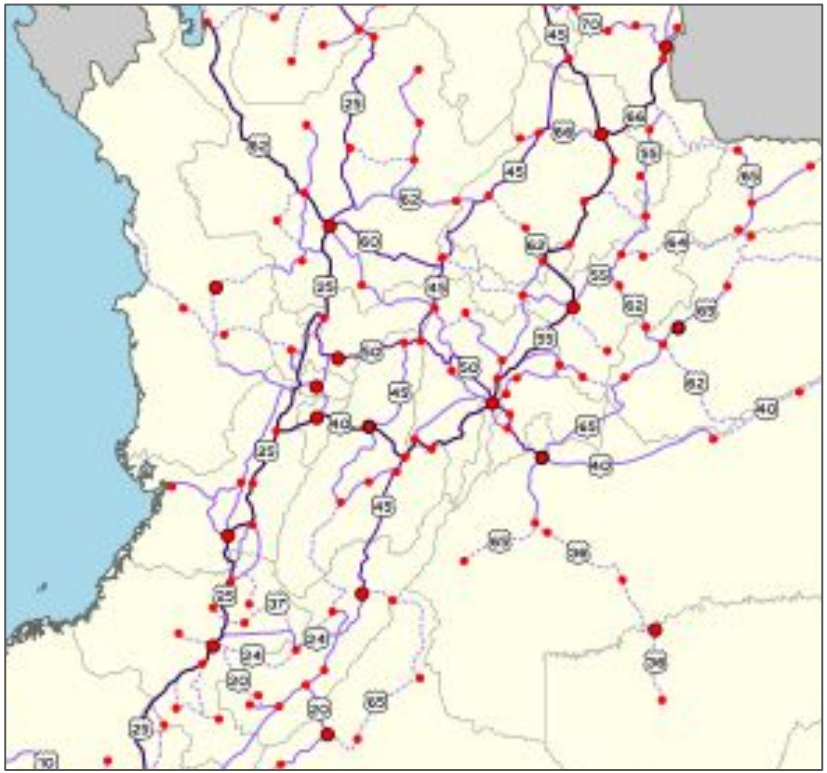
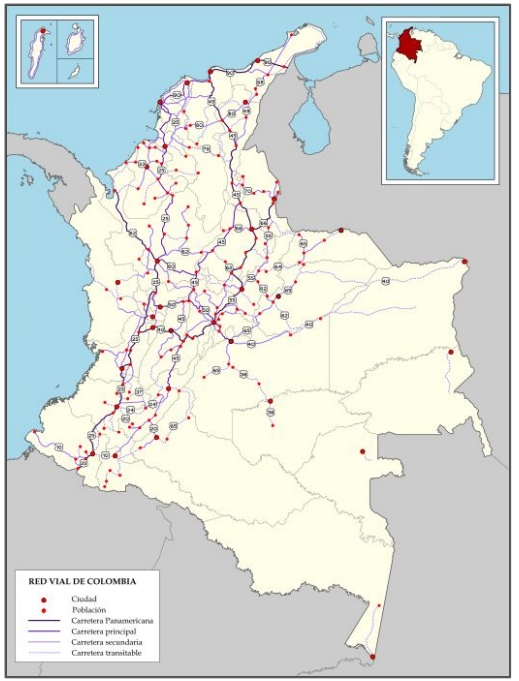
Grafos

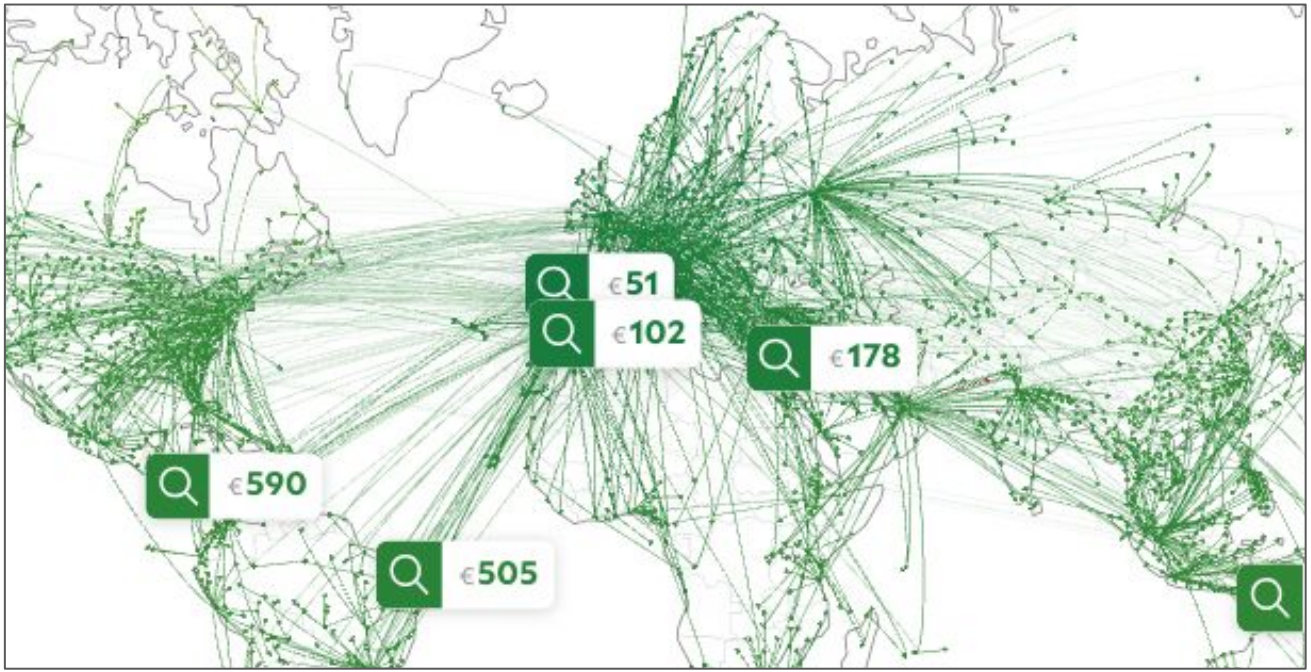
Prof.: Néstor Suat-Rojas. Ing., M.Sc.

