Librería de Grafos para C++

Hugo Zúñiga C.	A96988
Ernesto Céspedes M.	AXXXXX
Diego Álvarez A.	AXXXXX

Capítulo 1

Marco Teórico

1.1. Introducción a los Grafos

Un grafo es la representación abstracta de un conjunto de objetos, los cuales están conectados a través de enlaces. Los objetos interconectados se representan mediante una estructura denominada vértice y a los enlaces se les denomina bordes. La representación gráfica de un grafo se hace mediante un conjunto de puntos (vértices) unidos por un conjunto de líneas que representan los bordes, un ejemplo de esto se observa en la figura 1.1.

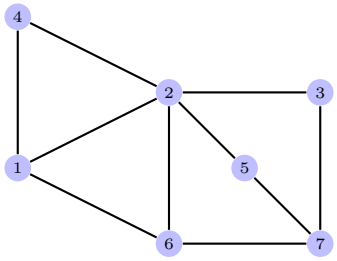

Figura 1.1: Representación Gráfica de un Grafo

Los grafos son estructuras de datos que tienen aplicaciones en muchas áreas del desarrollo humano, algunas de ellas son:

- *Desarrollo de Circuitos Integrados:* Los circuitos electrónicos constan de una gran cantidad de componentes, los cuales están interconectados mediante líneas de metal. Debido a la complexidad de los diseños es necesario contar con una plataforma la cual permita hacer solicitudes sobre la interconexión de los componentes.
- *Transacciones Comerciales:* Las instituciones financieras compran y venden acciones en la bolsa. En este caso el grafo permite representar la transferencia de dinero y bienes entre instituciones o instituciones y compradores.

- *Redes de Computadoras:* Las redes de computadoras consisten de un conjunto de computadoras interconectadas, las cuales envían y reciben mensajes de varios tipos. En este caso el grafo representa los nodos del sistema y las vías de comunicación que hay entre ellas.

1.1.1. Tipos de Grafos

Dentro del sistema de grafos existen diversas subclasificaciones, las cuales representan las distintas estructuras que se pueden obtener con el objetivo de representar las relaciones entre los objetos. Las clasificaciones en las cuales se basa este proyecto son:

- Grafos no Dirigidos
- Grafos Dirigidos
- Grafos con Peso

Grafos no Dirigidos

Los grafos no dirigidos son estruturas de datos que representan un sistema en el cual los enlaces permiten la conexión bidireccional entre los vértices, la representación gráfica de este tipo se observa en la figura 1.1. Asimismo dentr de la estructura de datos, existen subestructuras que permiten encontrar propiedades importantes de los grafos.

La primera de estas subestructuras es el «path» o camino, el cual es una secuencia de vértices conectados por enlaces. De esta manera existen caminos simples, los cuales cuentan con todos los vértices distintos y existen caminos cíclicos, en los cuales se repite el primer y último vértice. Esta definición es una de las más importantes en la teoría de grafos, ya que uno de los parámetros de mayor interés de un grafo es la capacidad de encontrar un camino que interconecte dos vértices y determinar si es el óptimo.

Otra estructura importante relacionada con los grafos es el árbol, el cual es un subgrafo que contiene todos los vértices pertenecientes al grafo. La importancia de esta estructura radica en su utilización en los algoritmos de procesamiento de grafos, esta representación permite obtener información importante acerca de la conectividad y estructura del grafo.

Grafos Dirigidos

La singularidad que tienen los grafos dirigidos es que los enlaces entre los vértices son unidireccionales a diferencia de los grafos no dirigidos. Esto implica que los enlaces pueden ser atravesados en una dirección únicamente.

De esta manera existen también la definición de camino simple y camino cíclico para los grafos dirigidos. La diferencia principal radica en la limitación de que el hecho de que

exista un camino que comunique dos vértices en una dirección no implica que exista un camino que los comunique de manera inversa.

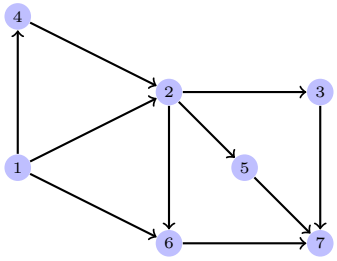

Figura 1.2: Representación Gráfica del Grafo Dirigido

Grafos con Peso

En realidad los grafos con peso se pueden basar en cualquiera de las dos estruturas discutidas anteriormente, la diferencia principal es que se le agrega información a los enlaces, de manera que se puedan adaptar al modelado de sistemas más amplios.

