Enseñanzas Prácticas y de Desarrollo
EPD 9: Grafos

Objetivos

- Conocer el concepto de grafo.
- Implementar y utilizar grafos.
- Recorrer un grafo en profundidad.
- Recorrer un grafo en anchura.

Conceptos

1. Concepto de Grafo

Un grafo es una estructura de datos que permite representar elementos que están relacionados entre si, de tal manera que dicha relación puede contar con distintas características. Pongamos varios ejemplos: un grafo se puede utilizar para representar circuitos, redes de comunicación, planificaciones de tareas, tiempos de vuelo entre ciudades, el plano del metro, redes de comercio, relaciones de comunicación entre miembros de una misma empresa, etc.:

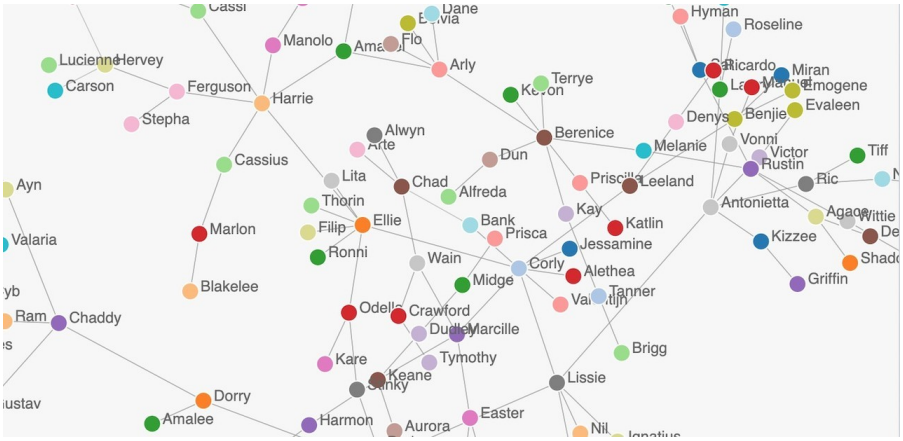


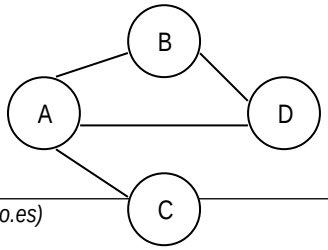
Figura 1: Representación de una red social.

La ADT de un grafo no dirigido incluye por lo menos los siguientes métodos:

- *endVertices*: devuelve los vértices terminales de una arista.
- *opposite*: devuelve el vértice opuesto a la arista dada.
- insertVertex: inserta un vértice.
- insertEdge: inserta una arista.
- removeVertex: elimina el vértice dado y las aristas incidentes.
- removeEdge: elimina la arista especificada.
- vertices: devuelve todos los vértices del grafo.
- edges: devuelve todas las aristas del grafo.

De manera más formal, un grafo se puede definir como la pareja $G=(V,A)$, donde V es un conjunto de elementos o *vértices* y A es un conjunto de pares de vértices relacionados entre si. A cada uno de estos pares de vértices se le llama *arista*. Veamos un ejemplo:

$V=\{A,B,C,D\}$

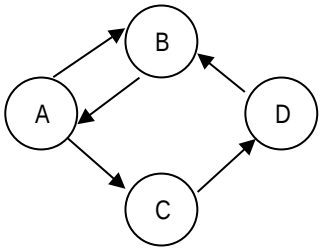


$A=\{AB,AD,AC,BD\}$

En algunos casos es necesario asignar un sentido a las aristas, por ejemplo, si se quiere representar la red de las calles de una ciudad con sus inevitables direcciones únicas. El conjunto de aristas podrá contener en este caso todos los posibles pares ordenados de vértices, con $(A,B)\neq(B,A)$. Veamos otro ejemplo:

$V=\{A,B,C,D\}$

$A=\{AB,BA,AC,CD,DB\}$



Cada vértice almacena información, la cual puede ser del mismo tipo en todos los vértices del grafo o no. Ya hemos visto que las aristas pueden ser dirigidas o no dirigidas. Además, a cada arista se le puede asociar un determinado valor (peso de la arista), pueden existir más de una arista entre dos vértices, pueden representar relacionas jerárquicas (como en el caso de los árboles, que podemos definir como casos particulares de los grafos), etc. Es decir, para poder representar una determinada realidad podemos configurar la estructura de nuestro grafo de la manera más apropiada.