Grafos

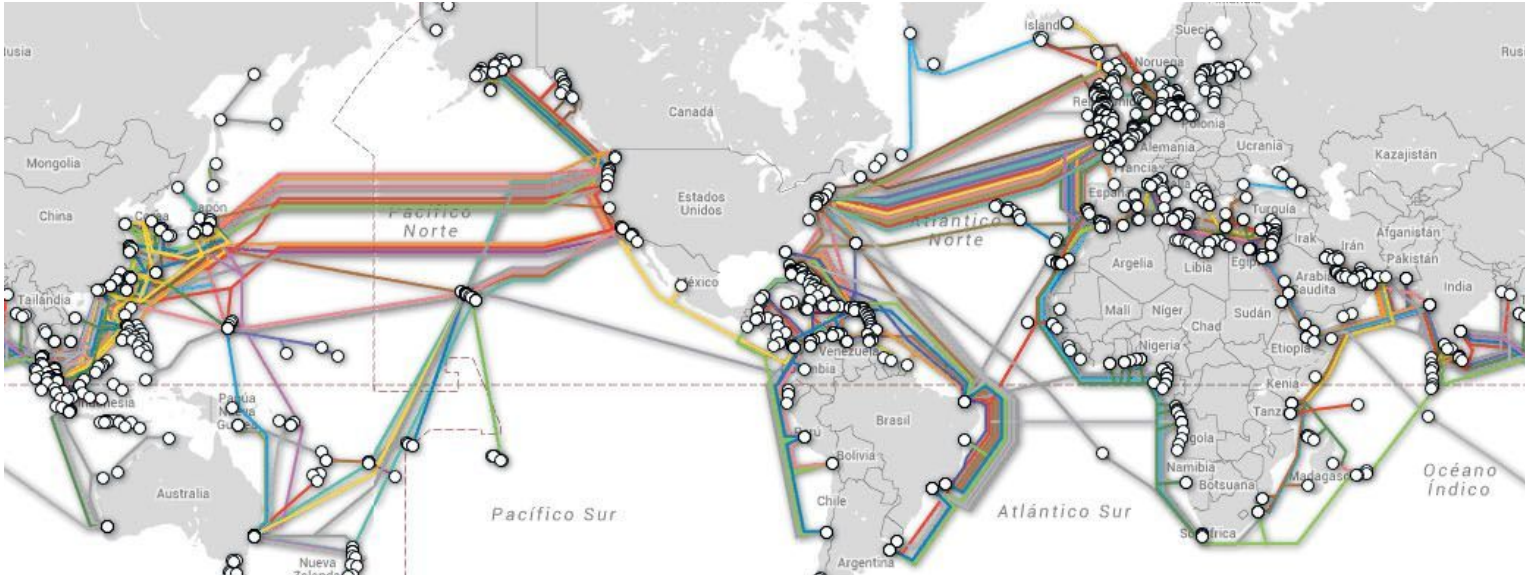



RED VIAL DE COLOMBIA
Ciudad
Población
Carretera Panamericana
Carretera principal
Carretera secundaria
Carretera transitable