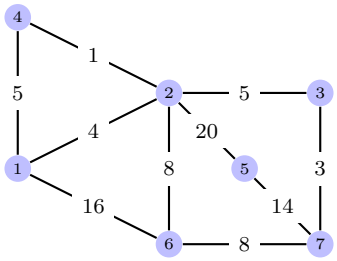

Figura 1.3: Representación Gráfica del Grafo con Peso

1.1.2. Estructura de Datos

Dentro del estudio de la eficiencia de los algoritmos, se toman en cuenta dos parámetros el tiempo y espacio de ejecución. El primero hace referencia al tiempo que toma completar el procesamiento del algoritmo, mientras que el segundo trata sobre la cantidad de espacio que ocupan todas las estructuras y funciones que son necesarias para la ejecución del algoritmo.

En consideración de estos parámetros es importante determinar una estructura de datos para la representación de grafos, la cual permita obtener la flexibilidad necesaria de manera que se optimicen tanto el tiempo de ejecución de los algoritmos como el espacio de almacenamiento. Para alcanzar este fin se estudió tres propuestas diferentes.

- Matriz de Adyacencia

- Arreglo de Enlaces
- Lista de Adyacencias

Matriz de Adyacencia

La matriz de adyacencia es una configuración, en la cual el grafo se representa mediante una matriz de N filas y N columnas, donde N representa la cantidad de vértices presentes en el grafo. Para reprecentar los enlaces entre vértices, se asigna un valor booleano a la celda representado por la fila y la columna de los nodos conectados, así por ejemplo el siguiente es un ejemplo de una matriz de adyacencia de 4 vértices.

	1	2	3	4
1	T	T	F	T
2	T	T	T	F
3	F	T	T	T
4	T	F	T	T

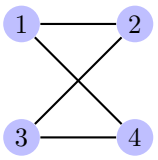

Figura 1.4: Representación de un grafo mediante una matriz de adyacencia

La ventaja de la configuración es que permite una rápida inserción de nuevas conexiones, sin embargo el espacio que ocupa es el cuadrado de la cantidad de vértices, por lo cual esta opción es descartable.

Arreglo de Enlaces

El arreglo de enlaces es un sistema, el cual parte de un tipo de dato denominado Enlace, el cual contiene dos variables instanciadas (los vértices conectados). De esta manera es sencillo crear nuevas uniones, sin embargo el tratamiento de los algoritmos para este tipo de organización es significativamente más difícil, ya que hay que revisar todos las instancias de «Enlace» para poder extraer información relevante del grafo.

Lista de Adyacencias

La lista de adyacencias es una configuración en la cual se crea una lista de todos los vértices que están conectados a un nodo en particular, además se crea una lista con los punteros a las listas de adyacencias. Una representación de una lista de adyacencias se observa en la figura 1.5

Esta estructura tiene una complejidad espacial de $E+V$, lo cual es menor que cualquiera de las dos opciones presentadas anteriormente, asimismo el tiempo computacional de procesamiento debe ser menor debido a que en caso de que se desee iterar a través delos

1	2	4
2	1	3
3	2	4
4	1	3

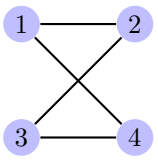

Figura 1.5: Representación de la lista de adyacencias

nodos adyacentes, sólo se debe revisar los nodos adyacentes, lo cual reduce el tiempo de procesamiento.

1.2. Algoritmos de Procesamiento de Grafos

1.2.1. Algoritmos básicos:

Depth First Search

El algoritmo depth fist search es un algoritmo recursivo, en el cual se siguen los enlaces en el algoritmo para ir revelando nuevos vértices que están directamente conectados a un vértice origen. Este algoritmo tiene diversas aplicaciones, como la determinación de que haya un camino que conecte dos vértices distintos, también se puede utilizar para separar subgrafos en un grafo determinado.

Dependiendo de la aplicación la implementación del depth first search se lleva a cabo de una manera diferente. Para el caso más sencillo que es para determinar conectividad se utiliza una lista para indicar si el vértice fue visitado anteriormente o no. Con base en esta lista se puede determinar si el sistema está conectado (Todos los vértices fueron visitados) o no. Un ejemplo del algoritmo de Depth First Search se presenta a continuación.