2. Terminología

Veamos algunos términos de interés asociados a los grafos:

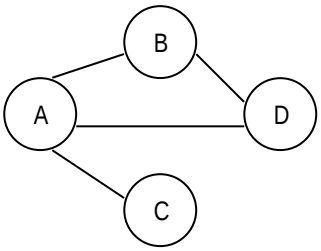
- Vértices adyacentes: unidos por una arista.
- Grado de un vértice: número de vértices adyacentes.
- Vértice Aislado: Es un vértice de grado cero.
- Camino: secuencia de vértices, tal que dos vértices consecutivos son adyacentes.
- Camino simple: aquel que no tiene vértices repetidos.
- Ciclo: camino simple, excepto que el último vértice es el primero.
- Longitud del camino: número de aristas del camino o número de vértices - 1.
- Un grafo es conexo si entre cada dos vértices hay un camino
- Un grafo es completo si existe una arista entre cualquier par de vértices.
- Grafo ponderado es aquel en el que las aristas tienen asociado un peso.
- Un bosque es un grafo sin ciclos

3. Representación de un grafo

Podemos representar un grafo a partir de dos estructuras de datos distintas.

3.1 Representación mediante matriz de adyacencia.

Es la forma más común de representación y la más directa. Sea un grafo $G=(V,A)$, la matriz de adyacencia consiste en una tabla de tamaño $V\times V,$ en la que, en el caso más sencillo, `a[i][j]` tendrá como valor 1 si existe una arista del vértice i al vértice j. En caso contrario, el valor será 0. Cuando se trata de grafos ponderados, en lugar de 1 el valor que tomará será el peso de la arista. Si el grafo es no dirigido hay que asegurarse de que se marca con un 1 (o con el peso) tanto la entrada `a[i][j]` como la entrada `a[j][i]`, puesto que se puede recorrer en ambos sentidos. Veamos un ejemplo:

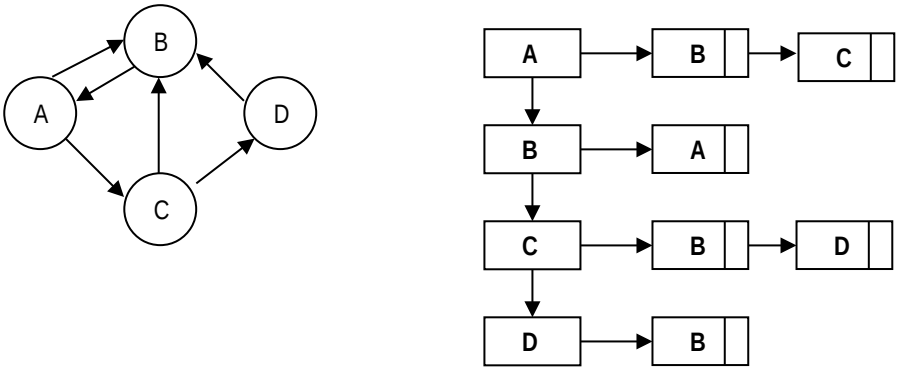


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

En la implementación más avanzada, cómo visto en clase de EB, `a[i][j]` sería un puntero a un objeto "arista".