Grafos

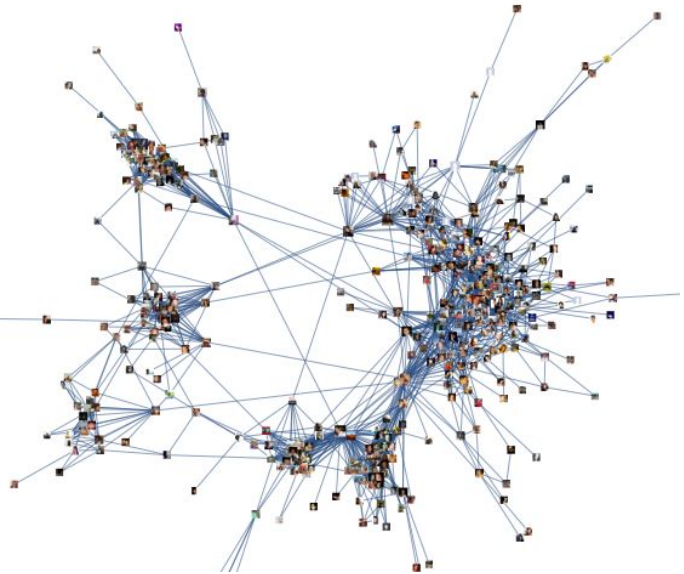


Grafos


Canadá
Estados Unidos
Pacífico Norte
Atlántico Norte
Pacífico Sur
Atlántico Sur
Océano Índico
Mongolia
China
Tailandia
Australia
Brasil
Bolivia
Chile
Argentina
Venezuela
Argelia
Libia
Mali
Niger
Chad
Sudán
Nigeria
Etiopía
Kenia
Namibia
Botsuana
Madagascar
Ucrania
Kazajistán
Irak
Irán
Afganistán
Pakistán
India
Alemania
Francia
Noruega
Islandia

Grafos

Amigos en Facebook

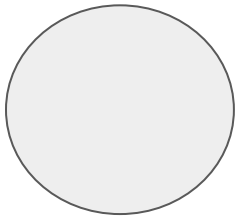


Es fácil para los seres humanos ver mapa de carreteras y entender las relaciones entre diferentes lugares, una computadora no tiene tal conocimiento.

Definiciones

- **Vértice:**

  También llamado “nodo”. Es una parte fundamental de un grafo. Puede tener un nombre, que llamaremos **“clave”**. Un nodo también puede tener información adicional (**“carga útil”**).



Nodo

Definiciones

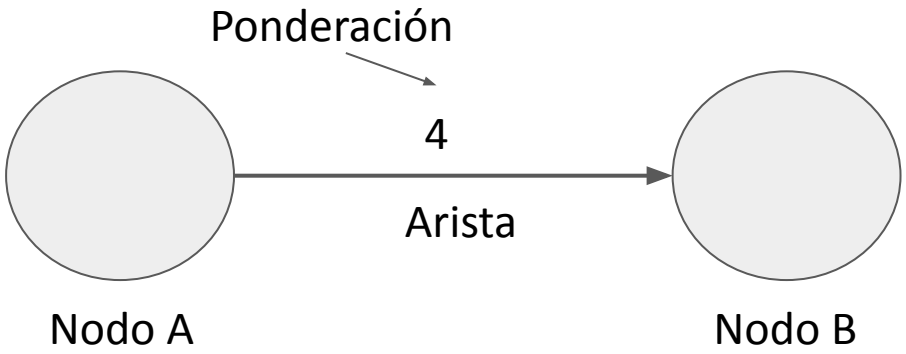
- **Arista:**

  Una arista (“arco”) conecta dos vértices para mostrar que hay una relación entre ellos. Las aristas pueden ser unidireccionales o bidireccionales. Si las aristas de un grafo son todas unidireccionales, decimos que el grafo es un **grafo dirigido** o un **digrafo**.