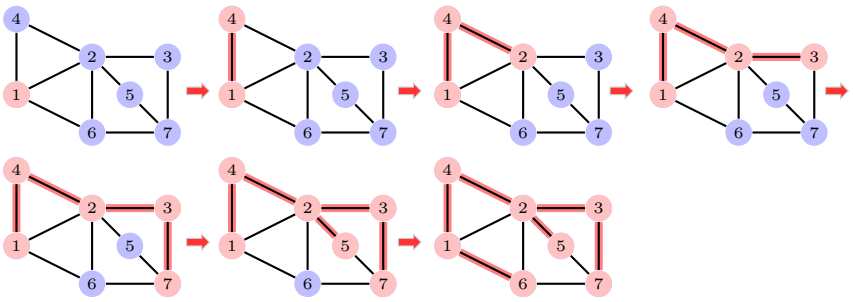

Figura 1.6: Ejemplo de la aplicación del DFS en grafos

Breadth First Search

El algoritmo Breadth First Search se utiliza para analizar todos los vértices de un grafo no dirigido, este algoritmo a diferencia del Depth First Search itera sobre los vértice en orden de nivel desde el origen, esto quiere decir que visita primero los vértices conectados directamente al origen, luego visita los vértices que están conectados a estos y así sucesivamente. Una de las ventajas que tiene el Breadth First Search es que permite obtener un Árbol del Camino más corto para un grafo no dirigido.

La implementación del algoritmo hace uso de una cola la cual alberga los vértices como deben irse revisando, para esto lo que se hace es que se introducen a la cola los vértices que estén conectados al vértice que se está analizando y que no habían sido visitados anteriormente. El algoritmo en este caso es iterativo y se revisan todos los vértices hasta que la cola esté vacía.

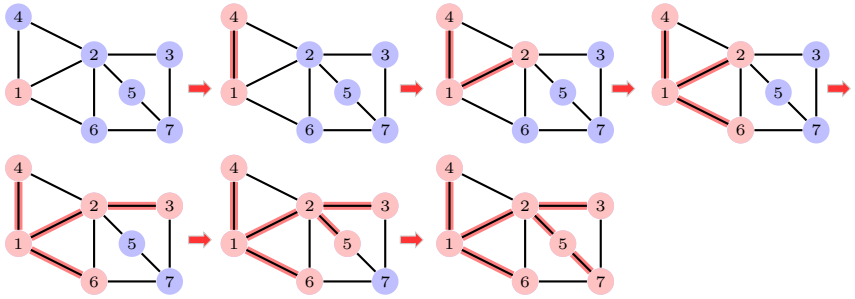

Figura 1.7: Ejemplo de la aplicación del BFS en grafos

1.2.2. Algoritmos enfocados en grafos dirigidos

Para el procesamiento de grafos dirigidos existen diversos algoritmos que permiten extraer información relevante de la estructura. Algunos de estos son:

- *Alcanzabilidad*: ¿Es posible llegar desde un vértice a a un vértice b?
- *Detección de Ciclos*: ¿Existe un camino cerrado dentro del grafo?
- *Ordenamiento Topológico*: ¿Se puede ordenar el sistema, de manera que ningún vértice que esté en la lista de adyacencia de otro se encuentre más a la izquierda que éste?

Alcanzabilidad

Para resolver el problema de saber si existe un camino que comunique dos vértices distintos, es necesario utilizar el algoritmo de depth first search, ya que este provee una manera sencilla de construir un árbol, el cual incluya todos los nodos alcanzables desde

un origen determinado. Con base en lo mencionado en la sección anterior (Algoritmos Básicos) se tiene que la aplicación de este algoritmo para los grafos dirigidos, tiene una sucesión como la que se muestra en la figura 1.7

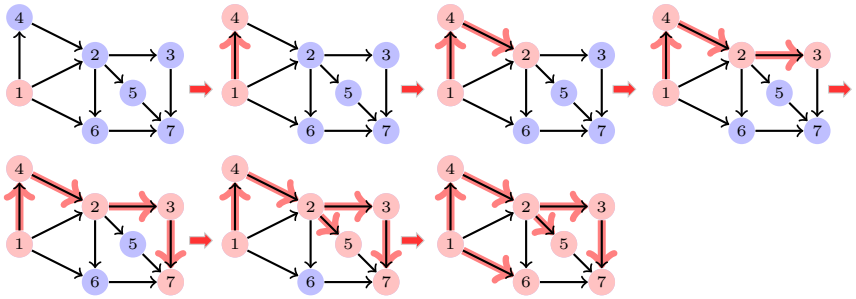

Figura 1.8: Ejemplo de la aplicación del DFS en grafos dirigidos

Como se aprecia en el ejemplo, la aplicación del DFS a los grafos dirigidos permite encontrar todos los vértices del grafos que tienen un camino desde el origen. El pseudocódigo se presenta en el algoritmo 1.