3.2 Representación mediante lista de adyacencia

Otra forma de representar un grafo es por medio de listas que definen las aristas que conectan los vértices. Lo que se hace es definir una lista enlazada para cada vértice, que contendrá los vértices a los cuales es posible acceder. Es decir, un vértice A tendrá una lista enlazada asociada en la que aparecerá un elemento con una referencia al vértice B si A y B tienen una arista que los une. Obviamente, si el grafo es no dirigido, en la lista enlazada de B aparecerá la correspondiente referencia al vértice A. Por ejemplo:



En clase de EB se han visto también otras dos posibles implementaciones de un grafo mediante listas de adyacencia.

4. Recorridos

4.1 BFS-Breadth First Search

El algoritmo BFS recorre en anchura un grafo. La estrategia sería partir de algún vértice *s*, visitar *s* y, después, visitar cada uno de los vértices adyacentes a *s*. Hay que repetir el proceso para cada nodo adyacente a *s*, siguiendo el orden en que fueron visitados.

```
BFS(grafo G, nodo_fuente s)
  {
     // recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
     // distancia INFINITA y padre de cada nodo NULL
     for u ∈ V[G] do
     {
        estado[u] = NO_VISITADO;
```

```
            distancia[u] = INFINITO; /* distancia infinita si el nodo no es alcanzable */
            padre[u] = NULL;
        }
        estado[s] = VISITADO;
        distancia[s] = 0;
        padre[s] = NULL;
        CrearCola(Q); /* nos aseguramos que la cola está vacía */
        Encolar(Q, s);
        while !vacia(Q) do
        {
            // extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
            u = extraer(Q);
            for  v ∈ adyacencia[u]  do
            {
                if estado[v] == NO_VISITADO then
                {
                     estado[v] = VISITADO;
                     distancia[v] = distancia[u] + 1;
                     padre[v] = u;
                     Encolar(Q, v);
                }
            }
        }
    }
```

4.2 DFS-Depth First Search

Es un algoritmo de recorrido en profundidad de grafo. La estrategia consiste en partir de un vértice determinado *v* y a partir de allí, cuando se visita un nuevo vértice, explorar cada camino que salga de él. Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente. Un camino deja de explorarse cuando se llega a un vértice ya visitado. Si existían vértices no alcanzables desde *v* el recorrido queda incompleto; entonces, se debe seleccionar algún vértice como nuevo vértice de partida, y repetir el proceso. El pseudocódigo en versión recursiva:

```
Algorithm DFS(G, v)
setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)
       if getLabel(e) == UNEXPLORED
              w = opposite(v,e)
              if getLabel(w) == UNEXPLORED
                     setLabel(e, DISCOVERY)
                     DFS(G, w)
              else
                     setLabel(e, BACK)
```

Bibliografía Básica

- *Introduction to Algorithms*. Thomas H. Cormen. Capítulo 12, Página 253.
- http://web.cs.wpi.edu/~cs2102/b10/Lectures/graphs-intro.html
- http://www.dreamincode.net/forums/topic/377473-graph-data-structure-tutorial/

Experimentos

E1. En esta practica vamos a usar la implementación de grafo proporcionada en la librería `ED-DataStructures.jar`, donde se proporciona una implementación de grafo basada en una lista de adyacencia.
Esta librería proporciona las siguientes interfaces:

```
public interface Vertex<V extends Object> {

    public V getElement();
}

public interface Edge<E extends Object> {

    public E getElement();
}

public interface Graph<V extends Object, E extends Object> {

    public int numVertices();

    public int numEdges();

    public Iterable<Vertex<V>> vertices();

    public Iterable<Edge<E>> edges();

    public int outDegree(Vertex<V> vertex) throws IllegalArgumentException;

    public int inDegree(Vertex<V> vertex) throws IllegalArgumentException;

     public Iterable<Edge<E>> outgoingEdges(Vertex<V> vertex) throws
                      IllegalArgumentException;

    public Iterable<Edge<E>> incomingEdges(Vertex<V> vertex) throws
                      IllegalArgumentException;

    public Edge<E> getEdge(Vertex<V> vertex, Vertex<V> vertex1) throws
               IllegalArgumentException;

    public Vertex<V>[] endVertices(Edge<E> edge) throws
                      IllegalArgumentException;

    public Vertex<V> opposite(Vertex<V> vertex, Edge<E> edge) throws
                      IllegalArgumentException;

    public Vertex<V> insertVertex(V v);

    public Edge<E> insertEdge(Vertex<V> vertex, Vertex<V> vertex1, E e) throws
               IllegalArgumentException;

    public void removeVertex(Vertex<V> vertex) throws IllegalArgumentException;

    public void removeEdge(Edge<E> edge) throws IllegalArgumentException;
}
```