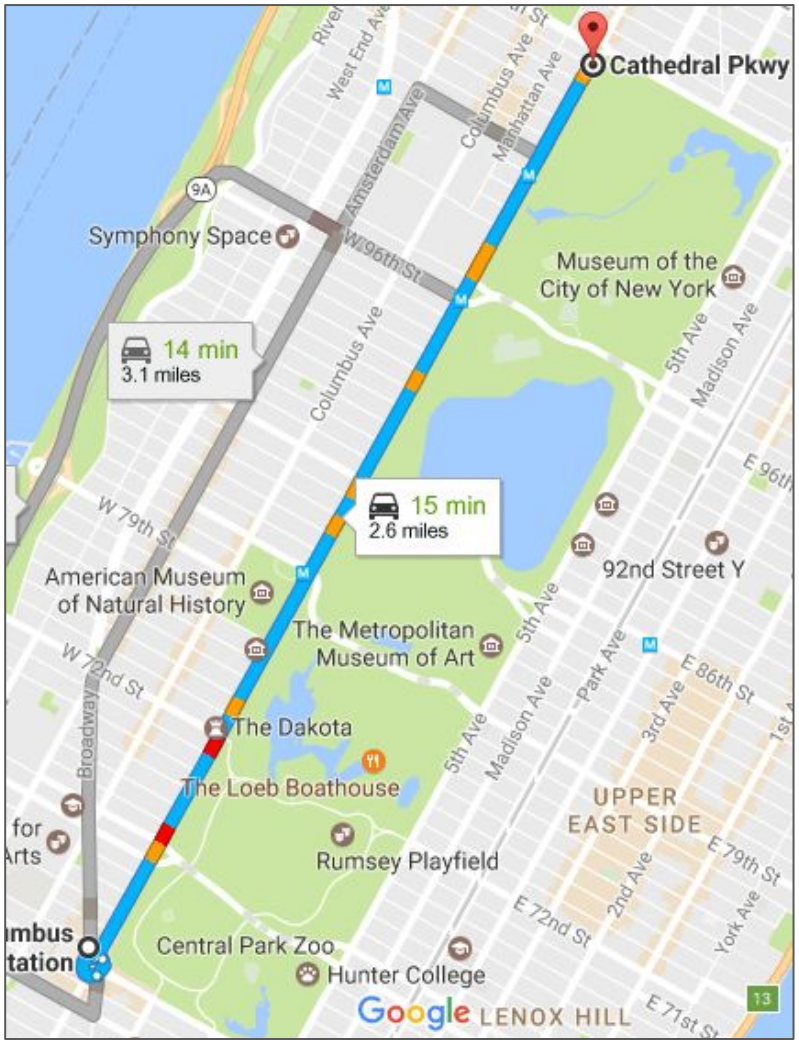
Definiciones

- **Ponderación o Peso:**

  Las aristas pueden ponderarse para mostrar que hay un costo para ir de una arista a otra.



Definiciones

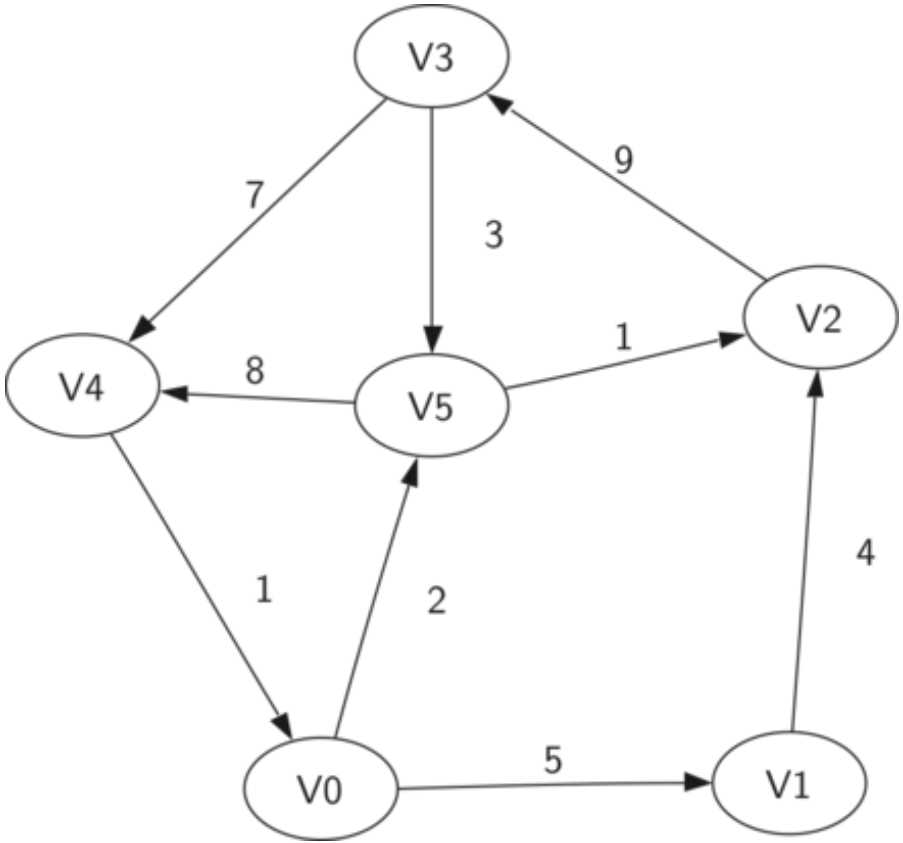


Definiciones

- **Grafo, definición formal:**

  Un grafo puede ser representado por G donde $G = (V, E)$. Para el grafo G, V es un conjunto de vértices y E es un conjunto de aristas. Cada arista es una tupla (v, w) donde $w, v \in V$. Podemos añadir un tercer componente a la tupla de la arista para representar una ponderación. Un subgrafo s es un conjunto de aristas e y de vértices v tales que $e \subset E$ y $v \subset V$.

Definiciones



$V=\{V0, V1, V2, V3, V4, V5\}$

$E=\{$
$(v0,v1,5),(v1,v2,4),(v2,v3,9),$
$(v3,v4,7),(v4,v0,1),(v0,v5,2),$
$(v5,v4,8),(v3,v5,3),(v5,v2,1)$
$\}$

Definiciones

- **Ruta:**

  Una ruta en un grafo es una secuencia de vértices que están conectados por las aristas.

  - La longitud de la ruta no ponderada es el número de aristas en la ruta.
  - La longitud ponderada de la ruta es la suma de las ponderaciones de todos las aristas en la trayectoria.

Por ejemplo, en la imagen anterior la ruta desde $V3$ hasta $V1$ es la secuencia de vértices $(V3,V4,V0,V1)$. Las aristas son $\{(v3,v4,7),(v4,v0,1),(v0,v1,5)\}$.