Algorithm 1 Algoritmo Depth First Search

```
procedure DFS(G, v)        ▷ The Depth First Search Algorithm for Directed Graphs
    for vertex in adj[v] do
        if vertex not marked then
            mark vertex
            DFS(G, vertex)
        end if
    end for
end procedure
```

Como se aprecia el DFS para los grafos dirigidos es muy similar al utilizado para los grafos no dirigidos, esto se debe a que la estructura de ambos es muy similar, sin embargo el tiempo de ejecución de éste es menor debido a que en la lista de adyacencia de cada vértice se presenta únicamente los vértices a los que apunta.

Detección de Ciclos

La presencia de ciclos dentro de un grafo dirigido es una característica importante de conocer, ya que ésta afecta a su vez muchos de los algoritmos que se destacan más adelante, como el ordenamiento topológico y la búsqueda del camino más corto. El algoritmo utilizado para la detección de ciclos es muy similar al DFS, la diferencia más importante es que para la búsqueda de ciclos es que se debe mantener una lista de los nodos que ya han sido revisados. La implementación que se hizo del algoritmo requiere un ¡Queue¿expresado como una lista de booleanos, con un tamaño equivalente a la cantidad de vértices en el

grafo, el cual indica si un nodo ya fue revisado anteriormente por la misma iteración del DFS; además se utilizó una lista para albergar la conexión que conecta al n-simo vértice al árbol creado por el DFS.

Algorithm 2 Algoritmo para Busqueda de Ciclos

```
procedure DFS(G, v)        ▷ The Depth First Search Algorithm for Cycle Detection
    for all vertex ∈ adj[v] do
        if vertex in queue then
            e ← edgeTo[vertex]
            while e ≠ vertex do
                cycle ← e
                e ← edgeTo[e]
            end while
        else if vertex not marked then
            mark vertex
            Add vertex to queue
            DFS(G, vertex)
        end if
    end for
    Remove vertex from queue
end procedure
```

Ordenamiento Topológico

El ordenamiento topológico de un grafo *G = (V,E)* es un ordenamiento lineal de los vértices, tal que si *G* contiene un borde *(u,v)*, entonces *u* aparece antes que v en el ordenamiento. El ordenamiento topológico puede ser comprendido como un ordenamiento de los vértices a lo largo de una línea horizontal, de manera que todos las conexiones vayan de izquierda a derecha.

La implementación del Ordenamiento topológico, se puede hacer de una manera bastante sencilla utilizando el DFS, el algoritmo básico para cualquier implementación es:

- Llamar al DFS(G)
- Cuando el procedimiento termina para cada vértice, insertarlo al frente de una lista enlazada.
- Retornar la lista enlazada con los vértices.

1.2.3. Algoritmos para la búsqueda de árboles de expansión mínima

Un grafo de aristas con peso o edge-weighted graph es un modelo de grafo donde se asocia pesos o costos con cada arista. Se pueden aplicar en muchos modelos naturales, por

ejemplo un mapa de rutas aéreas, donde los pesos pueden representar distancias, además por ejemplo en el diseño de un circuito electrónico a veces se necesita hacer que los pines de muchos componentes sean eléctricamente equivalentes, cableandolos juntos en un solo nodo. Para conectar un grupo de n pines, se puede usar un arreglo de n-1 cables, cada uno conectando 2 pines. De todos estos arreglos, el que utiliza la menor cantidad de cable es casi siempre el más deseado.

Este problema de cableado se puede modelar con un grafo conectado, indirecto G = (V,E), donde V es un grupo de nodos o pines, E representa un grupo de posibles interconecciones entre pares de nodos, y por cada arista $(u,v) \in$ E, tenemos un peso $\omega(u,v)$ especificando el costo(cantidad de cable requerido) para conectar u y v. Luego lo que se desea es encontrar un subgrupo acíclico T$\subseteq$E que conecte todos los vértices y cuyo peso total:

$$\omega(T) = \sum_{(u,v)\in T} \omega(u,v) \tag{1.1}$$

sea minimizado. Ya que T es acíclico y conecta todos los vértices, este debe formar un árbol, al cual se le llama árbol de expansión mínima, ya que se expande por todos los vértices del grafo G como se muestra en la figura 1.8.