Indicar cuales son las funcionalidades de cada método de la interfaz `Graph`.

E4. Importe la librería `ED-DataStructures.jar`. La librería proporciona la implementación de varias estructuras de datos, entre ellas los grafos (que implementan la interfaz `Graph`, ver el archivo `Graph.html` incluido en el material de la EPD). Estudie el siguiente código

```
import datastructures.Graph;
import datastructures.AdjacencyMapGraph;
import datastructures.Edge;
import datastructures.Vertex;
import java.util.HashMap;
import java.util.TreeSet;

public class EPDGrafos {

public static void main(String[] args) {
```

```
        //declaro un grafo no direccionado
        Graph<String, Integer> grafoNoDireccionado = new AdjacencyMapGraph<>(false);
        //declaro un grafo direccionado
        Graph<String, Integer> grafoDireccionado = new AdjacencyMapGraph<>(false);

        Vertex<String> v1 = grafoNoDireccionado.insertVertex("V1");
        Vertex<String> v2 = grafoNoDireccionado.insertVertex("V2");
        Vertex<String> v3 = grafoNoDireccionado.insertVertex("V3");
        Vertex<String> v4 = grafoNoDireccionado.insertVertex("V4");

        grafoNoDireccionado.insertEdge(v1, v2, 1);
        grafoNoDireccionado.insertEdge(v1, v3, 1);
        grafoNoDireccionado.insertEdge(v2, v3, 1);
        grafoNoDireccionado.insertEdge(v2, v4, 1);
        grafoNoDireccionado.insertEdge(v3, v4, 1);

        System.out.println("Numero vertices: " + grafoNoDireccionado.numVertices()
                           + " numero aristas: " + grafoNoDireccionado.numEdges());
        print(grafoNoDireccionado);
}
```

Observe cómo se pueden insertar vértices y aristas. ¿Qué grafo se ha construido? ¿Cómo se podría darle un peso a las aristas?

E5. Estudie los siguientes métodos estaticos:

```
    /**
     * Helper routine to get a Vertex (Position) from a string naming the vertex
     */
    public static Vertex<String> getVertex(String vertexLabel, Graph<String, Integer> sGraph) {
        // Go through vertex list to find vertex
        for (Vertex<String> vs : sGraph.vertices()) {
            if (vs.getElement().equals(vertexLabel)) {
                return vs;
            }
        }
        return null;
    }

    /**
     * Printing all the vertices in the list, followed by printing all the edges
     */
    public static void print(Graph<String, Integer> sGraph) {
        System.out.println("Vertices: " + sGraph.numVertices()
                + " Edges: " + sGraph.numEdges());

        //imprimir todos los vertices
        for (Vertex<String> vs : sGraph.vertices()) {
            System.out.println(vs.getElement());
        }
        //imprimir todas las aristas
        for (Edge<Integer> es : sGraph.edges()) {
            System.out.println(es.getElement());
        }
    }

/**
     * Constructs a graph from an array of array strings.
     *
     * An edge can be specified as { "SFO", "LAX" }, in which case edge is
     * created with default edge value of 1, or as { "SFO", "LAX", "337" }, in
     * which case the third entry should be a string that will be converted to
     * an integral value.
     */
    public static Graph<String, Integer> graphFromEdgelist(String[][] edges, boolean directed) {
        Graph<String, Integer> g = new AdjacencyMapGraph<>(directed);

        // first pass to get sorted set of vertex labels
        TreeSet<String> labels = new TreeSet<>();
        for (String[] edge : edges) {
            labels.add(edge[0]);
            labels.add(edge[1]);
        }
```

```
        // now create vertices (in alphabetical order)
        HashMap<String, Vertex<String>> verts = new HashMap<>();
        for (String label : labels) {
            verts.put(label, g.insertVertex(label));
        }

        // now add edges to the graph
        for (String[] edge : edges) {
            Integer cost = (edge.length == 2 ? 1 : Integer.parseInt(edge[2]));
            g.insertEdge(verts.get(edge[0]), verts.get(edge[1]), cost);

        }
        return g;
    }
```

