int>::iterator it = lista_gasolineras.begin(); it != lista_gasolineras.end() && continuar; it++){
            depositoCombustibleaux -= (*it);
            gasolineraMasLejana += (*it);

            if(depositoCombustibleaux < 0){
                depositoCombustibleaux += (*it);
                gasolineraMasLejana -= (*it);
                continuar = false;
            }

            if(!lista_gasolineras.empty()){
                it = lista_gasolineras.erase(it);
            }
        }
        itinerario_paradas.push_back(gasolineraMasLejana);
    }

    return itinerario_paradas;
}
```

```
carlosgs@pssito: /mnt/c/Users/carlo/Desktop
carlosgs@pssito:/mnt/c/Users/carlo/Desktop$ g++ gasolineras.cpp
carlosgs@pssito:/mnt/c/Users/carlo/Desktop$ ./a.out
Kilometros recorridos hasta la siguiente gasolinera:  72
Kilometros recorridos hasta la siguiente gasolinera:  78
Kilometros recorridos hasta la siguiente gasolinera:  94
Kilometros recorridos hasta la siguiente gasolinera:  75
Kilometros recorridos hasta la siguiente gasolinera:  59
Kilometros recorridos hasta la siguiente gasolinera:  90
Kilometros recorridos hasta la siguiente gasolinera:  63
Kilometros recorridos hasta la siguiente gasolinera:  60
Kilometros recorridos hasta la siguiente gasolinera:  67
Kilometros recorridos hasta la siguiente gasolinera:  71
Kilometros recorridos hasta la siguiente gasolinera:  85
Kilometros recorridos hasta la siguiente gasolinera:  72
Kilometros recorridos hasta la siguiente gasolinera:  96
Kilometros recorridos hasta la siguiente gasolinera:  66
Kilometros recorridos hasta la siguiente gasolinera:  89
Kilometros recorridos hasta la siguiente gasolinera:  77
Kilometros recorridos hasta la siguiente gasolinera:  94
Kilometros recorridos hasta la siguiente gasolinera:  89
Kilometros recorridos hasta la siguiente gasolinera:  44
Kilometros recorridos hasta la siguiente gasolinera:  27
Kilometros recorridos hasta la siguiente gasolinera:  40
Kilometros recorridos hasta la siguiente gasolinera:  96
Numero de gasolineras recorridas: 22
Tiempo ejecucion:11
carlosgs@pssito:/mnt/c/Users/carlo/Desktop$
```

Hemos usado un vector con 50 gasolineras donde el tamaño del depósito es 100. Como podemos observar hace uso de 22 gasolineras para realizar todo el recorrido y en ningún momento supera los 100 km de depósito de gasolina.

3. Ejercicio 3

3.1. Diseño de componentes

Nos encontramos ante el problema de la mochila fraccional.

- Conjunto de candidatos: conjunto de contenedores $(c_1, c_2, ..., c_n)$ por escoger.
- Candidatos ya usados: aquellos contenedores ya escogidos.
- Función Solución: una lista de contenedores.
- Condición de Factibilidad: El peso de los contenedores debe ser menor que la capacidad del buque.
- Función de Selección: escogemos el contenedor con más densidad p_i/w_i.
- Función Objetivo: maximizar la suma de los pesos de los contenedores a transportar en el barco, es decir, cargar el número máximo de contenedores.

Análisis del problema:

- ¿En qué consiste el problema?: tenemos una serie de contenedores, con una serie de pesos cada uno, que queremos introducir en un barco, pero el barco tiene una capacidad. El objetivo es maximizar la suma de los pesos de los contenedores sin que sobrepase la capacidad del barco.
- ¿Qué tipo de problema es?: Problema de la mochila fraccional.
- ¿Algoritmos relacionados?: Algoritmo de la mochila fraccional.

3.2. Diseño del algoritmo

Nos encontramos ante un problema de la **mochila fraccional**. Variables importantes del algoritmo:

- P(1:n) y W(1:n) contiene el número de contenedores y pesos respectivos de los N grupos de contenedores ordenados como P(I)/W(I) >= P(I+1)/W(I+1).
- k es la capacidad del buque.
- X(1:n) es el vector solución. Es un vector de números reales ya que el último objeto será fraccional.

A continuación, se presenta al algoritmo escrito en pseudocódigo:

buqueGreedy(P(1:N), W(1:N), Integer k, Integer N)