Definiciones

- **Ciclo:**

  Un ciclo en un grafo dirigido es una ruta que comienza y termina en el mismo vértice. Por ejemplo, en la imagen anterior la ruta **(*V*5,*V*2,*V*3,*V*5)** es un ciclo. Un grafo sin ciclos se denomina **grafo acíclico**. Un grafo dirigido sin ciclos se denomina **grafo acíclico dirigido** o **GAD**.

Implementación

`Graph()`	Crear un grafo nuevo y vacío.
`addVertex(clave)`	Agrega una instancia de `Vértice` al grafo.
`addEdge(from, to)`	Agrega al grafo una nueva arista dirigida que conecta dos vértices.
`addEdge(from, to, weight)`	Agrega al grafo una nueva arista ponderada y dirigida que conecta dos nodos.
`getVertex(clave)`	Encuentra el vértice en el grafo con nombre `clave.`

Implementación

Hay dos formas de representar la implementación, utilizando:

1. Matriz de adyacencia.
2. Lista de Adyacencia.

Matriz de Adyacencia

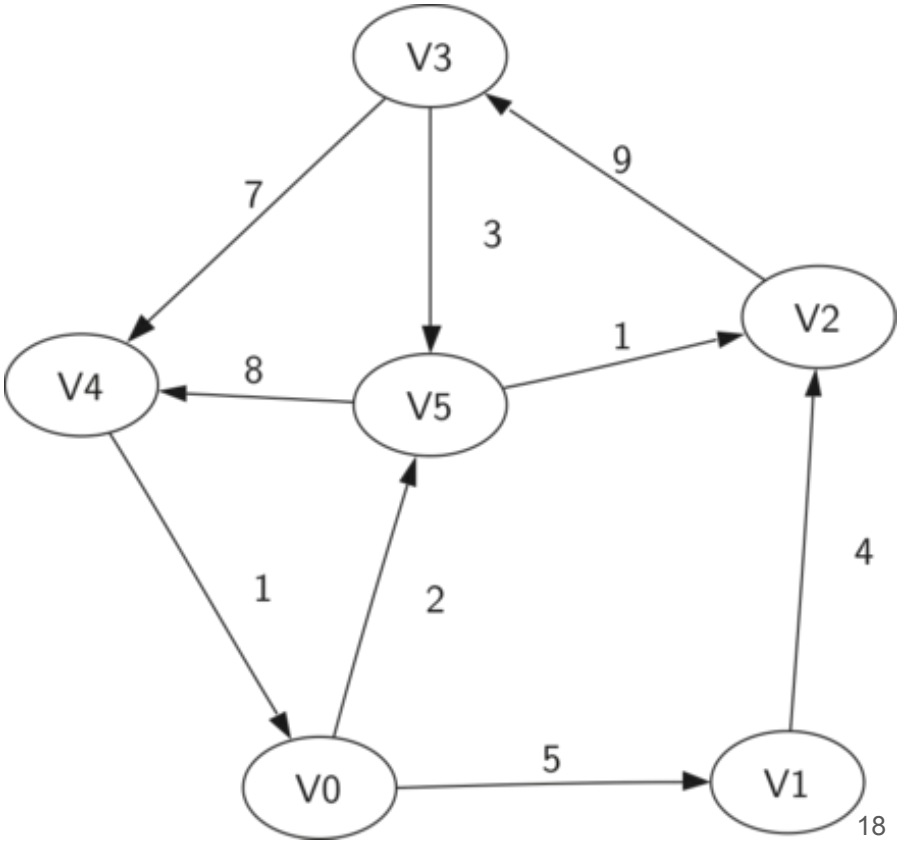
- Implementar un grafo usando una matriz bidimensional.
- En la matriz, cada fila y columna representa un vértice en el grafo.
- El valor que almacena la celda en la intersección de la fila v y la columna w indica si hay una arista desde el vértice v al vértice w.

Matriz de Adyacencia

- Cuando dos vértices están conectados por una arista, decimos que son **adyacentes**.
- Un valor en una celda representa la ponderación de la arista que une el vértice v con el vértice w.

Matriz de Adyacencia

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	



Matriz de Adyacencia

Dificultades

- Es simple y fácil de implementar para grafos pequeños. Sin embargo muchas celdas de la matriz están vacías.
- Para un diseño de grafo con muchos vértices vamos a tener una matriz demasiado grande, con muchos ceros.
- Una matriz está llena cuando cada vértice está conectado a todos los otros vértices. Hay pocos problemas reales que se aproximan a este tipo de conectividad.

Implementación

Hay dos formas de representar la implementación, utilizando:

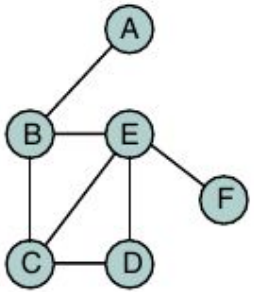
1. Matriz de adyacencia.
2. Lista de Adyacencia.