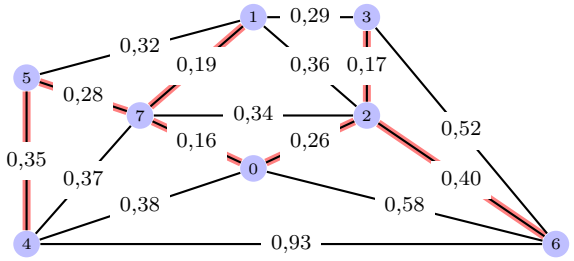

Figura 1.9: Representación Gráfica de un MST

En esta librería se implementaron dos algoritmos para resolver el problema de los arboles de expansión mínima: el de Kruskal y el de Prim. Cada uno de estos algoritmos es una variación del algoritmo de greedy, el cual se definirá más adelante.

A continuación se definirá como crece un árbol de mínima expansión. Asumiendo que que tenemos un grafo, conectado e indirecto G = (V,E) con un peso $\omega : E \longrightarrow \Re$, y queremos encontrar el árbol de expansión mínima para G. La estrategia de greedy sigue el siguiente método genérico, el cual aumenta el MST(por sus siglas en inglés) una arista a la vez. Este maneja un conjunto de aristas A, manteniendo el siguiente ciclo invariante: **Antes de cada iteración, A es un subconjunto de algún árbol de expansión mínima.**

En cada paso, se determina una arista (u,v) la cual puede ser agregada a A sin violar la declaración anterior, en el sentido que A$\cup(u,v)$ también es un subconjunto de un árbol de

mínima expansión. A esta arista le podemos llamar *arista segura* para A, ya que podemos agregarla a A mientras que se mantiene el ciclo invariante.

Seudocódigo del MST(G,ω) Genérico:
1- A=$\emptyset$
2- **while** (A no forma un arbol de expansion)
3- encuentre una arista (u, v) que sea *segura* para A
4- A=A$\cup(u, v)$
5- **return** A

Utilizamos este ciclo invariante de la siguiente forma:
Inicialización: La línea 1 muestra las condiciones iniciales, en las cuales el conjunto A trivialmente satisface la condición del ciclo invariante.
Mantenimiento: El ciclo se mantiene invariante, es decir cumpliendo la iteración en las líneas 2 a 4, agregando solo aristas seguras a A.
Terminación: Todos las aristas agregadas a A están en un árbol de expansión mínima, por lo tanto el conjunto A devuelto en la línea 5 debe ser el árbol de expansión mínima.

Obviamente la parte truculenta se encuentra tratando de encontrar la arista que es segura, en la línea 3. Una debe existir, ya que desde que la línea 3 es ejecutada, la invarianza dicta que hay un árbol de expansión T el cual A$\subseteq$T. Dentro del ciclo **while**, A debe ser un subconjunto de T, y por lo tanto debe haber una arista $(u, v) \in$ T, tal que $(u, v) \notin$ A y (u, v) es seguro para A.

Para definir una arista segura, primero debemos definir ciertos conceptos. Un **corte** (S, V-S) de un grafo indirecto G = (V,E) es una partición de V. La figura 1.9 ilustra esta idea, donde tenemos un conjunto de vertices rojos y azules. Decimos que un corte **respeta** un conjuto A de aristas, si ninguna arista en A cruza el corte. Una arista $(u, v) \in$ E **cruza** el corte (S, V-S) si uno de sus extremos está en S y el otro en V-S, los de color rojo. Una arista es una **arista liviana** cruzando un corte, si su peso es el mínimo de todas las aristas que también lo cruzan.

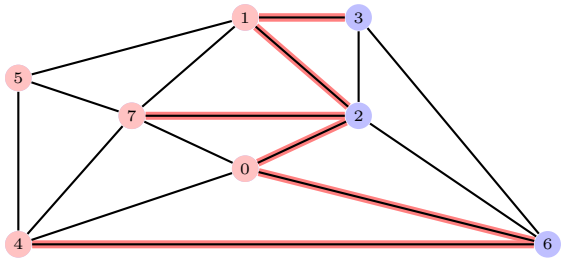

Figura 1.10: Representación Gráfica de un corte

Una regla para reconocer una arista segura es la siguiente: Sea G = (V,E) un grafo indirecto y conectado, con valores de peso reales ω definidos en E. Sea A un subconjunto de E dentro de un árbol mínimo de expansión para G, sea (S,V-S) cualquier corte de G

que respeta a A, y sea (u, v) una arista liviana cruzando (S, V-S). Entonces (u, v) es segura para A.