En el código se utilizan las siguientes funciones:

- Graph<String, Integer> graphFromEdgelist(String[][] edges, boolean directed) estre método construye y devuelve un grafo desde una lista de aristas proporcionada, con el segundo parametro podemos indicar si el grafo es direccionado o no;
- Vertex<String> getVertex(String vertexLabel, Graph<String, Integer> sGraph) proporciona un objeto Vertex (vértice del grafo) desde su etiqueta (vertexLabel). Por ejemplo: Vertex<String> vertex = getVertex(labelV, graph);
- print(Graph<String, Integer> sGraph) imprime el grafo, primero todos los vértices seguidos por todas las artistas;

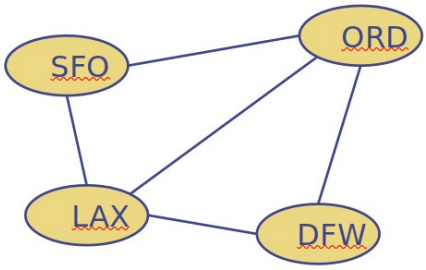
Las clases vértices y aristas implementan un método (getElement) que devuelve el elemento almacenado en el vértice o en la arista, en este caso un String o un Integer, respectivamente.

E6. ¿En los códigos de BFS y DFS cómo se podría implementar las etiquetas de los vértices y de las aristas?

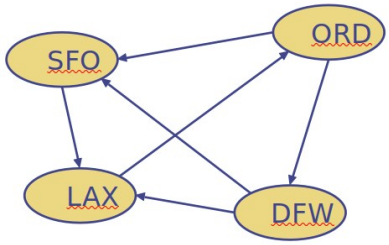
E7. En el código del BFS, ¿qué significa esto: `v ∈ adyacencia[u]` y cómo se puede implementar?

Ejercicios

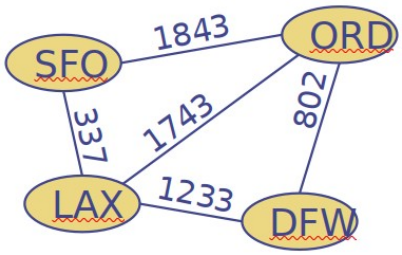
EJ1. (20 mins) Usando la librería proporcionada, crear un grafo no direccionado no ponderado como el de la figura



EJ2. (15 mins) Usando la librería proporcionada, crear un grafo direccionado no ponderado como el de la figura:



EJ3. (15 mins) Usando la librería proporcionada, crear un grafo ponderado como el de la figura:



en este caso la lista de aritas tiene que ser de elementos tipo { "SFO", "LAX", "337" }

EJ4. (20 mins) Implemente el método estatico `areAdjacent` que devuelve un valor booleano indicando si dos vértices son o no adyacentes.

EJ5. (30 mins) Implemente los siguientes métodos estaticos:

- List<Edge<Integer>> buildCompleteEdgeList(Graph<String, Integer> graph) devuelve una lista que contiene todas las artistas del grafo pasado como parámetro;
- List<Vertex<String>> buildCompleteVertexList(Graph<String, Integer> graph) devuelve una lista que contiene todos los vértices del grafo pasado como parámetro;

EJ6. (20 mins) Usando los grafos creados en la EPD anterior, implemente el recorrido DFS

Problemas

P1. (20 mins.) Usando los grafos creados en la EPD anterior, implemente el recorrido BFS.