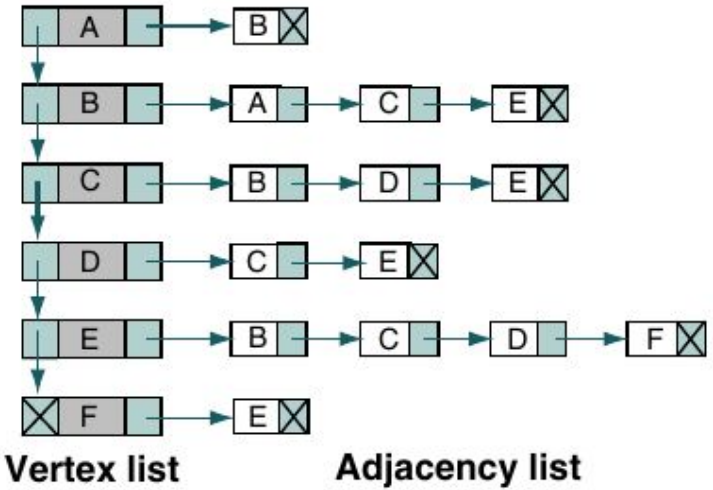
Lista de adyacencia

- Una implementación de lista de adyacencia mantiene una lista maestra de todos los vértices en el objeto Grafo.

- Cada objeto Vértice en el grafo mantiene una lista de los otros Vértices a los que está conectado.

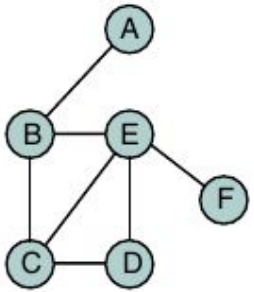
Lista de adyacencia

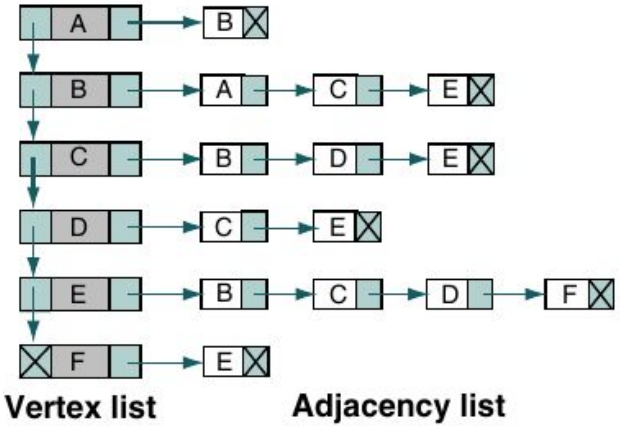




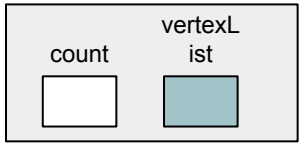
- Lista maestra de todos los vértices.
- Cada vértice mantiene una lista de los otros vértices que está seleccionado.