Una prueba de la regla anterior es la siguiente: Sea T un árbol de expansión mínima que incluye A, y asuma que T no contiene la arista liviana (u, v), de otra manera terminaríamos aquí. Debemos construir otro árbol de expansión mínima T'que incluya A$\cup(u, v)$ utilizando una técnica de çorte y pegue", mostrando así que (u, v) es segura para A.

La arista (u, v) forma un ciclo con las aristas en el camino p de u a v en T, como se ilustra en la figura 1.10. Los vértices negros están en S y los blancos en V-S. Las aristas en A están sombreadas, y (u, v) es una arista liviana cruzando el corte (S, V-S). La arista (x, y) yace en el camino p de u a v en T y no está en A. Para formar un árbol de expansión mínima T'que contenga (u, v), se remueve la arista (x, y) de T y se agrega (u, v), por lo tanto:

$$\omega(T') = \omega(T) - \omega(x, y) + \omega(u, v) \leq \omega(T) \tag{1.2}$$

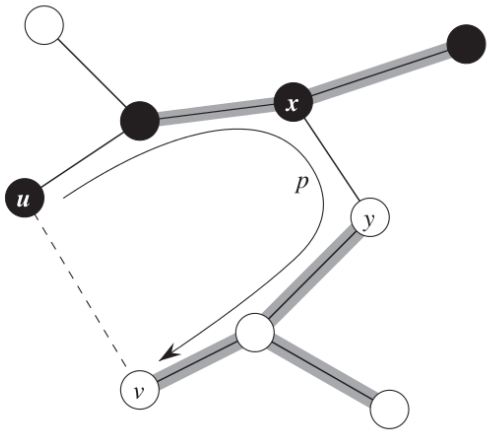

Figura 1.11: Ilustración de la prueba de la regla la encontrar una arista segura

De esta manera T' debe ser un árbol de expansión mínima también, además tenemos que A $\subseteq$ T', y A $\subseteq$ T además $(x, y) \notin$ A; en consecuencia A$\cup(u, v) \subseteq T'$, entonces (u, v) es seguro para A.

El ciclo **while** en las líneas 2-4, del seudocódigo anterior se ejecuta V-1 veces. Inicialmente cuando A =$\emptyset$, hay V árboles en $G_A = (G, A)$, y cada iteración reduce ese número en 1. Cuando el bosque contiene un solo árbol, el método termina. Ambos algoritmos implementados siguen esta regla, solo que cada uno usa una regla específica para encontrar la arista segura de la línea 3.

Clase WeightedEdge

Para representar un grafo de aristas con peso se procedió a extender la clase graph básica, en una representación de matriz de adyacencia, la cual puede contener aristas con peso en lugar de valores enteros, las cuales están formadas por los nodos y su peso. Para esto se creó una clase Edge y WeightedGraph que cuentan con los siguientes métodos:

API Clase Edge	
Edge(int u, int v, float w)	método constructor, inicializa la arista a partir de sus nodos u, v y peso w
float weight()	devuelve el peso de la arista
int either()	devuelve cualquiera de los nodos que conforman la arista
int other(int v)	devuelve el nodo u
int compareTo(Edge e)	compara el peso w con el de la arista e
void toString()	imprime los nodos y el peso de la arista
Edge& operator=(const Edge&)	sobrecarga del operador asignación para los objetos Edge

API Clase WeightedGraph	
WeightedGraph(int V)	método constructor, inicializa y reserva memoria para un grafo de V vértices
int V()	devuelve el número de vértices del grafo
int E()	devuelve el número de aristas del grafo
void addEdge(Edge e)	agrega un objeto edge al grafo y a la lista de adyacencia

Cada bolsa de la lista de adyacencia es una lista enlazada, respetando la implementación de los algoritmos de grafos indirectos y directos, cuyo contenido referencia a objetos Edge, o aristas que se conectan a el nodo v. Se selecciona esta estructura debido a que logra un código más compacto y limpio, sin embargo conlleva un pequeño precio, cada nodo de la lista de adyacencia, tiene una referencia a un objeto Edge con información redundante. Sin embargo tenemos solo una copia de cada uno.