P2. (60 mins.) Implementar un grafo mediante una matriz de adyacencia que implemente la interfaz `Graph`.

Pista 1: La matriz de adyacencia será una tabla bidimensional cuadrada con tantas filas como vértices. En la implementación más sencilla, cada elemento de la tabla podrá tomar los valores 1 si dos vértices son adyacentes y 0 en cualquier otro caso. En un implementación más avanzada cada elemento será una referencia a un objeto de tipo `Edge`.

Pista 2: Los vértices y las aristas pueden almacenarse en colecciones.

P2. (40 mins.) Implemente el algoritmo de Dijsktra. Considere la implementación basada en matriz de adyacencia. En este caso se debe usar un grafo ponderado.

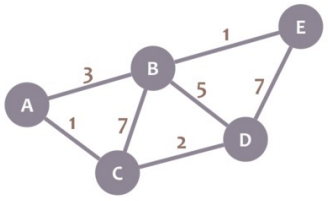
Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene.

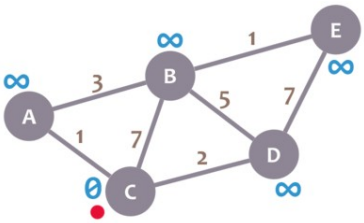
El algoritmo sigue los siguientes pasos:

1. Marca el nodo inicial que elegiste con una distancia actual de 0 y el resto con infinito.
2. Establece el nodo no visitado con la menor distancia actual como el nodo actual A.
3. Para cada vecino V de tu nodo actual A: suma la distancia actual de A con el peso de la arista que conecta a A con V. Si el resultado es menor que la distancia actual de V, establécelo como la nueva distancia actual de V.
4. Marca el nodo actual A como visitado.
5. Si hay nodos no visitados, ve al paso 2.

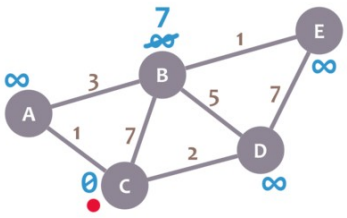
Por ejemplo, calcularemos la distancia más corta entre el nodo C y los demás nodos del grafo:



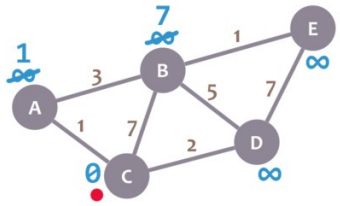
Durante la ejecución del algoritmo, iremos marcando cada nodo con su distancia mínima al nodo C (nuestro nodo elegido). Para el nodo C, esta distancia es 0. Para el resto de nodos, como todavía no conocemos esa distancia mínima, empieza siendo infinita (∞):



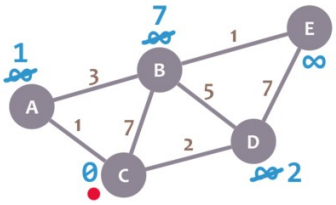
También tendremos un nodo actual. Inicialmente, el nodo actual será C (nuestro nodo elegido). En la imagen, marcaremos el nodo actual con un punto rojo. Ahora, revisaremos los vecinos de nuestro nodo actual (A, B y D) en cualquier orden. Empecemos con B. Sumamos la mínima distancia del nodo actual (en este caso, 0) con el peso de la arista que conecta al nodo actual con B (en este caso, 7), y obtenemos 0 + 7 = 7. Comparamos ese valor con la mínima distancia de B (infinito); el valor más pequeño es el que queda como la distancia mínima de B (en este caso, 7 es menos que infinito):