Lista de adyacencia

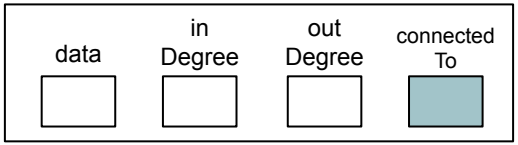




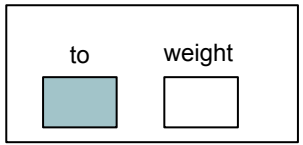
Graph



Vertex



Edges

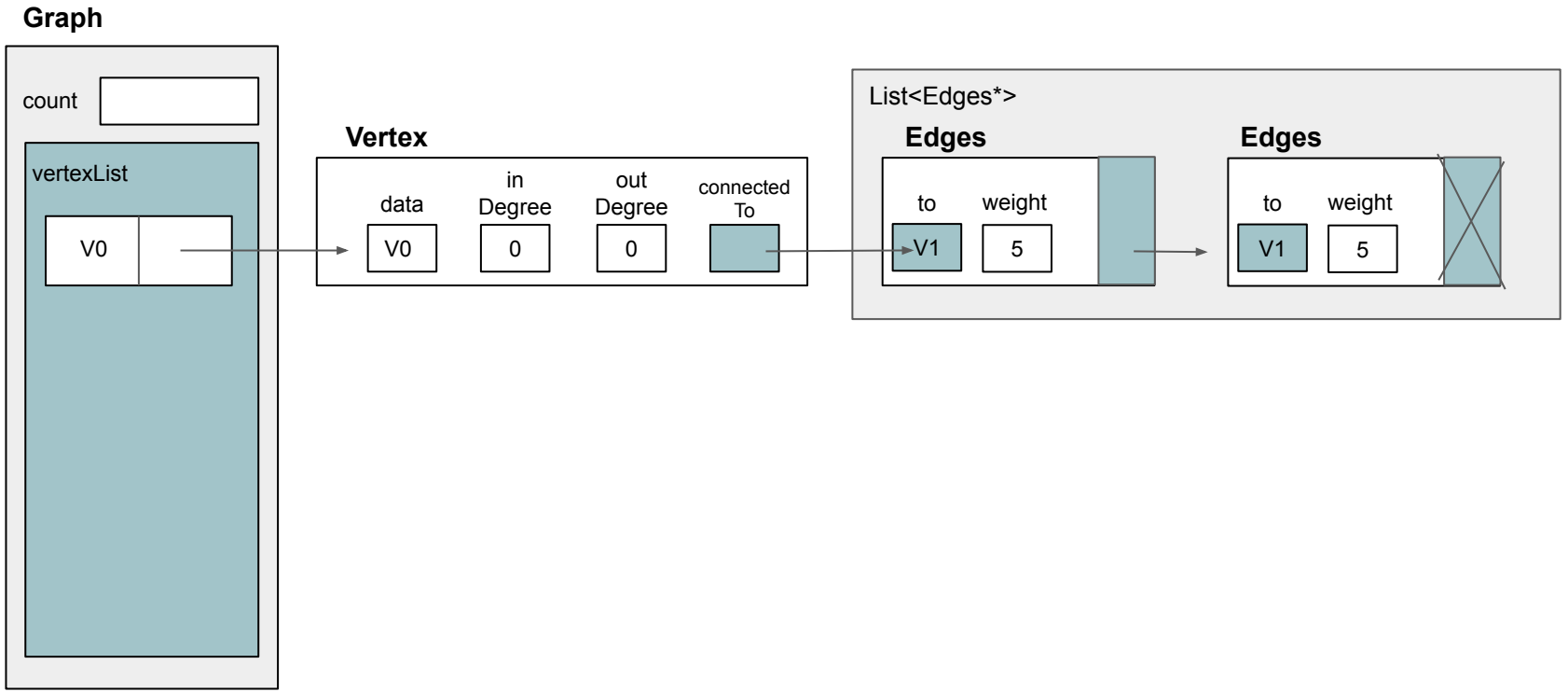


Lista de adyacencia

Implementación:

- Crear tres clases:
  - **Grafo:** Contiene la lista maestra de vértices
  - **Vértice:** Representa cada vértice del grafo.
  - **Arista:** Representa cada conexión del grafo.
- Cada vértice utiliza una lista para realizar un seguimiento de los vértices a los que está conectado, y la ponderación de cada arista.
  - La lista se llama `conectadoA`

Lista de adyacencia



Lista de adyacencia


```
template<class T>
class Edge{
public:
   Vertex<T>* to;
   int weight;
   friend ostream &operator<<(ostream &out, Edge<T>* edge) {
        out << "To: " << edge->to->data;
        out << ", Weight: " << edge->weight << endl;
        return out;
   }
};
```

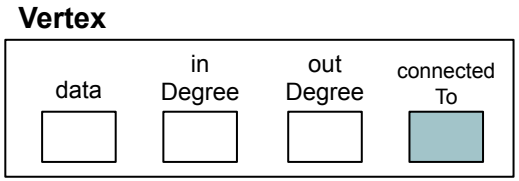
Lista de adyacencia

Implementación clase Vértice:

- Método `addNeighbor`
  - Se utiliza para agregar una conexión desde este vértice a otro.

- Método `getWeight`
  - Devuelve la ponderación de la arista de este vértice al vértice pasado como parámetro.

Lista de adyacencia




```
template<class T>
class Vertex{
public:
   T data;
   int inDegree;
   int outDegree;
   List<Edge<T>*> connectedTo;
   Vertex(const T& value);
   ~Vertex();
   void addNeighbor(Vertex<T>* to, int weight=0);
   int getWeight(const T& value);
   friend ostream &operator<<(ostream &out, Vertex<T>* vertex) {
       out << vertex->data << endl;
       out << "In degree: " << vertex->inDegree << endl;
       out << "out degree: " << vertex->outDegree << endl;
       out << "Edges: " << endl;
       vertex->connectedTo.print();
       return out;
   }
};
```

Implementación Vértice

Constructor

```
template<class T>
Vertex<T>::Vertex(const T& value) {
    data = value;
    inDegree = 0;
    outDegree = 0;
    connectedTo = {};
}
```

Agregar vecino

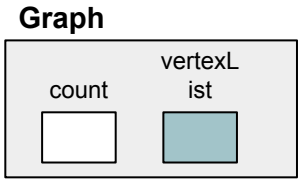
```
template<class T>
void Vertex<T>::addNeighbor(Vertex<T> *to, int weight) {
    Edge<T>* temp = new Edge<T>;
    temp->to = to;
    temp->weight = weight;

    outDegree++;
    to->inDegree++;

    connectedTo.insert(connectedTo.size(), temp);
}
```

Implementación Vértice

Obtener ponderación

```
template<class T>
int Vertex<T>::getWeight(const T &value) {
   for(int i=0; i < connectedTo.size(); i++){
       Edge<T>* temp = connectedTo.get(i);
       if(temp->to->data == value){
           return connectedTo.get(i)->weight;
       }
   }
   return NULL;
}
```