El tipo de datos que analizarán los programas tienen la siguiente estructura:
Grafo TinyEW, de la figura 1.8:

```
8
16
4 5 0.35
4 7 0.37
5 7 0.28
0 7 0.16
1 5 0.32
```

```
0 4 0.38
2 3 0.17
1 7 0.19
0 2 0.26
1 2 0.36
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93
```

Donde en la primera fila se especifica el número de vértices, en la segunda el número de aristas, y en las siguientes filas se define cada arista, en las cuales primero se muestran los dos nodos que la conforman y el tercer número corresponde a su peso. A este tipo de grafo se le llama Euclidiano, ya que todos sus vértices son puntos que se encuentran en un mismo plano. El objetivo es encontrar el MST de tal tipo de grafo en una cantidad de tiempo razonable. Obteniendo un resultado como el siguiente:

```
El MST del grafo anterior es:
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
su peso total es: 1.81
```

Algoritmo de Prim

Este algoritmo es un caso especial del método genérico mostrado en la sección anterior. Tiene la propiedad de que cada arista en el conjunto A siempre forma un solo árbol. Como lo muestra la figura ref3, el árbol empieza de un vértice raiz arbitrario r y crece hasta que se expanda por todos los vértices en V. Cada paso agrega una arista liviana al árbol A, que lo conecta con un vértice aislado, la cual es segura para A; por lo tanto cuando el algoritmo termina, A forma un árbol de expansión mínima.

Estructuras de datos: Estas van a representar los vértices en el árbol, las aristas en el árbol, y las aristas que cruzan los cortes, de la siguiente manera:

- *Vértices en el árbol*: Utilizamos un array booleano indexado llamado marked[ ], donde marked[v] es true si v está en el árbol.

- *Aristas en el árbol*: Se pueden utilizar dos estructuras: una cola llamada mst para almacenar objetos Edge o un array indexado llamado edgeTo[ ] de objetos Edge, donde edgeTo[v] es la arista que conecta v con el árbol.
- *Aristas cruzando un corte*: Utilizamos una cola de prioridad MinPQ que compara aristas por peso.

Cada vez que agregamos una arista al árbol, agregamos un vértice también. Para mantener un grupo de aristas cruzantes, debemos agregar a una cola de prioridad todas las aristas de ese vertice a todos los vértices que no forman parte del árbol, usando marked[ ] para identificarlos. Sin embargo debemos hacer algo adicional, toda arista conectada a el vértice recién añadido debe marcarse como ineligible, es decir, no sería más una arista cruzante, porque conectaría dos vértices del árbol, haciéndolo cíclico.

Se desarrollaron dos implementaciones del algoritmo de Prim, una en la cual no se eliminan estas aristas ineligibles y se dejan en la cola de prioridad, al cual llamamos *Lazy* o perezoso, y otro el cual los elimina de la cola llamado *Eager*.

Para implementar el algoritmo utilizamos un método privado llamado visit(), que agrega un vértice al árbol, marcándolo en marked[ ], y agregando todas las aristas incidentes (que no son ineligibles si es eager) a este vértice en la cola de prioridad. Un ciclo interno del método toma una arista de la cola, y (si no es ineligible) la agrega al árbol junto con su vértice, actualizando el grupo de aristas cruzantes llamando de nuevo a visit() con el nuevo vértice agregado como argumento.

Para implementar el Eager Prim, que es más eficiente, ya que selecciona de una forma más rápida las nuevas aristas que se agregan al árbol A, necesitamos mantener en la cola de prioridad solo una arista por cada vértice v que no esté en A: La mínima arista, o la más liviana. Las demás se convierten en ineligibles.

En este algoritmo se reemplazan las estructuras de datos marked[ ] y mst[ ] en LazyPrim por dos arrays indexados edgeTo[ ] y distTo[ ], que tienen las siguientes propiedades:

- Si v no está en el árbol pero tiene al menos una arista conectada a este, entonces edgeTo[v] es la mínima arista conectando v a el árbol, y distTo[v] es el peso de esa arista.
- Todos esos vértices v son mantenidos en la cola de prioridad indexada, como un índice v asociado a esa arista.

La mínima clave o *key* en la cola de prioridad, es el peso de la arista liviana, y su vértice asociado v, es el siguiente que se agregará a A. La estructura marked[ ] no es necesaria, ya que la condición !marked[v] es equivalente a la condición distTo[v]=∞.

Algoritmo de Kruskal

Este algoritmo encuentra una arista segura para agregar al bosque en crecimiento (MST), tomando de todas la que conectan dos árboles cualquiera en el bosque, la de menor peso. Sea C_1 y C_2 estos dos árboles cualquiera, conectados por (u, v), y ya que este debe ser una arista liviana y además conecta a C_1 a otro árbol, implica que (u, v) es segura para C_1.

Empezamos con un bosque degenerado de V vértices individuales y realizamos una operación de combinación de parejas de árboles por medio de aristas livianas, hasta que quede un solo árbol: el MST.