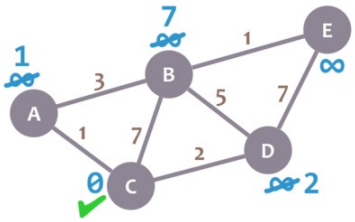
Bien. Ahora revisaremos al vecino A. Sumamos 0 (la distancia mínima de C, nuestro nodo actual) con 1 (el peso de la arista que conecta nuestro nodo actual con A) para obtener 1. Comparamos ese 1 con la mínima distancia de A (infinito) y dejamos el menor valor:



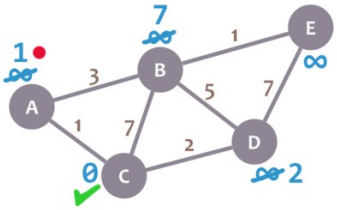
Repetimos el procedimiento para D:



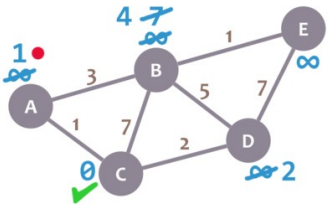
Hemos revisado todos los vecinos de C. Por ello, lo marcamos como visitado. Representemos a los nodos visitados con una marca de verificación verde:



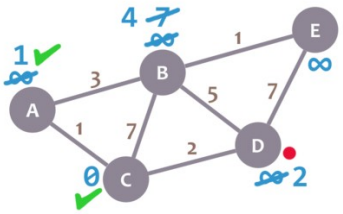
Ahora debemos seleccionar un nuevo nodo actual. Ese nodo debe ser el nodo no visitado con la menor distancia mínima, es decir, el nodo con el menor número y sin marca de verificación verde. En este caso, ese nodo es A. Vamos a marcarlo con el punto rojo:



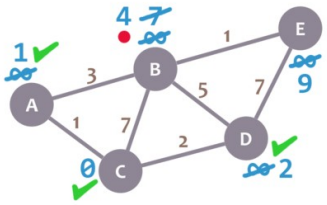
Ahora, repetimos el algoritmo. Revisamos los vecinos de nuestro nodo actual, ignorando los visitados. Esto significa que solo revisaremos B. Para B, sumamos 1 (la distancia mínima de A, nuestro nodo actual) con 3 (el peso de la arista conectando a A con B) para obtener 4. Comparamos ese 4 con la distancia mínima de B (7) y dejamos el menor valor: 4.



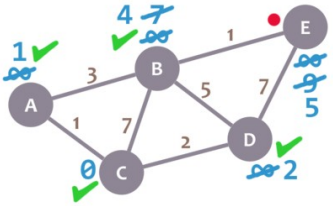
Después, marcamos A como visitado y elegimos un nuevo nodo: D, que es el nodo no visitado con la menor distancia mínima.



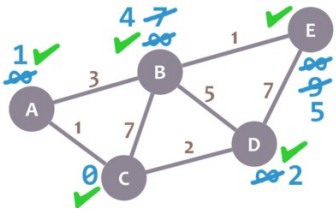
Repetimos el algoritmo de nuevo. Esta vez, revisamos B y E. Para B, obtenemos 2 + 5 = 7. Comparamos ese valor con la distancia mínima de B (4) y dejamos el menor valor (4). Para E, obtenemos 2 + 7 = 9, lo comparamos con la distancia mínima de E (infinito) y dejamos el valor menor (9). Marcamos D como visitado y establecemos nuestro nodo actual en B.



Sólo debemos verificar E. 4 + 1 = 5, que es menos que la distancia mínima de E (9), así que dejamos el 5. Marcamos B como visitado y establecemos E como el nodo actual.



E no tiene vecinos no visitados, así que no verificamos nada. Lo marcamos como visitado.



Como no hay nodos no visitados, finaliza la ejecución del algoritmo. La distancia mínima de cada nodo ahora representa la mínima distancia entre ese nodo y el nodo C (el nodo que elegimos como nodo inicial).