```
X(1:N);
Boolean continuar := true;
Integer cr := k;

Para i:= 0 hasta N-1 hacer
        Si W[i] > cr entonces
                continuar = false;
        En otro caso
                X[i] := 1;
                cr = cr - W[i];
        fin condicional
        Si continuar == false AND cr > 0
                X[i] := cr/W[i];
        fin condicional
fin bucle

Devolver X.
```

FIN buqueGreedy

¿Qué hace el algoritmo?

- El algoritmo, inicialmente, tiene ordenado W en **orden densidad no creciente**. Después, va introduciendo los contenedores en el buque (es decir, va añadiendo 1's al vector solución), hasta que el siguiente peso es mayor que la capacidad restante. En ese caso comprobamos si podemos introducir un último objeto de manera fraccionada y, si se puede, lo introducimos.
- El orden es $O(n)$.

A continuación, se presenta al algoritmo escrito en C++:

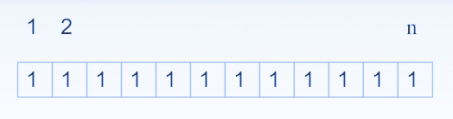
```cpp
struct  Conjunto_Contenedor{
    int num_contenedores;
    int peso;
    float numcont_peso;
    bool operator < (Conjunto_Contenedor & obj1) {
        return (numcont_peso > obj1.numcont_peso);
    }
    friend ostream & operator<< (ostream & os, const Conjunto_Contenedor & o) {
        os << "Peso(toneladas) " << o.peso << '\t' << "Numero de contenedores: " <<         o.num_contenedores    << '(' << o.numcont_peso << ')';
        return os;
    }
    Conjunto_Contenedor (int nc, int p) {
        num_contenedores = nc;
        peso = p;
        numcont_peso = (float)num_contenedores/peso;
    }
};

list<float> Barco (int lim_peso, list<Conjunto_Contenedor> & contenedores) {
    list<float> sol;
    int peso_actual = 0;
    list<Conjunto_Contenedor>::iterator rit;

    for (rit=contenedores.begin(); rit!=contenedores.end() && lim_peso > peso_actual; ++rit) {
        if ((peso_actual + (*rit).peso) <= lim_peso) {
            sol.push_back(1);
            peso_actual += (*rit).peso;
        }

        else {
            float num_c = (float)(lim_peso-peso_actual)/((*rit).peso);
            sol.push_back(num_c);
            peso_actual = lim_peso;
        }
    }

    while (rit != contenedores.end()){
        sol.push_back(0);
        ++rit;
    }

    return sol;
}
```


```cpp
list<float> Barco (int lim_peso, list<Conjunto_Contenedor> & contenedores) {
    list<float> sol;
    int peso_actual = 0;
    list<Conjunto_Contenedor>::iterator rit;

    for (rit=contenedores.begin(); rit!=contenedores.end() && lim_peso > peso_actual; ++rit) {
        if ((peso_actual + (*rit).peso) <= lim_peso) {
            sol.push_back(1);
            peso_actual += (*rit).peso;
        }

        else {
            float num_c = (float)(lim_peso-peso_actual)/((*rit).peso);
            sol.push_back(num_c);
            peso_actual = lim_peso;
        }
    }

    while (rit != contenedores.end()){
        sol.push_back(0);
        ++rit;
    }