La figura ref44 ilustra paso por paso un ejemplo de la operación de Kruskal sobre el grafo TinyEW, cuyas aristas se muestran ordenas por peso a continuación:

```
0-7 0.16 -arista del MST
2-3 0.17 -arista del MST
1-7 0.19 -arista del MST
0-2 0.26 -arista del MST
5-7 0.28 -arista del MST
1-3 0.29 -arista obsoleta (forma un ciclo)
1-5 0.32 -arista obsoleta (forma un ciclo)
2-7 0.34 -arista obsoleta (forma un ciclo)
4-5 0.35 -arista del MST
1-2 0.36 -arista obsoleta (forma un ciclo)
4-7 0.37 -arista obsoleta (forma un ciclo)
0-4 0.38 -arista obsoleta (forma un ciclo)
6-2 0.40 -arista del MST
3-6 0.52 -arista obsoleta (forma un ciclo)
6-0 0.58 -arista obsoleta (forma un ciclo)
6-4 0.93 -arista obsoleta (forma un ciclo)
```

1.2.4. Algoritmos para la búsqueda de la ruta más corta

Una de las incógnitas más importantes que tiene en el procesamiento de algortmo es: ¿Se puede encontrar un camino óptimo entre dos vértices del grafo?. La importancia de este problema se puede observar en fenómenos tecnológicos como el ruteo de paquetes en las redes de computadoras.

La definición del problema es que dado un grafo dirigido con pesos *G = (V,E)*, con una función de pesos $w : E \longrightarrow \mathbb{R}$ que mapea los enlaces a valores reales de peso. El peso *w(p)* de un path *p = (v0,...,vk)* es la suma de los pesos de sus enlaces constituyentes. De este modo el path más corto de un vértice u a un vértice v se define como:

INSERTAR ECUACION

Existen dos fenómenos importantes que pueden suceder cuano se trata con grafos:

- *Enlaces Negativos*
- *Ciclos*

Con respecto a los enlaces negativos, el problema principal radica en que exista un ciclo de peso negativo (que la suma de los bordes que forman el ciclo sea negativa), ya que si esto ocurre, entonces los paths más cortos no están bien definidos. Esto es debido a que ningún camino desde s hasta cualquiera de los nodos en el ciclo puede ser tomado como el más corto.

Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford resuelve el problema del camino más corto, para el caso general en el cual los pesos pueden tomar valores negativos. Éste propone que dado un grafo dirigido con peso, es posible encontrar un path entre cualquier par de nodos que cumpla con la condición de ser el camino más corto, siempre y cuando no existan ciclos negativos.

El algoritmo se lleva a cabo relajando los enlaces de manera progresiva, disminuyendo un estimado de la distancia a un vértice dado desde la fuente. El pseudocódigo que describe al Bellman-Ford es el propuesto en el algoritmo 3

Algorithm 3 Pseudocodigo Bellman Ford

```
procedure BellmanFord(G)
    for all i = 1 to G.V - 1 do
        for all edge ∈ G do
            Relax(u,v,w)
        end for
    end for
    for all edge ∈ G do
        if v.d ≥ u.d + w(u,v) then
            return FALSE
        end if
    end for
end procedure
```

Como se observa en el algoritmo se llama a un procedimiento denominado relajación. En este procedimiento se itera sobre los enlaces para definir distintos pesos para cada uno y se itera hasta saber el valor óptimo del camino hasta el vértice. El algoritmo corre en un tiempo polinomial de O(VE).

Algoritmo de Djikstra

El algoritmo de Djikstra es utilizado para resolver el problema del camino más corto en grafos dirigidos con valores de peso positivo. El algoritmo mantiene un conjunto de vértices a los cuales ya se les ha determinado el camino más corto desde el origen. El algoritmo selecciona de manerar repetitiva el vértice con el estimado más bajo, para esto se hace uso de una cola de prioridad. El algoritmo básico es el 4.

Algorithm 4 Algoritmo de Djikstra

```
procedure DJIKSTRA(G,w,s)
    S = 0
    Q = G.V
    while Q ≠ 0 do
        u = Min(Q)
        S = S U u
        for all vertex ∈ adj[u] do
            Relax(u,v,w)
        end for
    end while
end procedure
```

El algoritmo relaja los enlaces de manera que pueda ir determinando el valor más bajo de la cola de prioridad. La cola de prioridad es sumamente importante en la implementación debido a que permite obtener el valor mínimo del sistema que será elegido como el camino óptimo hacia cierto vértice.