Lista de adyacencia




```
template<class T>
class Graph {
public:
    int count;
    List<Vertex<T>*> vertexList;
    Graph();
    ~Graph();
    Vertex<T>* addVertex(const T& value);
    Vertex<T>* getVertex(const T& value);
    void addEdge(const T& from, const T& to, int weight=0);
};
```

Implementación Grafo

Constructor

```
template<class T>
Graph<T>::Graph() {
    count = 0;
    vertexList = {};
}
```

Agregar vecino

```
template<class T>
Vertex<T>* Graph<T>::addVertex(const T &value) {
    Vertex<T>* newVertex = new Vertex<T>(value);
    vertexList.insert(vertexList.size(), newVertex);
    count++;
    return newVertex;
}
```

Implementación Grafo

Agregar arista

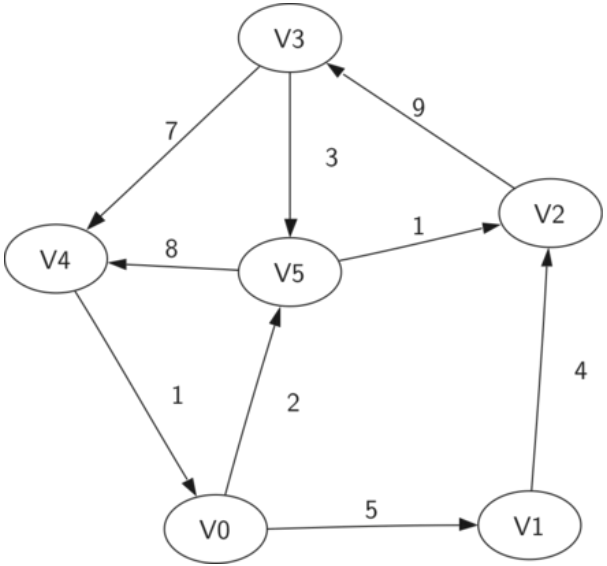
```
template<class T>
void Graph<T>::addEdge(const T& from, const T& to, int weight) {
   Vertex<T>* fromVertex = getVertex(from);
   if(!fromVertex){
       fromVertex = addVertex(from);
   }
   Vertex<T>* toVertex = getVertex(to);
   if(!toVertex){
       toVertex = addVertex(to);
   }
   fromVertex->addNeighbor(toVertex, weight);
}
```

Implementación Grafo

Obtener vértice

```
template<class T>

Vertex<T> *Graph<T>::getVertex(const T &value) {

   for(int i=0; i < vertexList.size();i++ ){

       if(vertexList.get(i)->data == value) return vertexList.get(i);

   }

   return NULL;

}
```

```
int main() {
    Graph<int> g;
    for(int i=0; i < 6; i++){
        g.addVertex(i);
    }

    g.addEdge(0,1,5);
    g.addEdge(0,5,2);
    g.addEdge(1,2,4);
    g.addEdge(2,3,9);
    g.addEdge(3,4,7);
    g.addEdge(3,5,3);
    g.addEdge(4,0,1);
    g.addEdge(5,4,8);
    g.addEdge(5,2,1);

    for(int vertexPos=0; vertexPos < g.vertexList.size(); vertexPos++ ){
        Vertex<int>* vertex = g.vertexList.get(vertexPos);
        for(int edgePos=0; edgePos < vertex->connectedTo.size(); edgePos++){
            Edge<int>* edge = vertex->connectedTo.get(edgePos);
            cout << "(" << vertex->data << ", " << edge->to->data << ", " << edge->weight << ")" << endl;
        }
    }

    cout << "Weight of Vertex 3 -> 5: "<<g.getVertex(3)->getWeight(5) << endl;

    return 0;
}
```



```
int main() {
    Graph<int> g;
    for(int i=0; i < 6; i++){
        g.addVertex(i);
    }

    g.addEdge(0,1,5);
    g.addEdge(0,5,2);
    g.addEdge(1,2,4);
    g.addEdge(2,3,9);
    g.addEdge(3,4,7);
    g.addEdge(3,5,3);
    g.addEdge(4,0,1);
    g.addEdge(5,4,8);
    g.addEdge(5,2,1);

    for(int vertexPos=0; vertexPos < g.vertexList.size(); vertexPos++ ){
        Vertex<int>* vertex = g.vertexList.get(vertexPos);
        for(int edgePos=0; edgePos < vertex->connectedTo.size(); edgePos++){
            Edge<int>* edge = vertex->connectedTo.get(edgePos);
            cout << "(" << vertex->data << ", " << edge->to->data << ", " << edge->weight << ")" << endl;
        }
    }

    cout << "Weight of Vertex 3 -> 5: "<<g.getVertex(3)->getWeight(5) << endl;

    return 0;
}
```

Output:

```
(0, 1, 5)
(0, 5, 2)
(1, 2, 4)
(2, 3, 9)
(3, 4, 7)
(3, 5, 3)
(4, 0, 1)
(5, 4, 8)
(5, 2, 1)
Weight of Vertex 3 -> 5: 3
```

Gracias