    return sol;
}
```

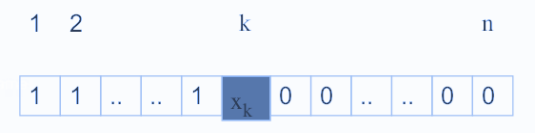
3.3. Estudio de optimalidad (demostración o contraejemplo)

A continuación se presenta la Demostración de la Optimalidad, es decir, vamos a demostrar que el algoritmo siempre encuentra la solución optimal del problema:

- Sea $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$, sea $X = (x_1, x_2, \ldots, x_n)$ la solución generada por "buqueGreedy" para una capacidad del buque M, y sea $Y = (y_1, y_2, \ldots, y_n)$ una solución factible cualquiera.
- Queremos demostrar que: $\sum_{i=1}^{n} (x_i - y_i)p_i \geq 0$
- Si todos los x_i son 1, la solución es claramente optimal (ya que será única solución).



- En otro caso, sea k el menor número tal que $x_k < 1$



- Descomponemos $\sum_{i=1}^{n} (x_i - y_i)p_i$ en tres partes:

$$\sum_{i=1}^{n} (x_i - y_i)p_i = (1)\left[\sum_{i=1}^{k-1} (x_i - y_i)w_i \frac{p_i}{w_i}\right] + (2)\left[(x_k - y_k)w_k \frac{p_k}{w_k}\right] + (3)\left[\sum_{i=k+1}^{n} (x_i - y_i)w_i \frac{p_i}{w_i}\right]$$

  - (1): $antes\ de\ x_k$
  - (2): x_k
  - (3): $después\ de\ x_k$
- Y consideramos cada uno de esos bloques:
  - $\sum_{i=1}^{k-1} (x_i - y_i)w_i \frac{p_i}{w_i} \geq \sum_{i=1}^{k-1} (x_i - y_i)w_i \frac{p_k}{w_k}$
  - $(x_k - y_k)w_k \frac{p_k}{w_k} = (x_k - y_k)w_k \frac{p_k}{w_k}$
  - $\sum_{i=k+1}^{n} (x_i - y_i)w_i \frac{p_i}{w_i} \geq \sum_{i=k+1}^{n} (x_i - y_i)w_i \frac{p_k}{w_k}$
- Finalmente:
  - $\sum_{i=1}^{n} (x_i - y_i)p_i \geq \sum_{i=1}^{n} (x_i - y_i)w_i \frac{p_k}{w_k}$, $es\ decir$, $\sum_{i=1}^{n} (x_i - y_i)p_i \geq \frac{p_k}{w_k} \sum_{i=1}^{n} (x_i - y_i)w_i$

- Vamos a decir que: $W = \frac{p_k}{w_k} \sum_{i=1}^{n} (x_i - y_i) w_i$

- Por hipótesis tenemos que $\sum_{i=1} x_i p_i = M$, pero que $\sum_{i=1} y_i p_i \leq M$

- De forma que para $W > 0$ se cumple $\sum_{i=1}^{n} (x_i - y_i) p_i \geq 0$

3.4. Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para una instancia pequeña

- Tenemos las siguientes variables inicializadas:
  - PesoTotal = 50
  - PesoActual = 0

- Y tenemos los siguientes números de contenedores y los siguientes pesos:
  - P1 = 20 , N1 = 40
  - P2 = 60 , N2 = 10
  - P3 = 10 , N3 = 23
  - P4 = 15 , N3 = 25

- Y tenemos un vector solución.
  - X[4]

- Lo primero que hacemos es obtener las densidades de cada par de peso y número de contenedores y ordenarlas de mayor a menor.
  - D1 = 40/20 = 2
  - D2 = 10/60 = 0.16
  - D3 = 23/10 = 2.3
  - D4 = 25/15 = 1.66
- Ordenamos de la siguiente manera:
  - D3 > D1 > D4 > D2
  - 2.3 > 2 > 1.66 > 0.16

- Comparamos en el orden de densidades que acabamos de crear si el peso de los contenedores de carga correspondientes es mayor que el peso restante (PesoBarco-PesoActual).
  - Si es menor o igual añadimos los contenedores en su totalidad al barco.
  - Si es mayor obtenemos la proporción de contenedores que caben en el barco.
- Tenemos que:
  - Peso restante = 50

- Como 10 (P3) < 50 (Peso restante):
    - X[0] = 1
    - Peso restante = 50-10 = 40

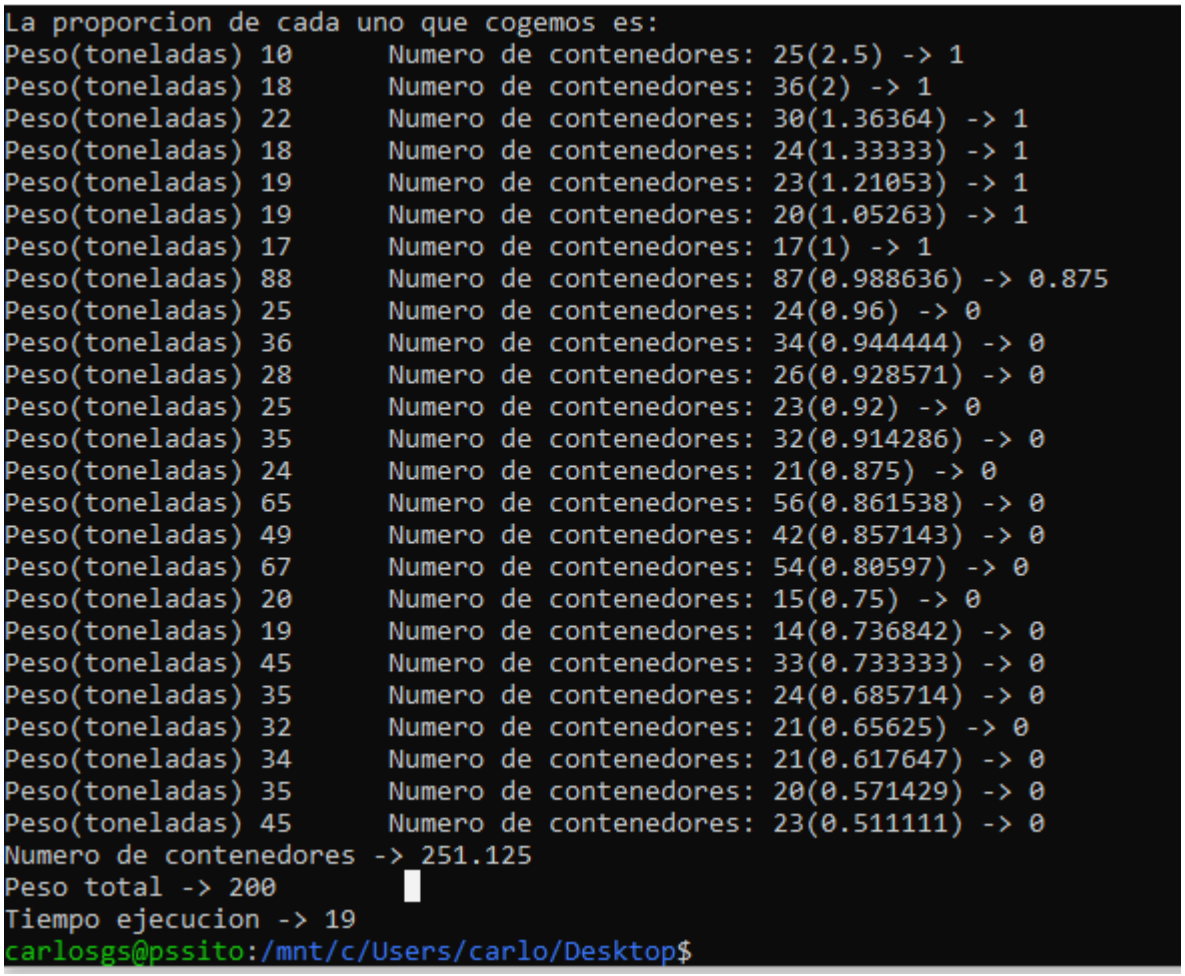
- Como 20 (P1) < 40 (Peso restante):
    - X[1] = 1
    - Peso restante = 40 - 20 = 20

- Como 25 (P4) > 20 (Peso restante):
    - Proporción = 20 (Peso restante) / 25 (P4) = 0.8
    - X[2] = 0.8
    - Peso restante = 20 - (25*0.8) = 0

- Como 60 (P2) > 0 (Peso restante):
    - Proporción = 0(Peso restante)/60(P2) = 0
    - X[3] = 0
    - Peso restante = 0-(60*0) = 0

3.5. Correcto funcionamiento de la implementación

```
struct  Conjunto_Contenedor{
    int num_contenedores;
    int peso;
    float numcont_peso;
    bool operator < (Conjunto_Contenedor & obj1) {
        return (numcont_peso > obj1.numcont_peso);
    }
    friend ostream & operator<< (ostream & os, const Conjunto_Contenedor & o) {
        os << "Peso(toneladas) " << o.peso << '\t' << "Numero de contenedores: " <<         o.num_contenedores    << '(' << o.numcont_peso << ')';
        return os;
    }
    Conjunto_Contenedor (int nc, int p) {
        num_contenedores = nc;
        peso = p;
        numcont_peso = (float)num_contenedores/peso;
    }
};

list<float> Barco (int lim_peso, list<Conjunto_Contenedor> & contenedores) {
    list<float> sol;
    int peso_actual = 0;
    list<Conjunto_Contenedor>::iterator rit;

    for (rit=contenedores.begin(); rit!=contenedores.end() && lim_peso > peso_actual; ++rit) {
        if ((peso_actual + (*rit).peso) <= lim_peso) {
            sol.push_back(1);
            peso_actual += (*rit).peso;
        }

        else {
            float num_c = (float)(lim_peso-peso_actual)/((*rit).peso);
            sol.push_back(num_c);
            peso_actual = lim_peso;
        }
    }

    while (rit != contenedores.end()){
        sol.push_back(0);
        ++rit;
    }

    return sol;
}

float CalculaNum_ContenedoresTotal (list<float> & m, list<Conjunto_Contenedor> & contenedores) {
    float num_c = 0;
    list<float>::iterator it_f;
    list<Conjunto_Contenedor>::iterator rit_o;
    for (it_f = m.begin(), rit_o = contenedores.begin(); it_f != m.end() &&
        rit_o != contenedores.end(); ++it_f, ++rit_o)
        num_c += (*it_f) * (*rit_o).num_contenedores;

    return num_c;
}
float CalculaPesoTotal (list<float> & m, list<Conjunto_Contenedor> & contenedores) {
    float peso = 0;
    list<float>::iterator it_f;
    list<Conjunto_Contenedor>::iterator rit_o;
    for (it_f = m.begin(), rit_o = contenedores.begin(); it_f != m.end() &&
        rit_o != contenedores.end(); ++it_f, ++rit_o)
        peso += (*it_f) * (*rit_o).peso;

    return peso;
}
```



Como podemos observar la ejecución se hace correctamente, los datos mostrados en la ejecución están ordenados según la densidad de los contenedores (numeroContenedores/peso). Este mismo orden es el que usa para introducir los contenedores en el barco. Como podemos observar en el octavo caso (Peso 88, número 87) ya hay dentro del barco 123 kilos y queremos meter ahora 88kg cuando solo quedan 77 libres. En este momento el algoritmo divide 77/88 para conseguir la proporción restante (0.875), lo que multiplica por el número de contenedores (87*0.875 = 76.125) por lo que se añadirán 76 contenedores de los 87.