P3. (40 mins.) Implemente el algoritmo de Kruskal para encontrar el árbol de recubrimiento mínimo de un grafo conexo y ponderado. Considere la implementación basada en matriz de adyacencia.

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca
un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del

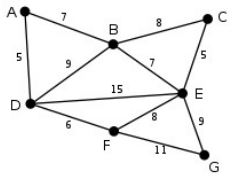
árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).
El pseudocódigo del algoritmo es:

```
función Kruskal(G)
   Para cada v en V[G] hacer
     Nuevo conjunto C(v) ← {v}.
   Nuevo heap Q que contiene todas las aristas de G, ordenando por su peso.
   Defino un arbol T ← Ø
   // n es el número total de vértices
   Mientras T tenga menos de n-1 aristas y !Q.vacío() hacer
     (u,v) ← Q.sacarMin()
     // previene ciclos en T. agrega (u,v) si u y v están diferentes componentes en el conjunto.
     // Nótese que C(u) devuelve la componente a la que pertenece u.
     Si C(v) ≠ C(u) hacer
       Agregar arista (v,u) a  T.
       Merge C(v) y C(u) en el conjunto
   Responder arbol T
```

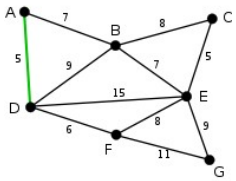
Resumiendo, se selecciona de entre todas las aristas restantes, la de menor peso siempre que no cree ningún ciclo. Se repite el paso anterior hasta que se hayan seleccionado |V| - 1 aristas, siendo V el número de vértices.

Ejemplo:

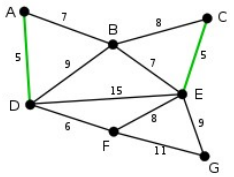
Este es el grafo inicial. Los números indican el peso de las aristas. Se elige de manera aleatoria uno de los vértices que será el vértice de partida. N este caso se ha elegido el vértice D.



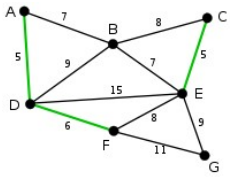
Se selecciona, de entre todas las aristas restantes, la de menor siempre que no cree ningún ciclo. Las aristas de menor peso son las aristas AD y CE (5). Se ha seleccionado de manera aleatoria la arista AD.



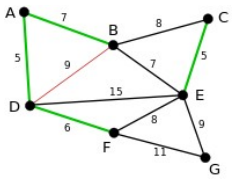
Se selecciona, de entre todas las aristas restantes, la de menor siempre que no cree ningún ciclo. Ésta es la arista CE.



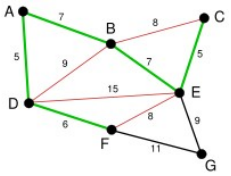
Seleccionamos DF, con peso 6, que es la siguiente arista de menor peso que no forma ciclos.



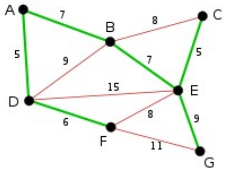
De las aristas restantes, las de menor peso son las aristas AB y BE, de peso 7. AB se elige aleatoriamente, y se añade al conjunto de las aristas seleccionadas. De este modo, la arista DB no puede ser seleccionada ya que formaría el ciclo ADB. Por tanto la marcamos en rojo.



Siguiendo el proceso seleccionamos la arista BE con peso 7. Además marcamos en rojo las aristas BC, DE y FE ya que formarían los ciclos BCE, DEBA, FEBAD respectivamente.



Por último se selecciona la arista EG de peso 9. Como han sido seleccionadas un número de aristas igual al número de vértices menos uno, el proceso ha terminado. Se ha obtenido el árbol de expansión mínima con un peso de 39.



P5. (40 mins.) Implementar el siguiente grafo mediante lista de adyacencia que implemente la interfaz `Graph`: