Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

Estructura de Datos

Semana 12

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

Logro de la sesión

Al finalizar la sesión, el estudiante:

- **Comprenderá el concepto de grafo y sus principales propiedades. Implementará las diferentes representaciones de un grafo así como los diferentes algoritmos de recorridos, finalmente resolverá problemas utilizando grafos.**

Estructuras de datos no lineales

Grafos

01

Agenda

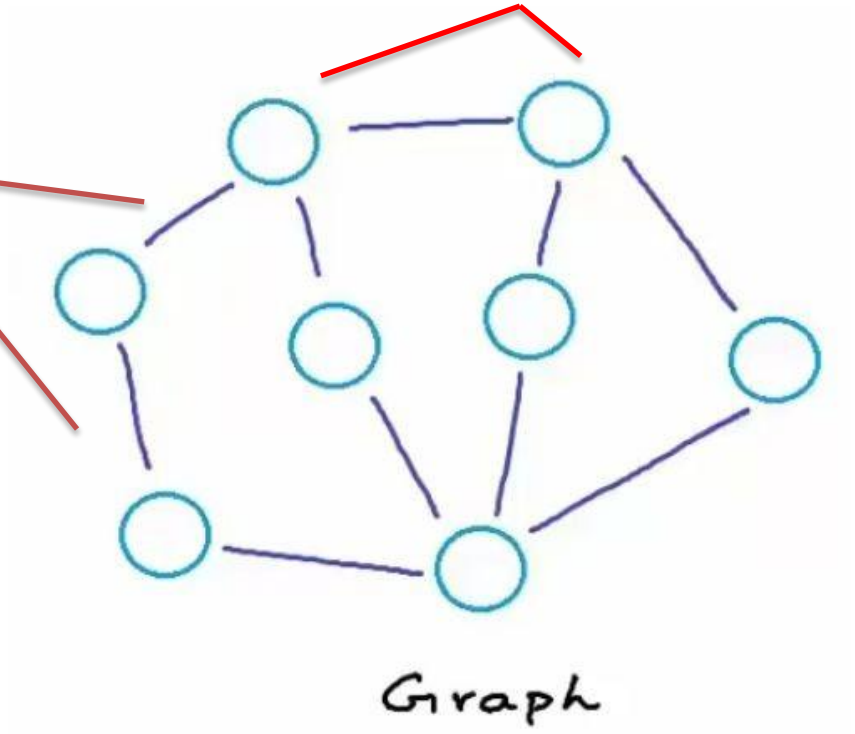
- Árboles 2-3
- Árboles B
- Código de Huffman

Introducción

Estructuras no lineales: grafos

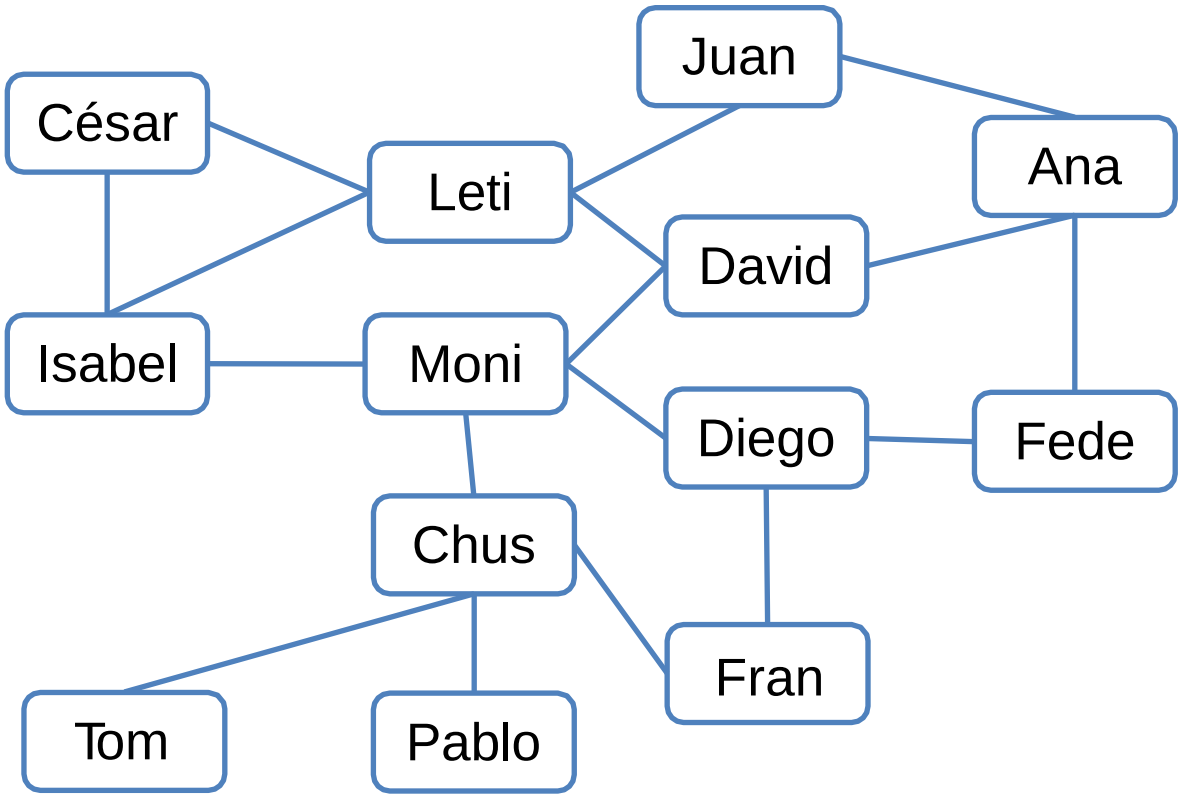
Nodos o vértices

Aristas



Permiten representar cualquier tipo de conexión

Introducción

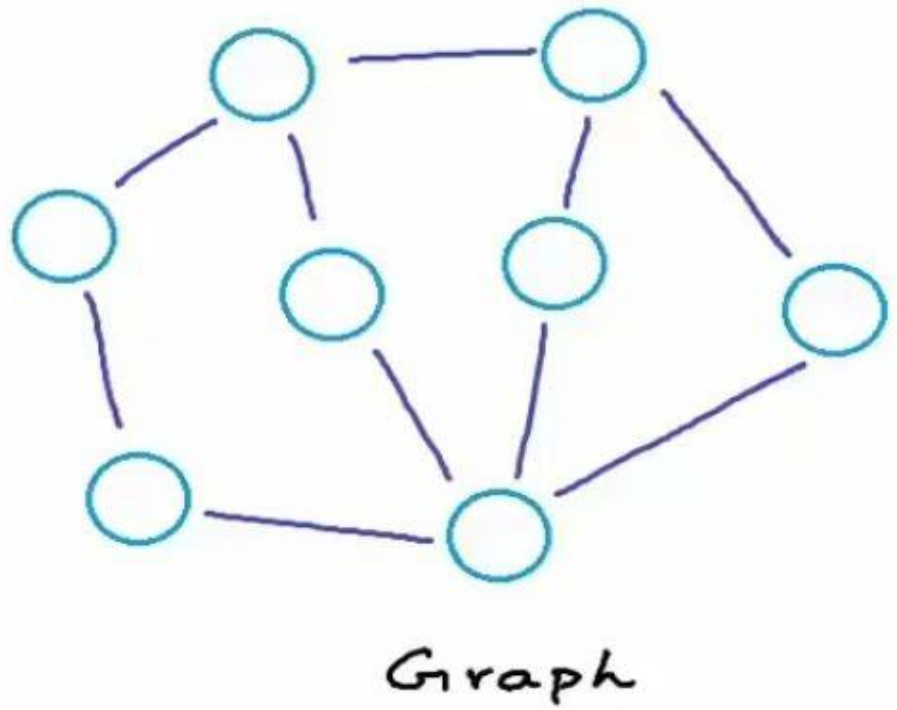


Una red social se puede representar como un grafo no dirigido y no ponderado. Los usuarios son los vértices y sus relaciones de amistad las aristas.

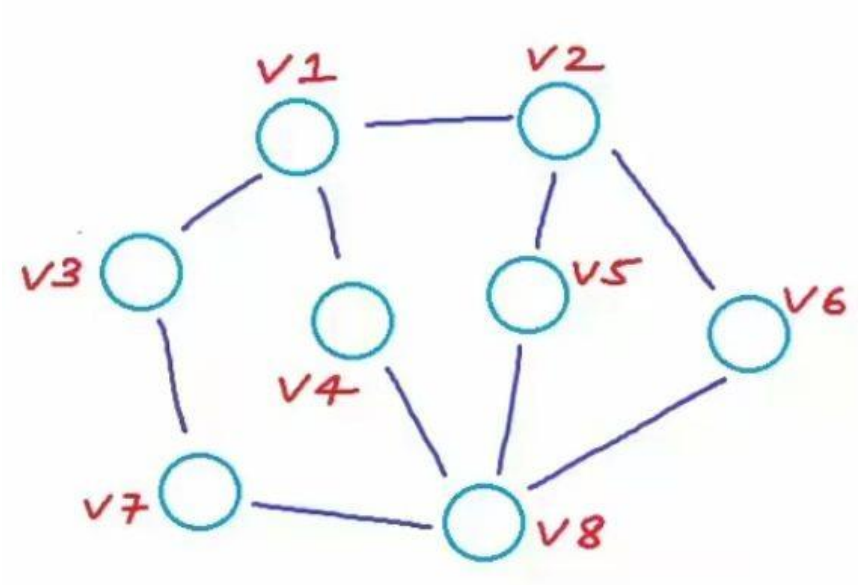
Conceptos sobre grafos

Grafo: G=(V,A)

Un grafo G es un par (V,A), donde V es un conjunto de vértices (nodos) y A un conjunto de aristas



Conceptos sobre grafos



Grafo: G=(V,A)

V = { v1, v2, v3, v4, v5, v6, v7, v8}

¿Cómo puedo representar una arista?

Conceptos sobre grafos

Tipos de Aristas

(u,v)
(u,v)!=(v,u) if u!=v

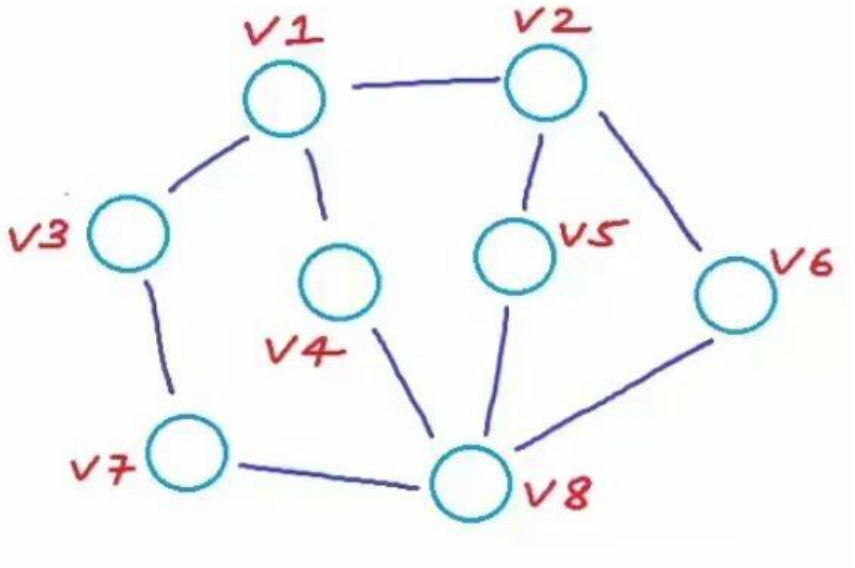
{u,v} ,
{u,v} = {v,u}

Conceptos sobre grafos

Grafo: G=(V,A)

V = { v1, v2, v3, v4, v5, v6, v7, v8}

A= { {v1, v2}, {v1, v3}, {v1, v4}, {v2,v5}, {v2,v6},
{v3,v7}, {v4,v8}, {v5,v8}, {v6,v8}, {v7,v8}}



|V|=número de vértices

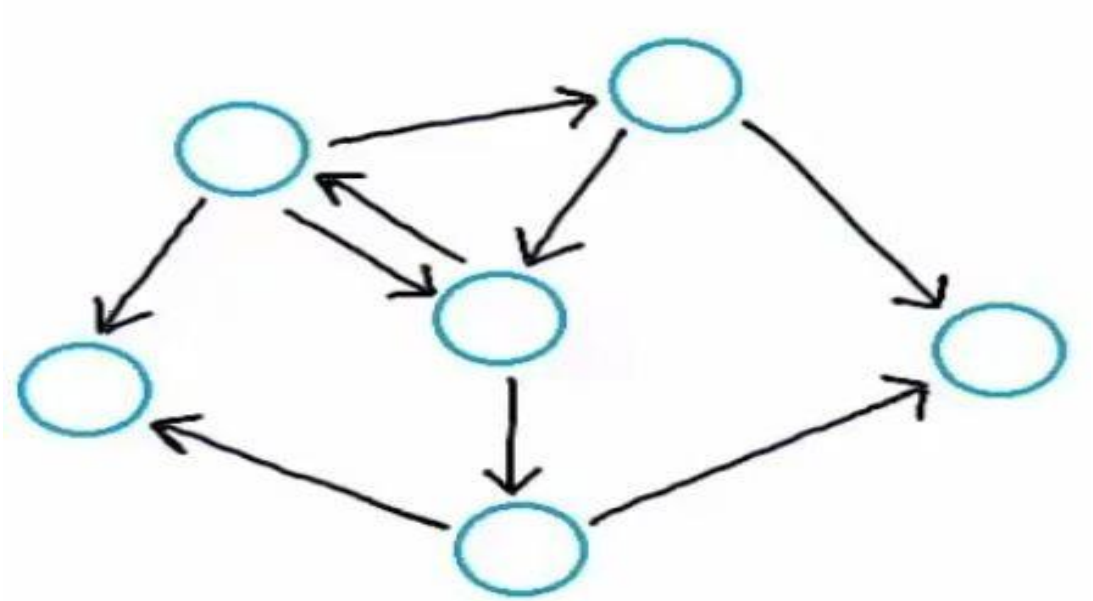
|A|=número de aristas

|V|=8

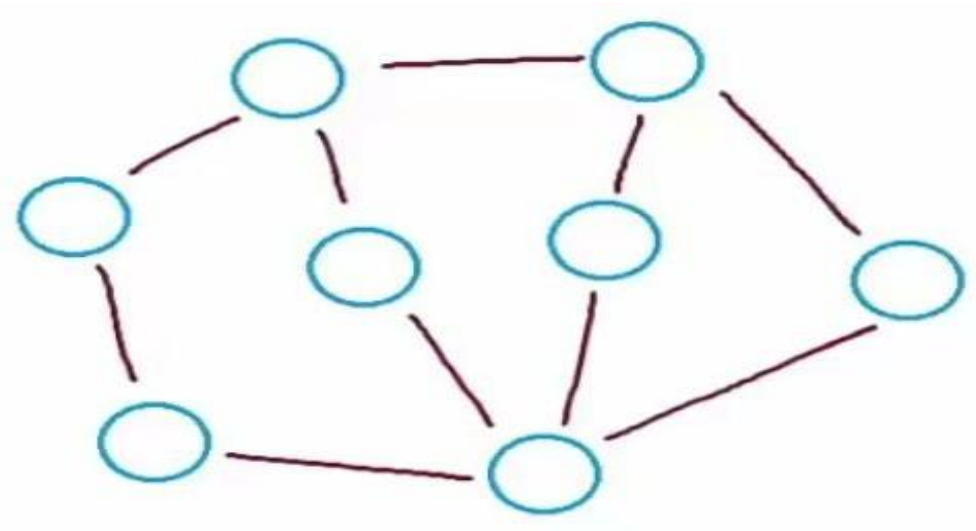
|A|=10

Conceptos sobre grafos

Grafo dirigidos



Grafo no dirigido



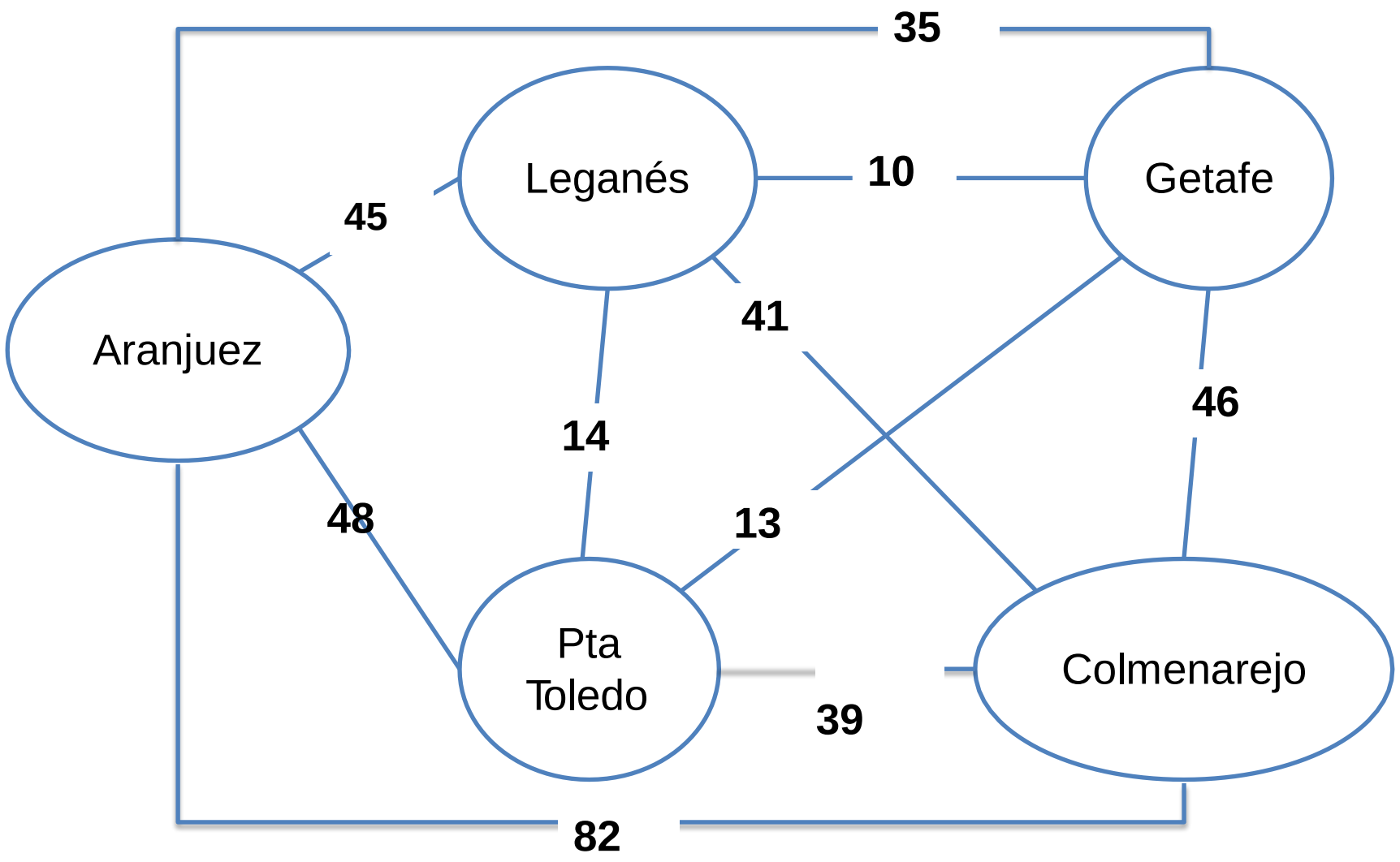
Tipos de Grafos

- **Grafos no dirigidos.**
  - Las aristas no tiene dirección, es decir, (u,v)=(v,u). La arista se puede recorrer en ambos sentidos.
  - Nos permiten representar relaciones simétricas y de colaboración.

- **Grafos dirigidos.**
  - Cada arista (u,v) tiene una única dirección, siendo u el vértice origen y v el vértice final. (u,v) ≠(v,u)
  - Nos permiten representar relaciones asimétricas y jerárquicas.

Conceptos sobre grafos

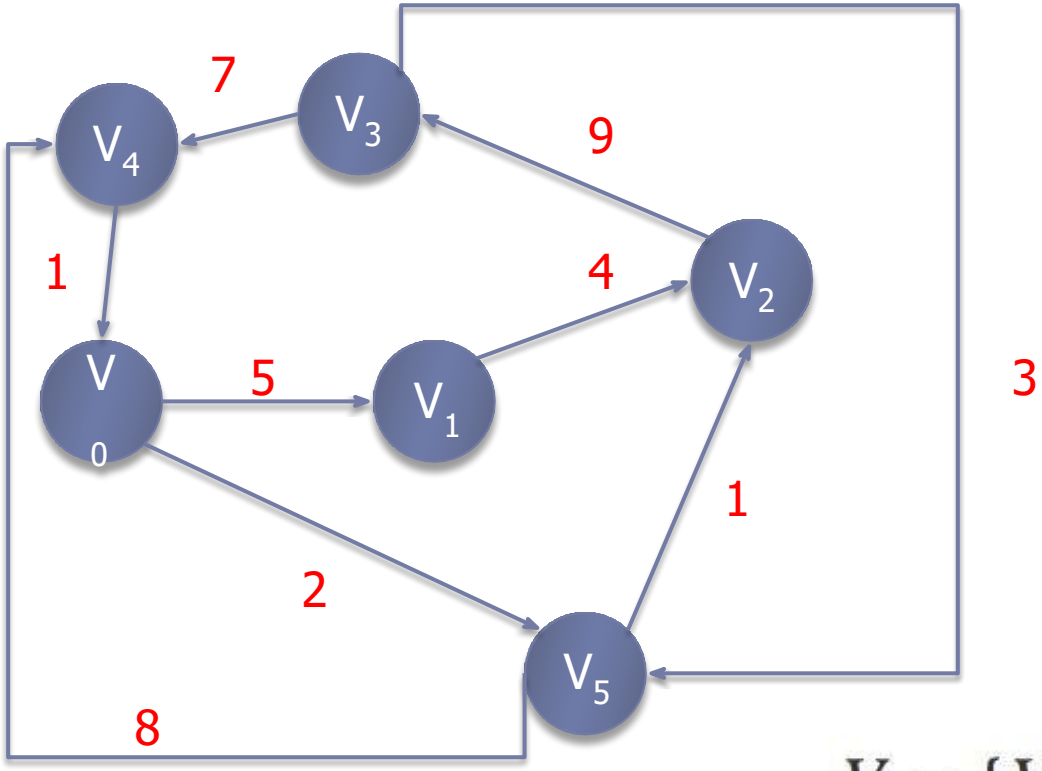
Grafo ponderado (weighted)



Conceptos Básicos

- Grafo G donde $G=(V,A)$.
  - V es un conjunto de vértices (nodos)
  - A es un conjunto de aristas (arcos).
    - Una arista es una conexión entre dos vértices.
    - Cada arista puede ser representada como una tupla (v,w) donde $w,v \in V$
    - Además, cada artista puede tener un peso asociado (**grafo ponderado**). En este caso, la arista quedaría representada por una terna (v,w,p) donde p es el peso asociado a la arista entre v y w.

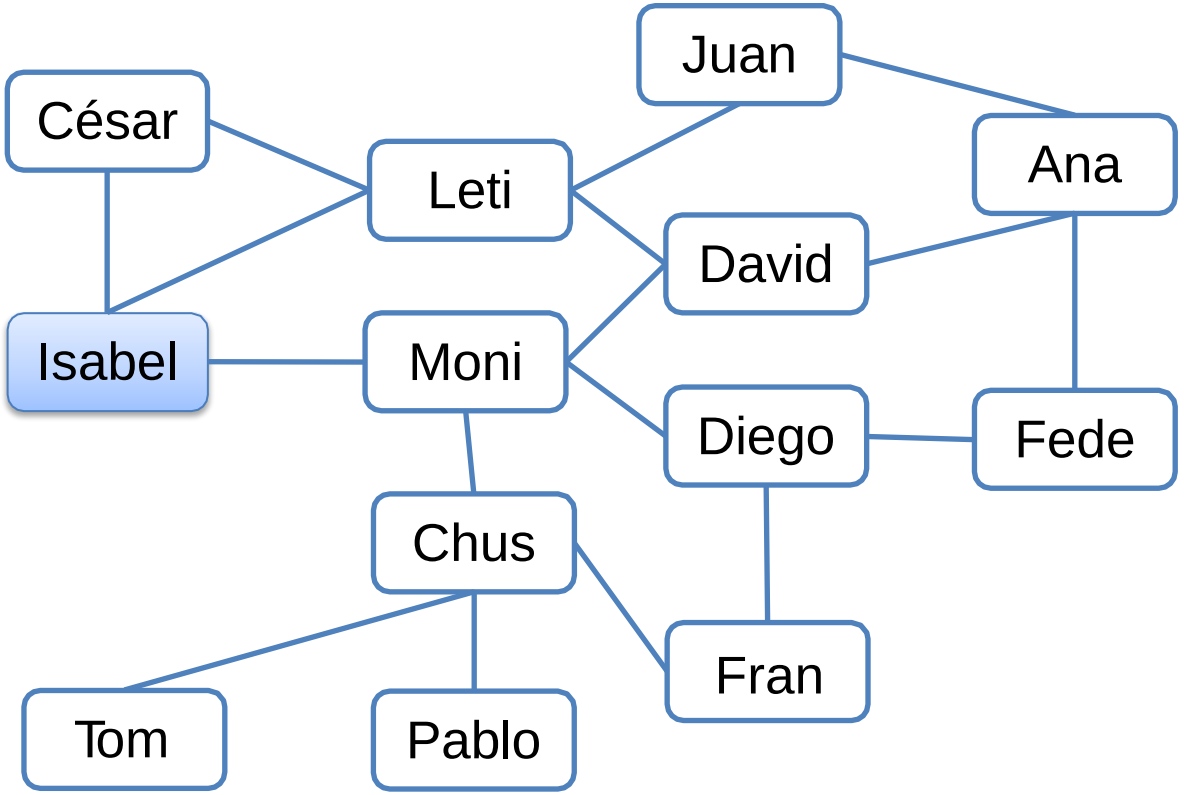
Grafo Ponderado



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

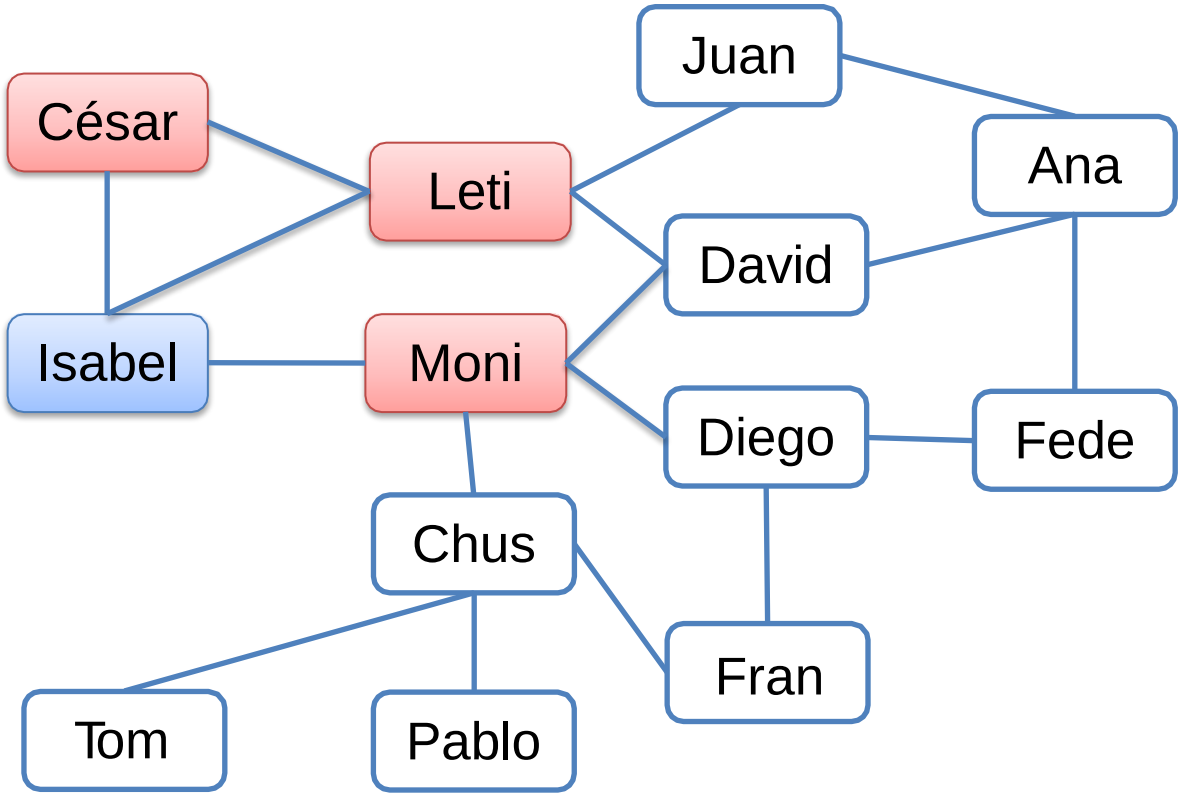
$$A = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$

Conceptos sobre grafos

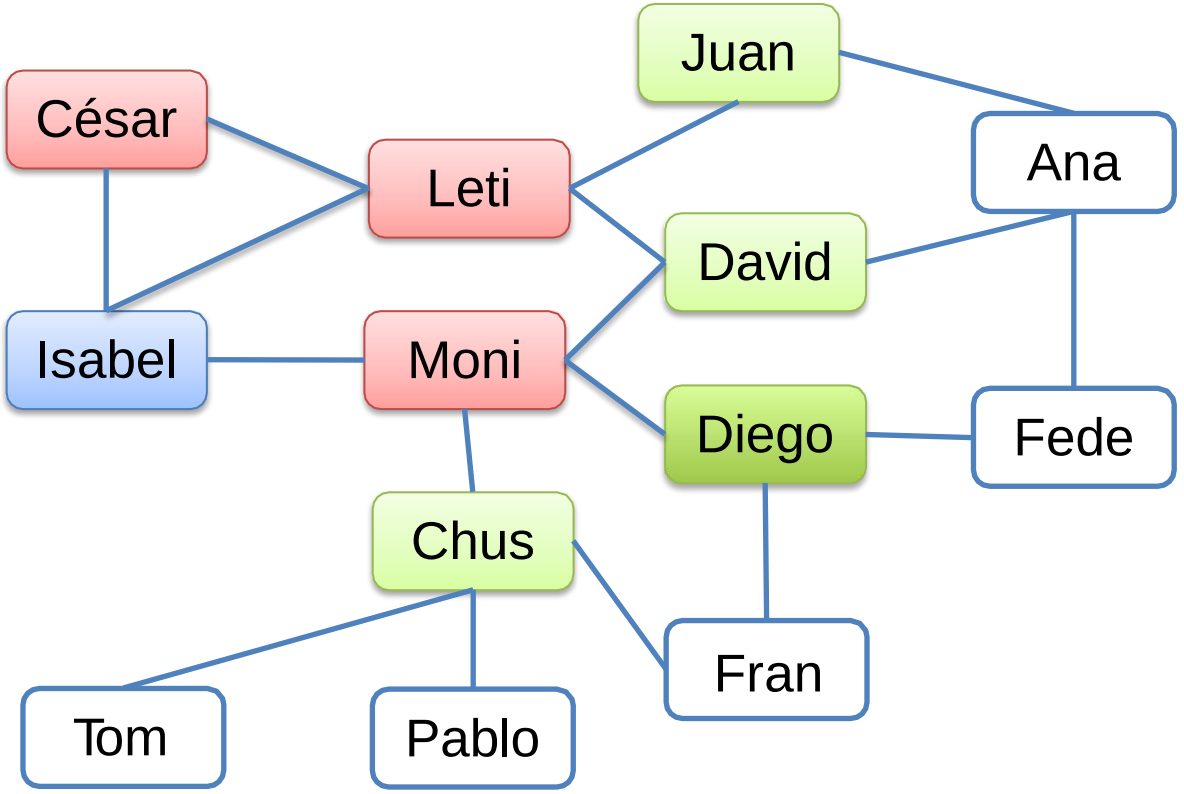


¿Cómo sugerir nuevos amigos a Isabel?

Conceptos sobre grafos

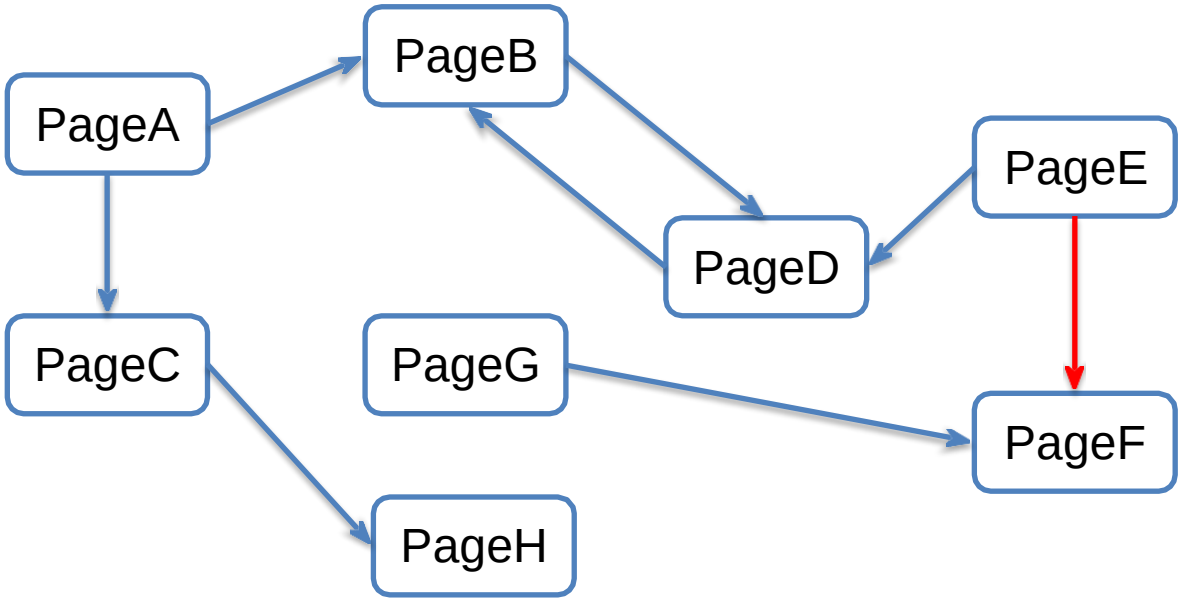


Conceptos sobre grafos



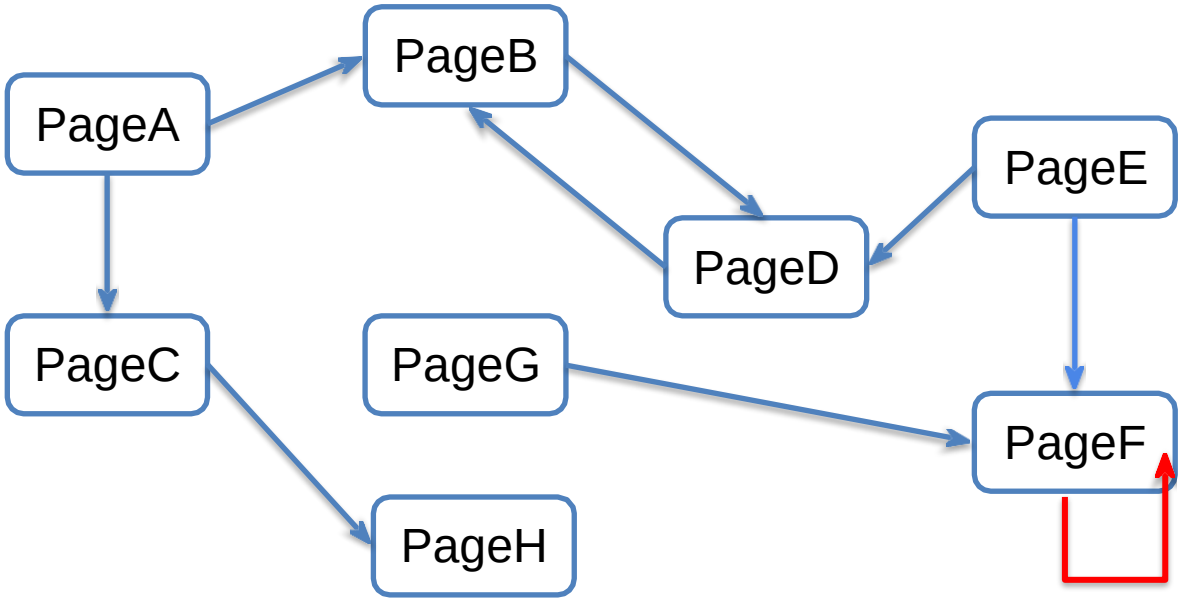
Encontrar todos los nodos para los que exista un camino de longitud 2

Conceptos sobre grafos



La Web se puede representar como un grafo dirigido. Los vértices son las páginas web y las conexiones entre estas son las aristas del grafo.

Conceptos sobre grafos

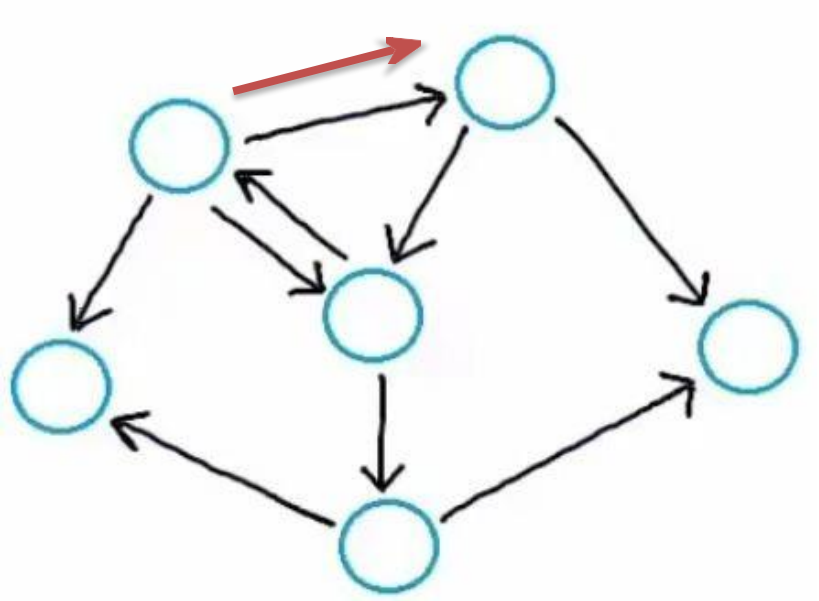


Bucle (loop)

Una página puede contener un enlace a ella misma. Ese tipo de aristas son conocidas como **bucles** (**loop**) y son aristas que conectan un vértice consigo mismo.

Conceptos sobre grafos

Aristas múltiples (aristas paralelas)



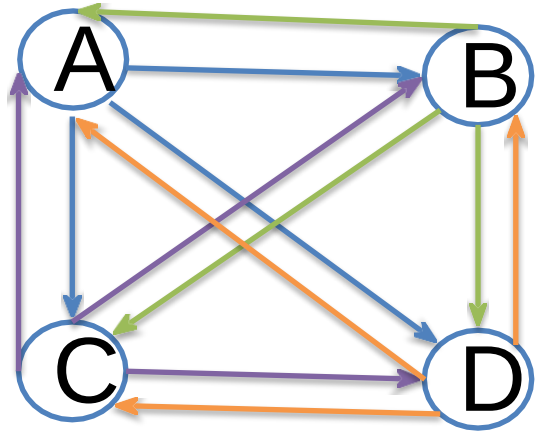
Conceptos sobre grafos

- Los bucles y las aristas paralelas tienden a hacer más complejos los algoritmos de grafos.

- Un **grafo simple** es un grafo que no tiene bucles ni aristas paralelas.

Conceptos sobre grafos

- ¿Cuál es el número mínimo y máximo de aristas en un **grafo simple dirigido**?



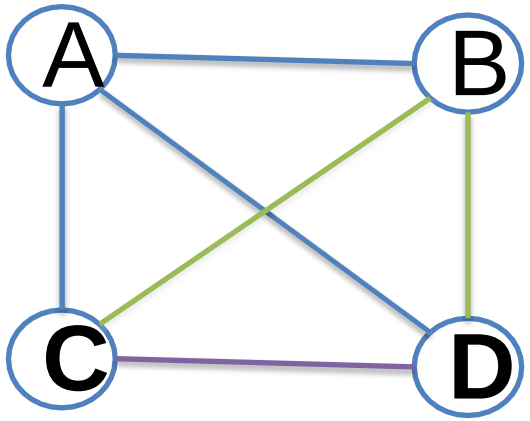
|V| = 4
|A| = 0 (mínimo)

|V| = 4
|A| = 12 (máximo)

Si *|V| = n, cada vértice podría tener un máximo de n-1 aristas. Por lo tanto,* ***0<= |A| <=n(n-1)***

Conceptos sobre grafos

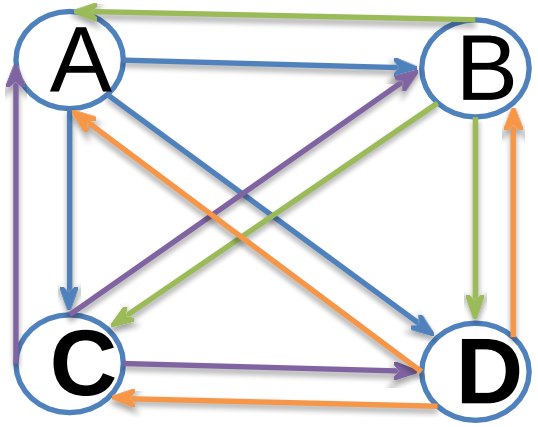
- ¿Cuál es el número máximo de aristas en un **grafo simple no dirigido**?



Si */V/ = n, cada vértice podría tener n-1 aristas.* ***0<= /A/<=n(n-1)/2***

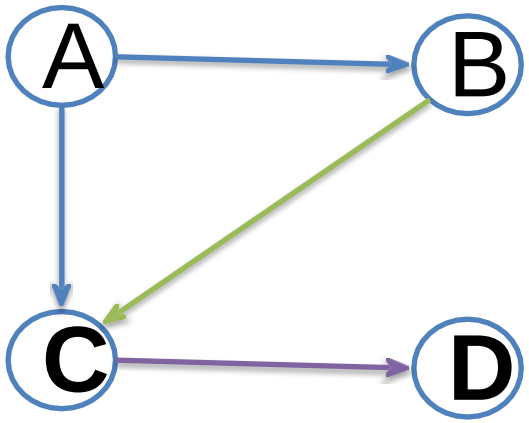
Conceptos sobre grafos

- Un grafo es **denso** si el número de sus aristas es cercano a su número máximo posible (n(n-1) o n(n-1)/2) ($\approx n^2$)



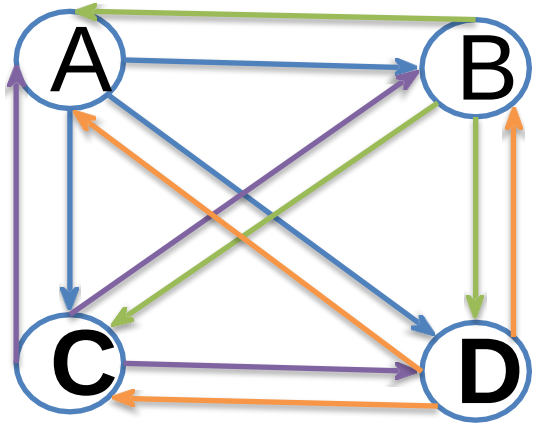
Conceptos sobre grafos

- Un grafo es **disperso** si el número de sus aristas es cercano a el número de sus vértices ($\approx n$)

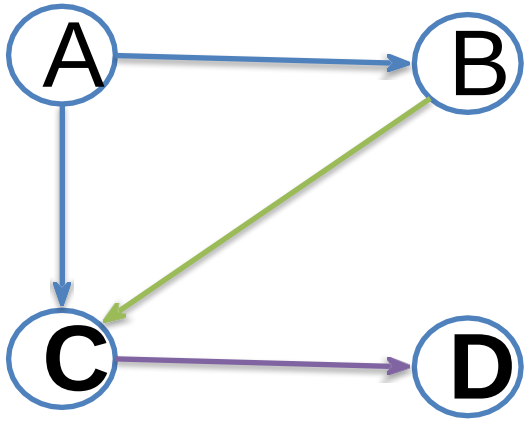


Conceptos sobre grafos

- Conocer si un grafo es denso o disperso nos ayudará a elegir la implementación más apropiada.



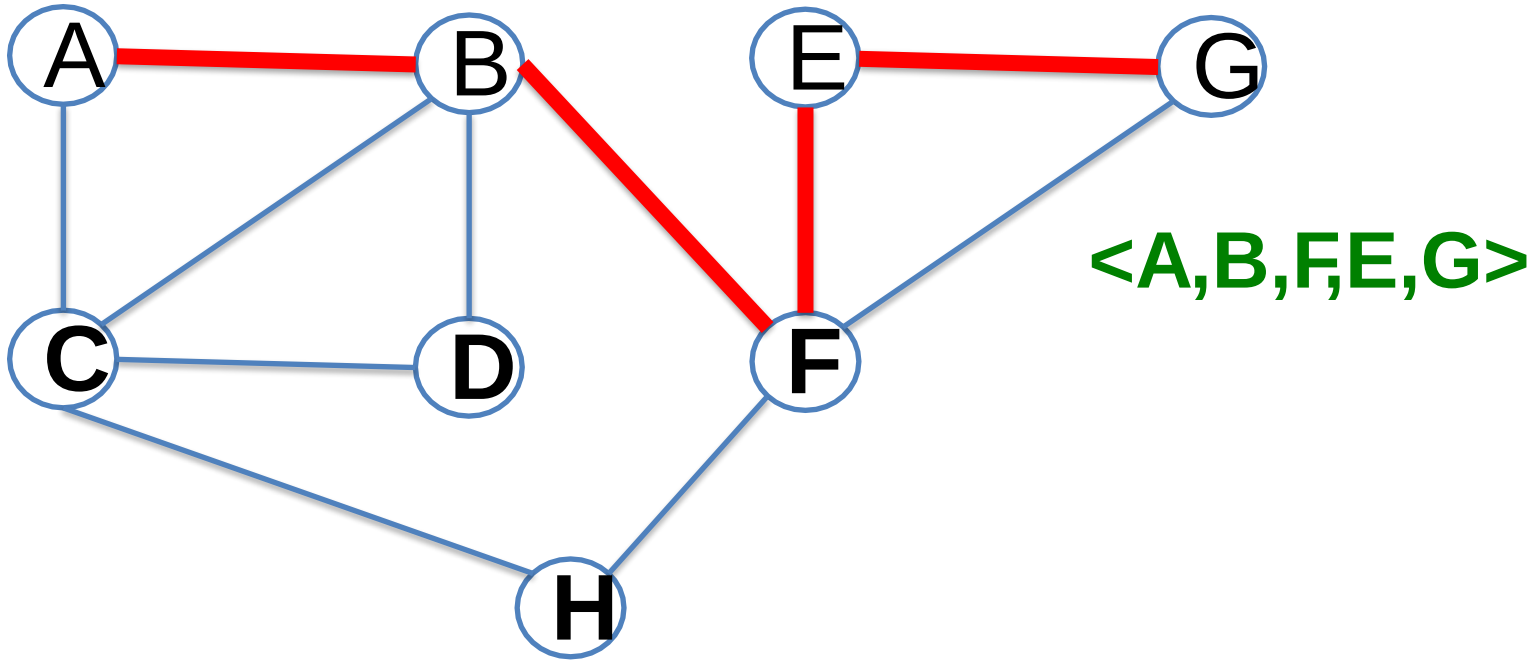
dense



sparse

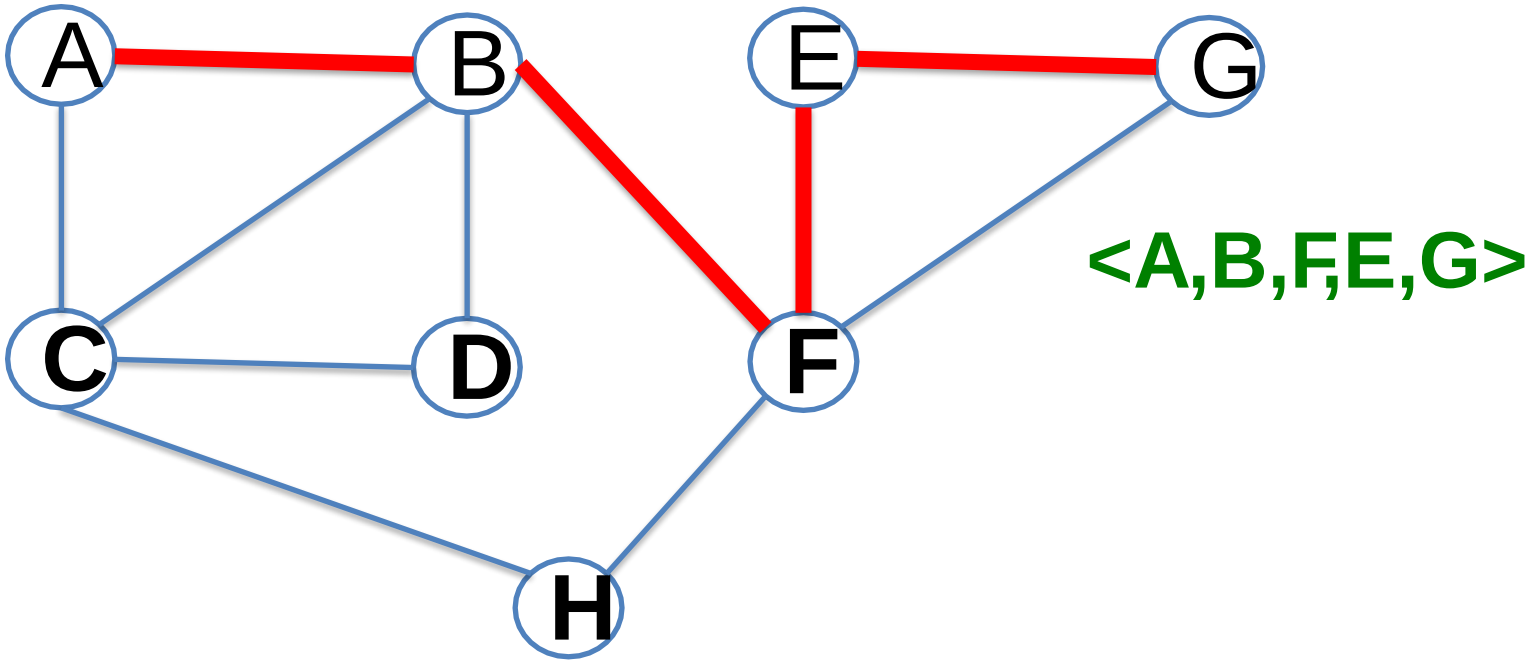
Conceptos sobre grafos

- Un **camino** es una secuencia de vértices tal que exista una arista entre cada vértice y el siguiente.



Conceptos sobre grafos

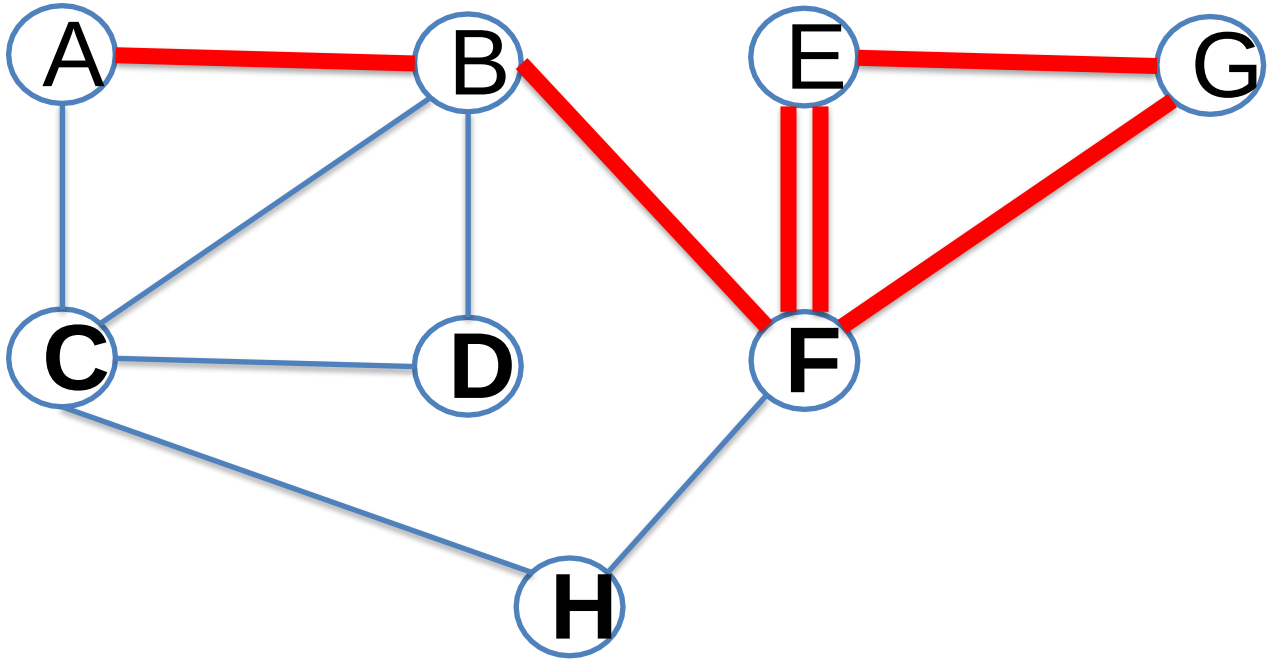
- Un **camino simple** es aquel que no repite vértices en su recorrido.



Conceptos sobre grafos

- Este camino no es simple porque hay dos vértices repetidos

  <A, B, F, E, G, F, E>



TAD Grafo

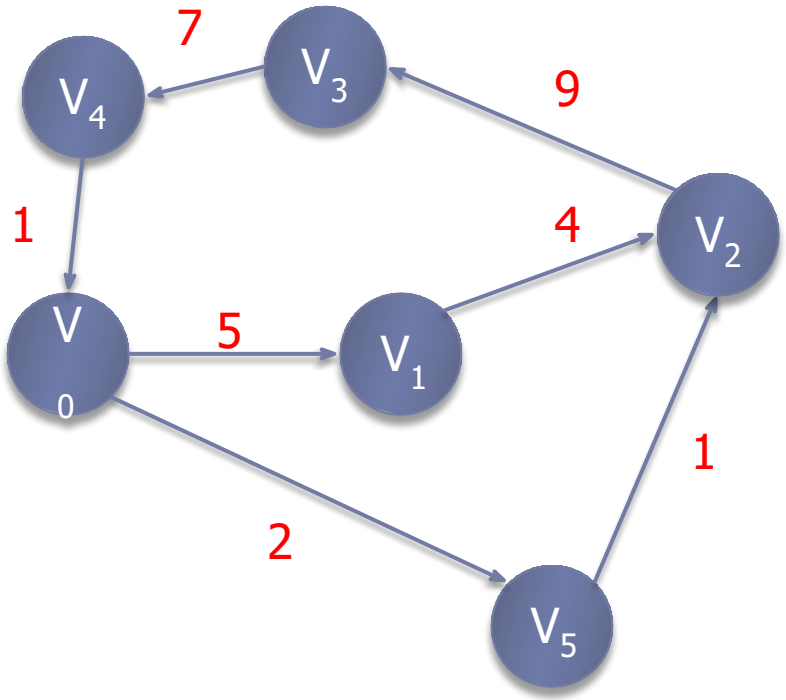
```
public interface IGraph {

    //devuelve el número de vértices
    public int sizeVertices();
    //devuelve el número de aristas
    public int sizeEdges();
    //muestra los vertices y sus aristas
    public void show();

    //devuelve el grado del vértice i
    public int getDegree(int i);
    //devuelve el grado de entrada del vértice i
    public int getInDegree(int i);
    //devuelve el grado de salida del vértice i
    public int getOutDegree(int i);
```

TAD Grafo (cont.)

```
//crea un nuevo vértice
public void addVertex();
//añade una arista entre i y j
public void addEdge(int i, int j);
//añade una arista entre i y j con peso w
public void addEdge(int i, int j, float w);
//borra la arista entre i y j
public void removeEdge(int i, int j);

//comprueba si existe una arista entre i y j
public boolean isEdge(int i, int j);
//devuelve el peso asociado a la arista (i,j).
public Float getWeightEdge(int i, int j);
//devuelve un array con los vértices adyacentes a i
public int[] getAdjacents(int i);
```

Implementación basada en matriz

- **La matriz de adyacencias**

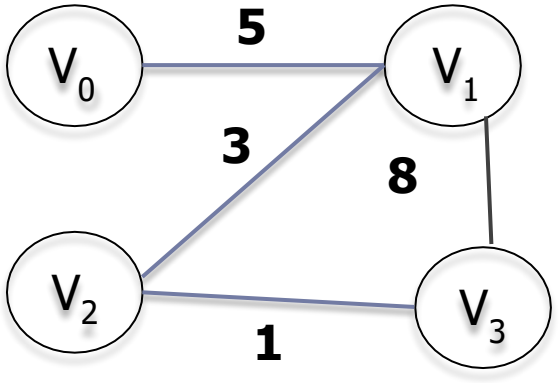


	V_0	V_1	V_2	V_3	V_4	V_5
V_0		5				2
V_1			4			
V_2				9		
V_3					7	
V_4	1					
V_5			1			

Implementación – Matriz de adyacencias

- Un grafo puede ser representado como una matriz cuadrada nxn, siendo n el número de vértices del grafo.

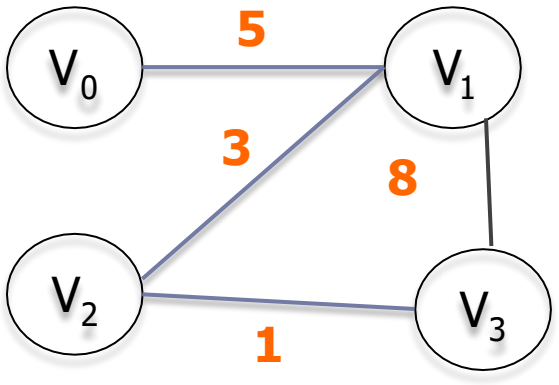
- Cada vértice v es representado por un entero (índice de v), en el rango {0,1,..,n-1} siendo n el número de vértices



- V_0 -> índice 0
- V_1 -> índice 1
- V_2 -> índice 2
- V_3 -> índice 3

Implementación – Matriz de adyacencias

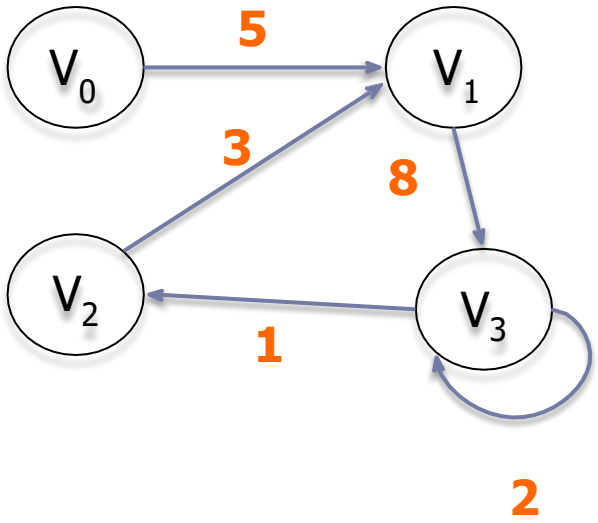
- La matriz se puede implementar como un array bidimensional n x n M, tal que el elemento **M[i,j]** guarda información sobre la arista (v,w), si existe, donde v es el vértice con índice i y w es el vértice con índice j.



	0	1	2	3
0		5		
1	5		3	8
2		3		1
3		8	1	

Si el grafo no es dirigido la matriz es simétrica

Implementación – Matriz de adyacencias

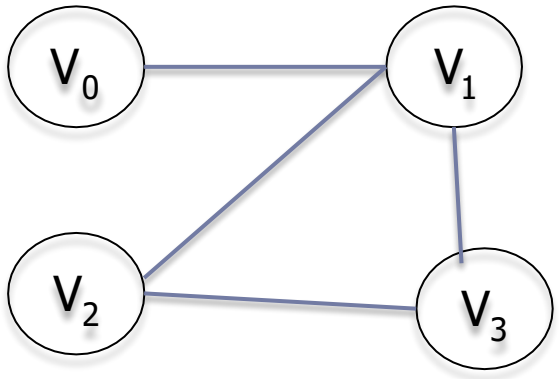


Si el grafo es dirigido la matriz NO es simétrica

	0	1	2	3
0		5		
1				8
2		3		
3			1	2

Implementación – Matriz de adyacencias

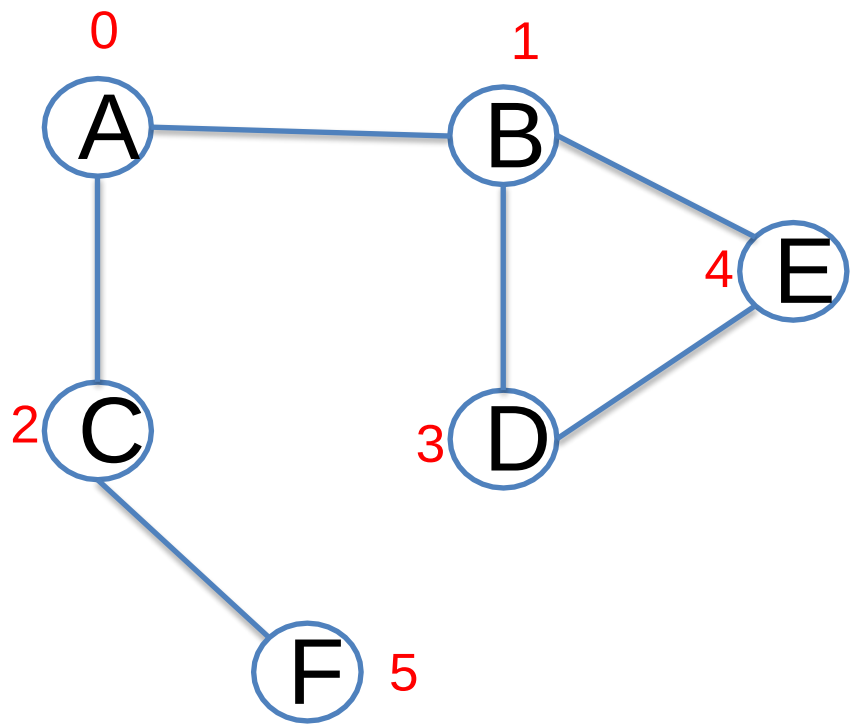
- Si es un grafo no etiquetado, el grafo se podría representar con una matriz de booleanos,



	0	1	2	3
0	false	true	false	false
1	true	false	true	true
2	false	true	false	true
3	false	true	true	false

Si el grafo no es dirigido la matriz es simétrica

Matriz de Adyacencia

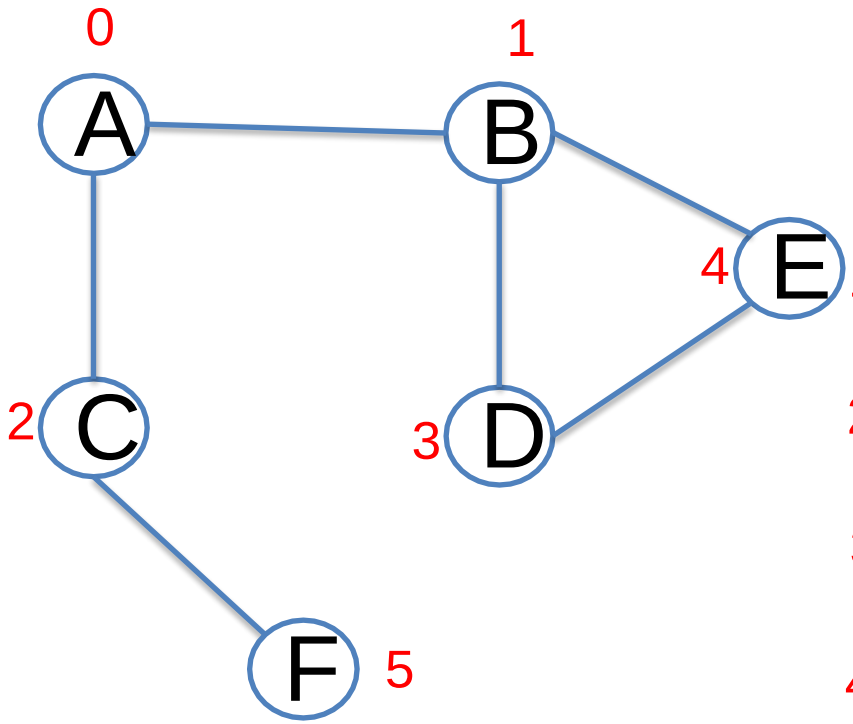


Lista de vértices

0	A
1	B
2	C
3	D
4	E
5	F

Cada vértice es representado por un índice. Podemos usar una lista de Python (array) para almacenar los vértices.

Matriz de Adyacencia (grafo no dirigido)

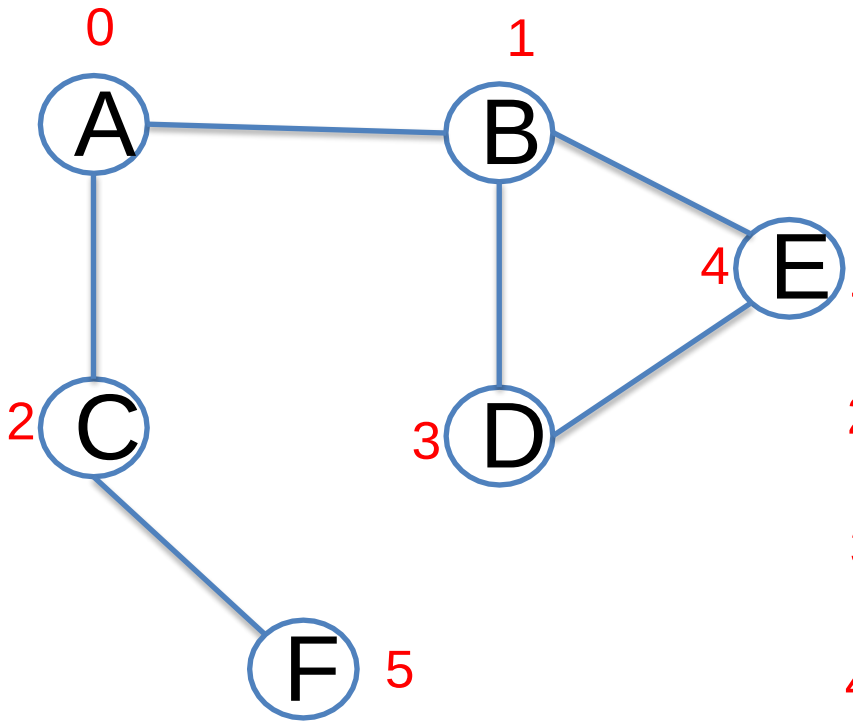


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

$$Mij = \begin{cases} 1, & \text{if } \{i, j\} \text{ si es una arista} \\ 0, & \text{eoc} \end{cases}$$

Matriz de Adyacencia (grafo no dirigido)

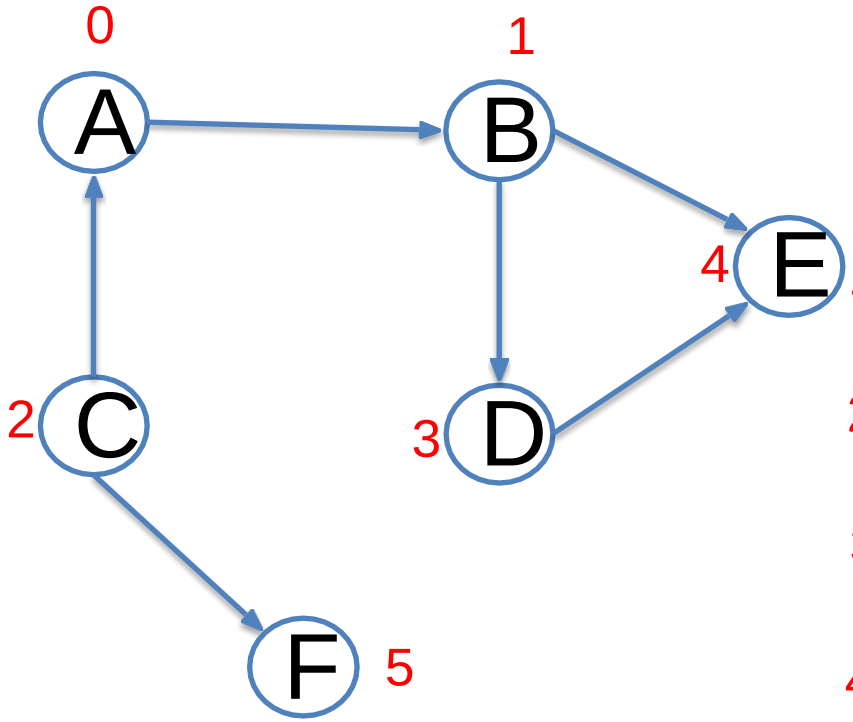


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

En los grafos no dirigidos, la matriz de adyacencia va a ser simétrica: Mij =Mji

Matriz de Adyacencia (grafo dirigido)

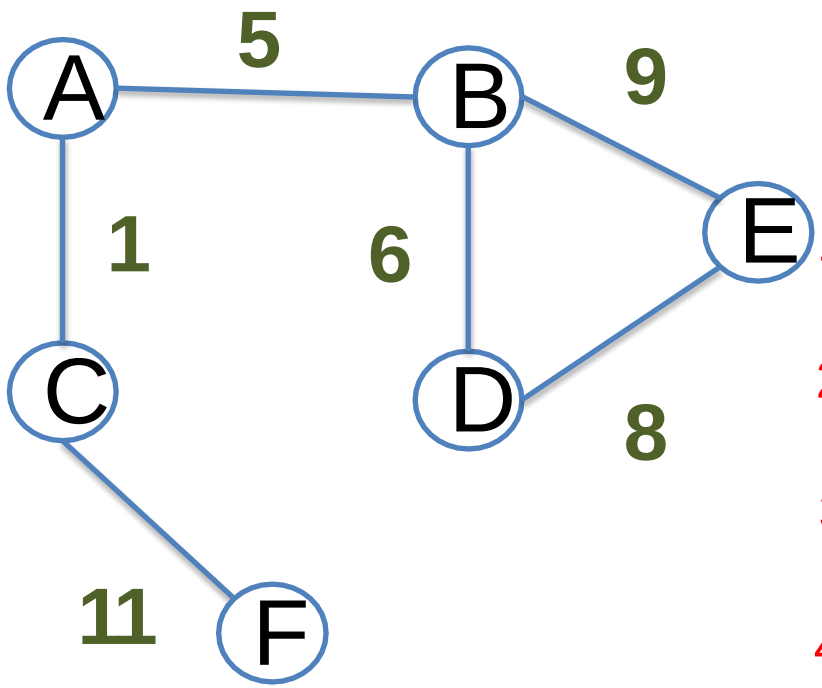


		0	1	2	3	4	5
0	A	0	1	0	0	0	0
1	B	0	0	0	1	1	0
2	C	1	0	0	0	0	0
3	D	0	0	0	0	1	0
4	E	0	0	0	0	0	0
5	F	0	0	0	0	0	0

Mij = { 1, if (i, j) si es una arista
0, eoc

En este caso, **la matriz no es simétrica**

Matriz de Adyacencia (grafo no dirigido)

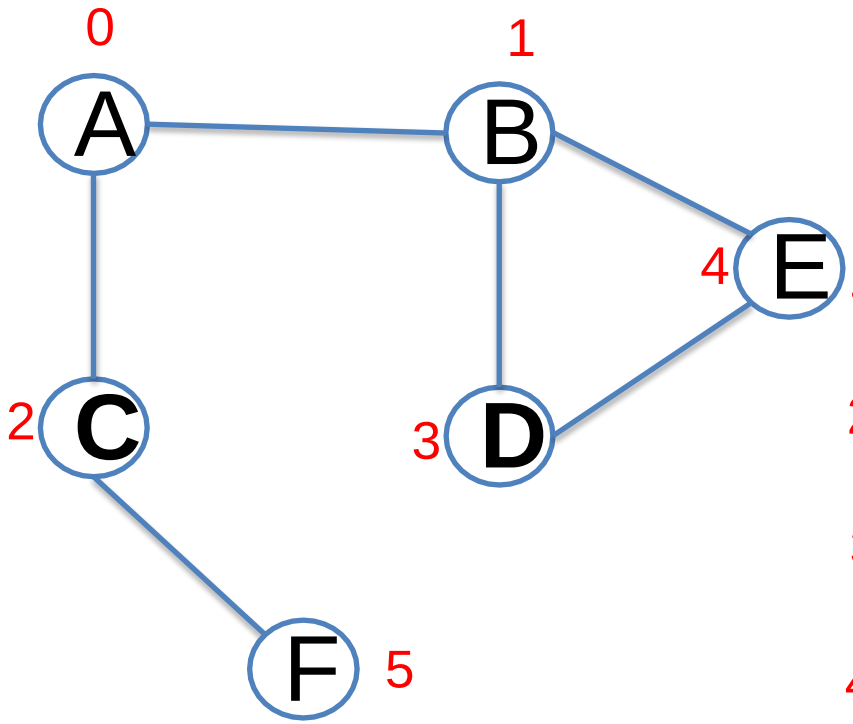


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	∞	5	1	∞	∞	∞
1	5	∞	∞	6	9	∞
2	1	∞	∞	∞	∞	11
3	∞	6	∞	∞	8	∞
4	∞	9	∞	8	∞	∞
5	∞	∞	11	∞	∞	∞

¿Cómo representar grafos ponderados?

Matriz de Adyacencia - Complejidad Espacial



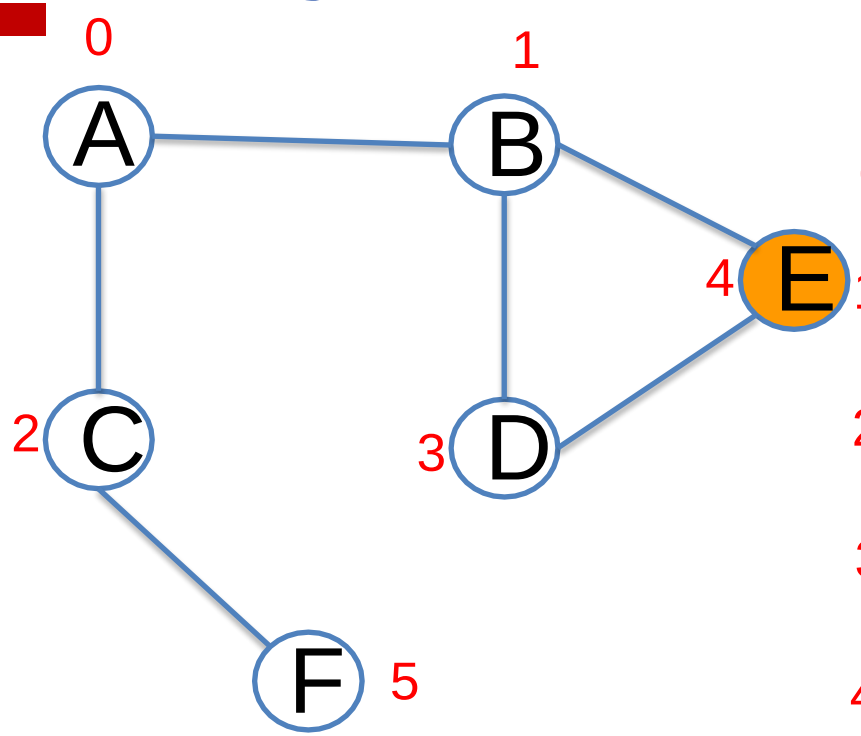
0	A
1	B
2	C
3	D
4	E
5	F

0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

Complejidad espacial

If $|V| = n$, $O(n^2)$

Matriz de Adyacencia - Complejidad Temporal



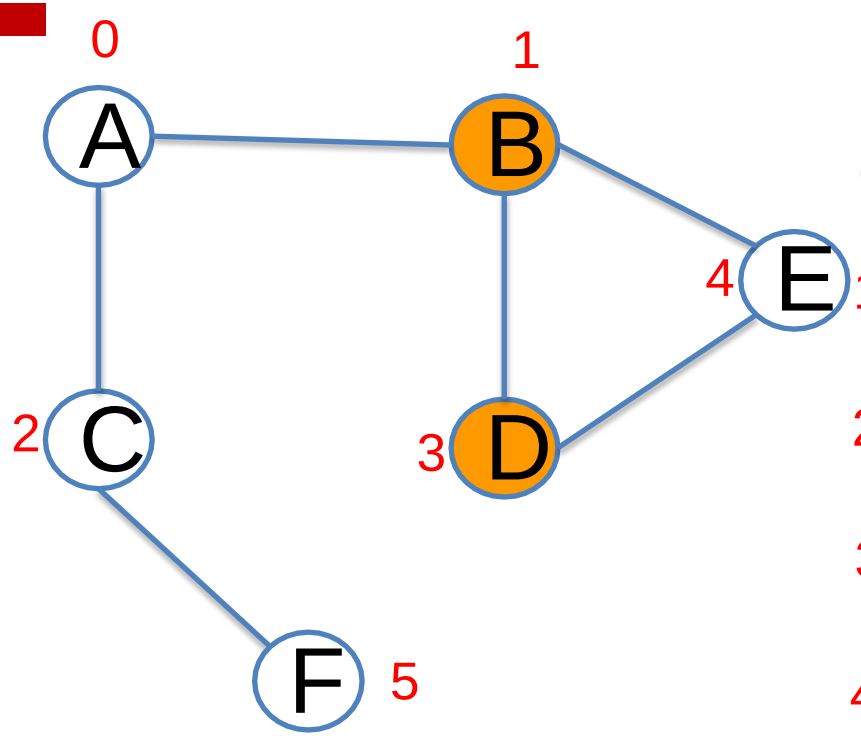
0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

Complejidad temporal
¿buscar los vecinos de E?

O(n)

Matriz de Adyacencia - Complejidad Temporal



		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

¿son vecinos? **O(n) + O(1)**

Implementación – Matriz de adyacencias

- Vamos a ver una implementación para un grafo no ponderado.
- La matriz puede ser almacenada en un array bidimensional de booleanos (true indicará que existe arista y false que no existe).
- La creación de nuevos vértices podría implicar la necesidad de modificar el tamaño asignado a la matriz.
- Para evitarlo, vamos a definir un atributo que almacene el **número máximo de vértices** (en ningún caso, se permitirá añadir un nuevo vértice cuando ese umbral se haya alcanzando) y otro atributo que almacene el número actual de vértices.

Implementación – Matriz de adyacencias

```
public class GraphMA implements IGraph {

    boolean matrix[][];
    //maximum number of vertices
    int maxVertices;
    //current number of vertices
    int numVertices;
    //true if the graph is directed, false eoc
    boolean directed;
```

Implementación – Matriz de adyacencias

```
public GraphMA    (int n, int max, boolean d) {
    //We checks if the values are right for the graph
    if (max<=0)
        throw new IllegalArgumentException("Negative maximum number of vertices!!!");
    if (n<=0)
        throw new IllegalArgumentException("Negative number of vertices!!!.");
    if (n>max)
        throw new IllegalArgumentException("number of vertices can never be greater than the maximum.");


    maxVertices=max;
    numVertices=n;
    matrix=new Float[maxVertices][maxVertices];
    directed=d;
}
```

- Primero deberemos comprobar que todos los argumentos del constructor reciben valores apropiados: tanto el número máximo como el número de vértices debe ser siempre un número positivo.
- Además, el número de vértices nunca deberá sobrepasar el número máximo de vértices.
- El constructor crea el array bidimensional. Por defecto, todos las posiciones son inicializadas a false.

Implementación – Matriz de adyacencias

```
public void addVertex() {
    if (numVertices==maxVertices) {
        System.out.println("Cannot add new vertices!!!");
        return;
    }
    numVertices++;
}
```

- Lo primero que tenemos que hacer es comprobar que el nuevo número total de vértices no va a sobrepasar el número máximo permitido.
- Por último, sólo tendremos que incrementar en uno el número actual de vértices.
- No hace falta inicializar la matriz para el nuevo vértice, porque por defecto todas sus posiciones en la matriz son false.

Implementación – Matriz de adyacencias

```
//check if i is a right vertex
private boolean checkVertex(int i) {
    if (i>=0 && i<numVertices) return true;
    else return false;
}
```

- Vamos a usar un método auxiliar para comprobar si un índice representa o no un vértice en el grafo.
- Para que sea un vértice del grafo siempre deberá ser positivo y menor que numVertices, porque los vértices del grafo toman valores en el rango [0, numVertices-1].

Implementación – Matriz de adyacencias

```
public void addEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    matrix[i][j]=true;
    if (!directed) matrix[j][i]=true;
}
```

Una vez comprobado que ambos índices son correctos, simplemente lo que tenemos que hacer es actualizar la posición matrix[i,j] a true. Si no es dirigido, también tendremos que poner su posición simétrica

Implementación – Matriz de adyacencias

```
@Override
public boolean isEdge(int i, int j) {
    //checks if the indexes are right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);
    return matrix[i][j];
}
```

- En primer lugar, tenemos que comprobar que los índices i y j son correctos.
- El par (i,j) es un arista si matrix[i,j] guarda true.

Implementación – Matriz de adyacencias

```
@Override
public void removeEdge(int i, int j) {
    //checks if the indexes are right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);
    matrix[i][j]=false;
    if (!directed) matrix[j][i]=false;

}
```

- Primero tenemos que comprobar que son índices válidos
- Una vez comprobado que son índices válidos, basta con modificar el valor del array en esa posición (i,j) a false.
- Si no es dirigido, también tendremos que hacerlo en su elemento simétrico (j,i)

Implementación – Matriz de adyacencias

```
public int sizeVertices() {
    return numVertices;
}

public int sizeEdges() {
    int numEdges=0;
    if (directed) {
        for (int i=0;i<numVertices;i++) {
            for (int j=0;j<numVertices;j++) {
                if (matrix[i][j]!=false) numEdges++;
            }
        }
    } else {
        for (int i=0;i<numVertices;i++) {
            for (int j=i;j<numVertices;j++) {
                if (matrix[i][j]!=false) numEdges++;
            }
        }
    }
    return numEdges;
}
```

- Equivale a contar todos los elementos true en la matriz.

- Si no es dirigido, como la matriz es simétrica, sólo necesitaremos visitar una de las dos partes divididas por la diagonal.

Implementación – Matriz de adyacencias

```
public int getOutDegree(int i) {
    if (!directed) {
        System.out.println("Graph non directed!!!");
        return 0;
    }
    //checks if the vertex is right
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int outdeg=0;
    for (int col=0;col<numVertices;col++) {
        if (matrix[i][col]!=false) outdeg++;
    }
    return outdeg;
}
```

Las filas representan los vértices de origen y las columnas los vértices destino

- Incrementamos 1 por cada columna cuyo índice tenga una arista con i, es decir, matrix[i,col].

Implementación – Matriz de adyacencias

```
public int getInDegree(int i) {
    if (!directed) {
        System.out.println("Graph non directed!!!");
        return 0;
    }
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int indeg=0;
    for (int row=0;row<numVertices;row++) {
        if (matrix[row][i]!=false) indeg++;
    }

    return indeg;
}
```

- Las filas representan los vértices de origen y las columnas los vértices destino

- Incrementamos 1por cada fila cuyo índice tenga una arista con i, es decir, matrix[row,i].

Implementación – Matriz de adyacencias

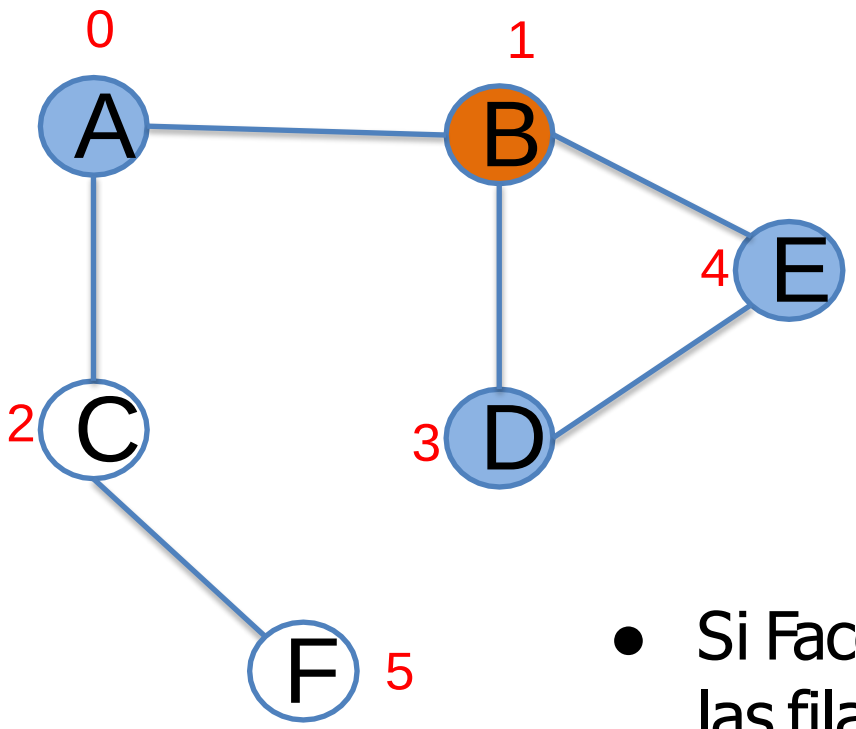
```
public int getDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int degree=0;
    if (directed) degree=getInDegree(i)+getOutDegree(i);
    else {
        for (int row=0;row<numVertices;row++) {
            if (matrix[row][i]!=false) degree++;
        }
    }
    return degree;
}
```

Si el grafo no es dirigido, el grado será la suma del grado de entrada y el grado de salida.
En otro caso, bastará con que contemos las aristas de entrada en ese vértice. También se podría hacer contando las aristas de salida (pero nunca ambas).

Implementación – Matriz de adyacencias

```
//returns an array with the adjacent vertices for i
public int[] getAdjacents(int i) {
    if (!checkVertex(i))
            throw new IllegalArgumentException("Nonexistent vertex  " + i);

    //obtains the number of adjacent vertices,
    //which will be the size of the array
    int numAdjacents=0;
    if (directed) numAdjacents=getOutDegree(i);
    else numAdjacents=getDegree(i);

    int[] adjacents=new int[numAdjacents];

    if (numAdjacents>0) {
        int j=0;
        //gets the edges (i,col) and saves col into adjacents
        for (int col=0; col<numVertices;col++) {
            if (matrix[i][col]!=null) {
                adjacents[j]=col;
                j++;
            }
        }
    }
    //return an array with the adjacent vertices of i
    return adjacents;
}
```

Matriz de Adyacencia - Conclusiones



0	1	2	3	4	5
1	0	0	1	1	0

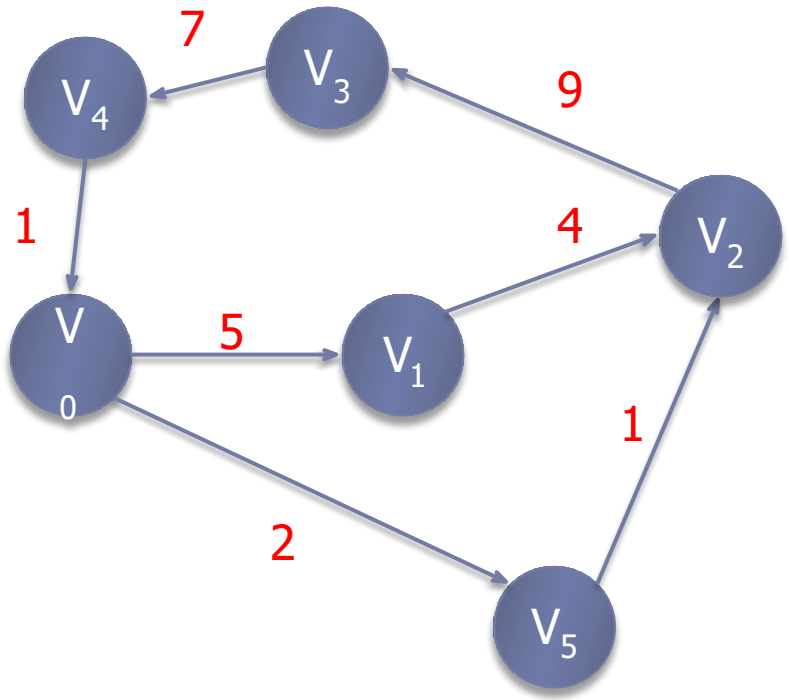
- Si Facebook tiene 1000 millones de usuarios (10^9), las filas de la matriz de adyacencia son de dimensión 10^9
- Si un usuario, B, tiene 1000 amigos, en su fila, habrá:
  - Número de 1s: 1000 =1KB y
  - Número de 0s: 10^9- 1000 = 1GB

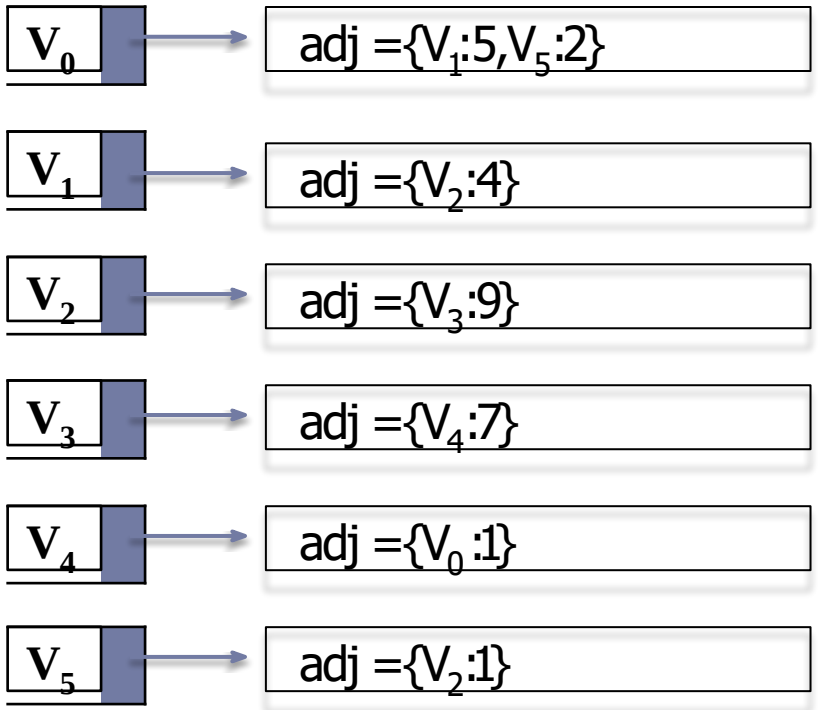
Implementación – Lista de adyacencias

- La matriz de adyacencia consume memoria y la complejidad de las operaciones con la matriz es alta (por ejemplo, el método que muestra la matriz tiene complejidad cuadrática).
- Una lista de adyacencia sólo almacena la información para los aristas existentes, en lugar de almacenar todas las posibles combinaciones como ocurría en la matriz de adyacencias.
- Necesita menos espacio de memoria y su coste computacional es menor.

Implementación – Lista de adyacencias

- Lista adyacencias

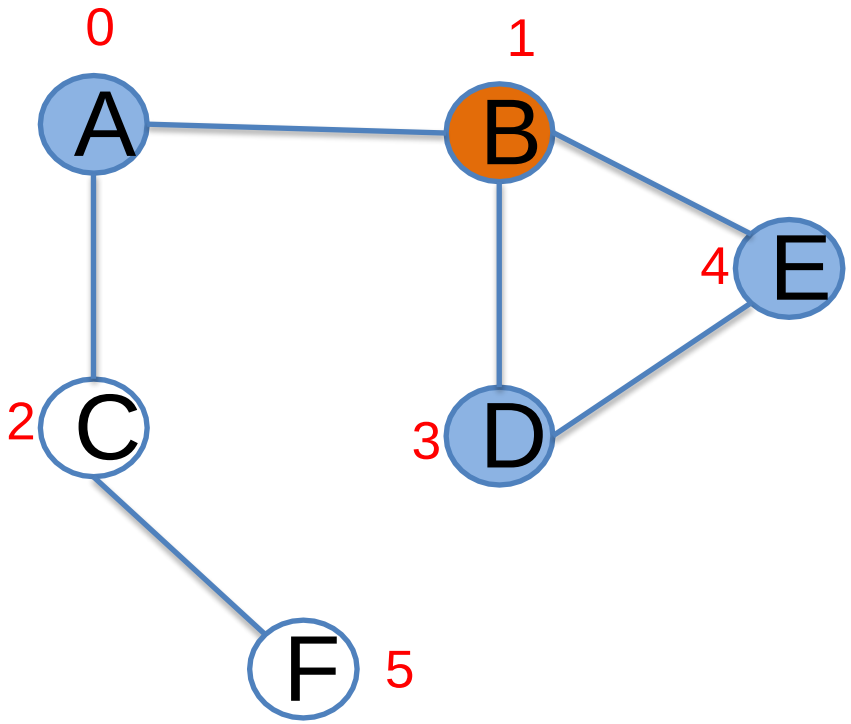




Implementación – Lista de adyacencias

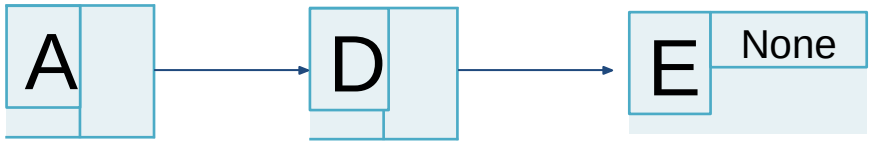
- Si tenemos un número fijo de vértices, el grafo se puede representar como un array de listas enlazadas.
- Cada posición del array representa un vértice y almacena la referencia a la lista de vértices adyacentes a dicho vértice (implementada como lista enlazada).
- Cada uno de los nodos almacenará la información sobre el vértice adyacente. Si el grafo es ponderado, también debería almacenar su peso.

Lista de Adyacencia

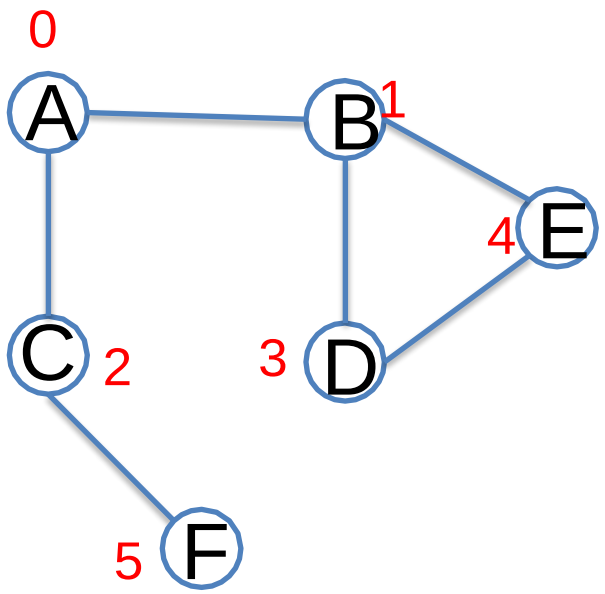


Para representar los **vecinos de B,** sería suficiente con almacenar sus índices en un **lista de python** o en una **lista enlazada.**

Vecinos de B:



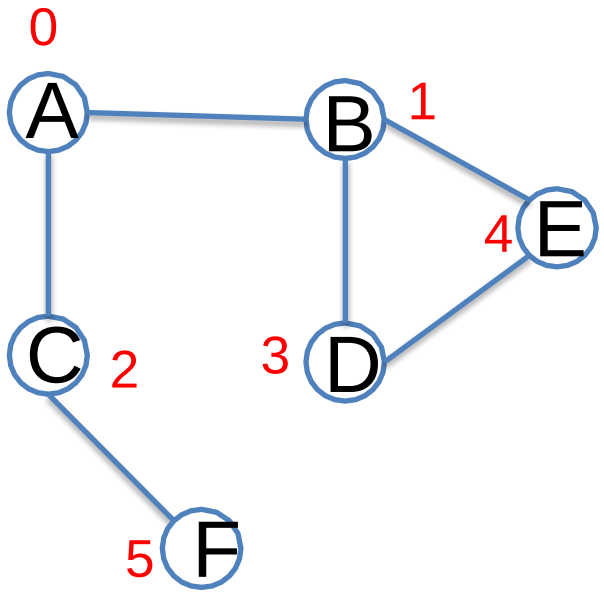
Lista de Adyacencia

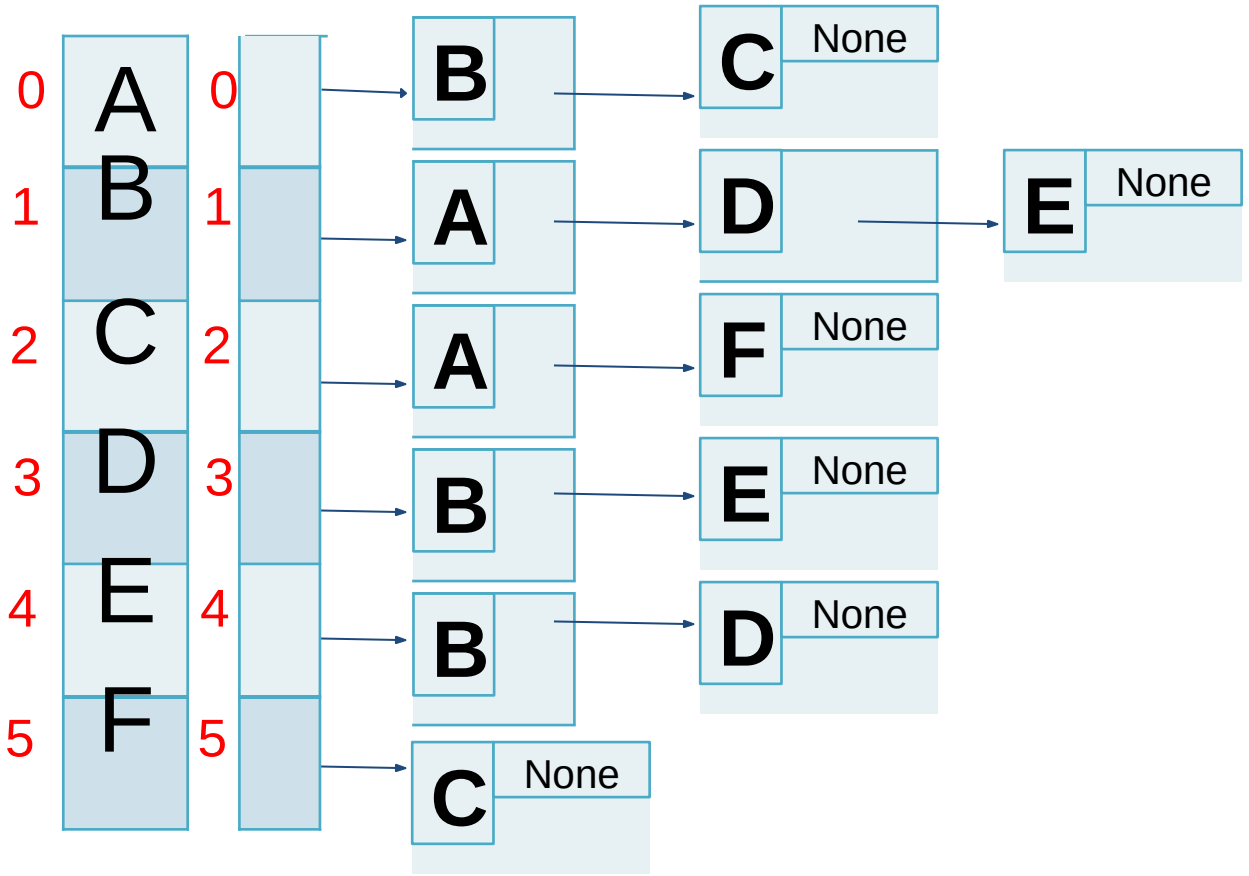


0	A
1	B
2	C
3	D
4	E
5	F

Seguiremos utilizando un array (lista de python) para almacenar los vértices

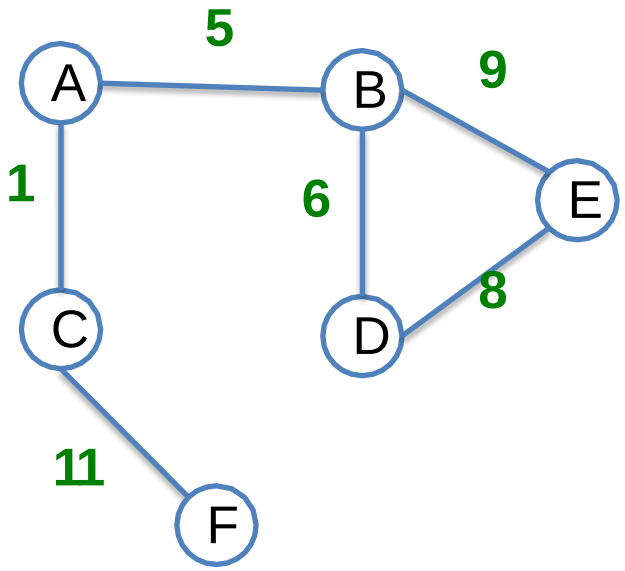
Lista de Adyacencia

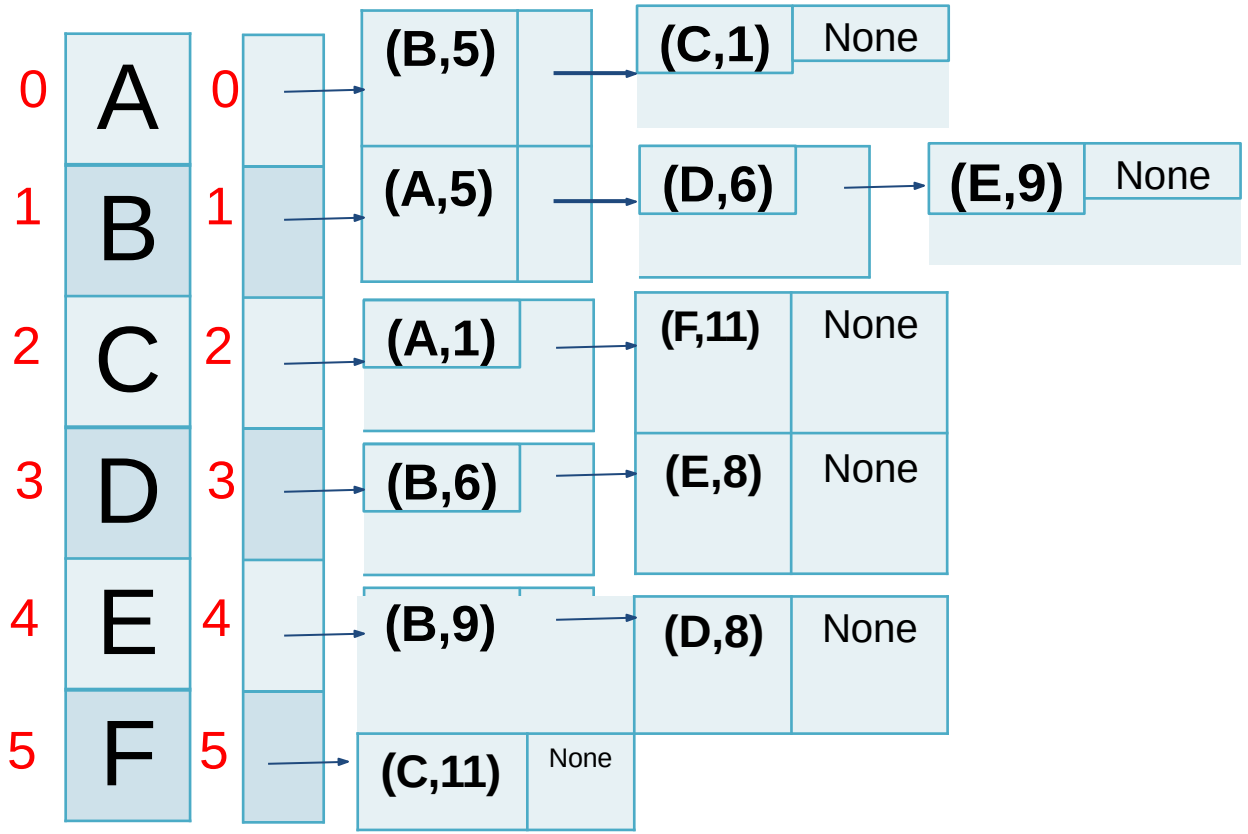




Usaremos un array de listas enlazadas para almacenar los vértices adyacentes a cada vértice.

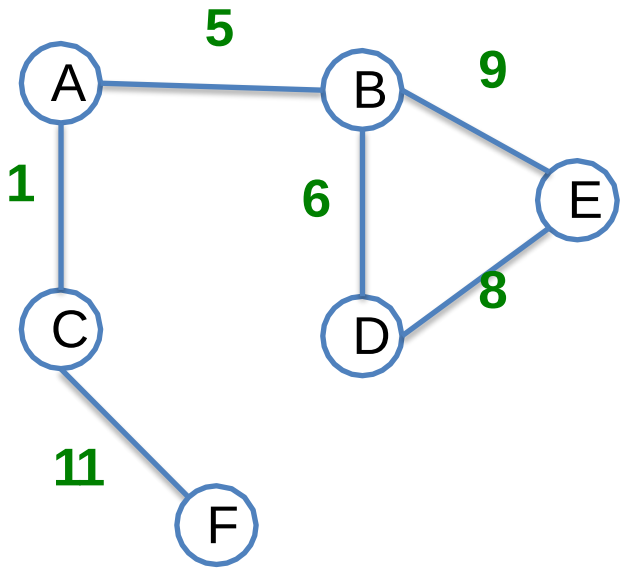
Lista de Adyacencia (grafo ponderado)

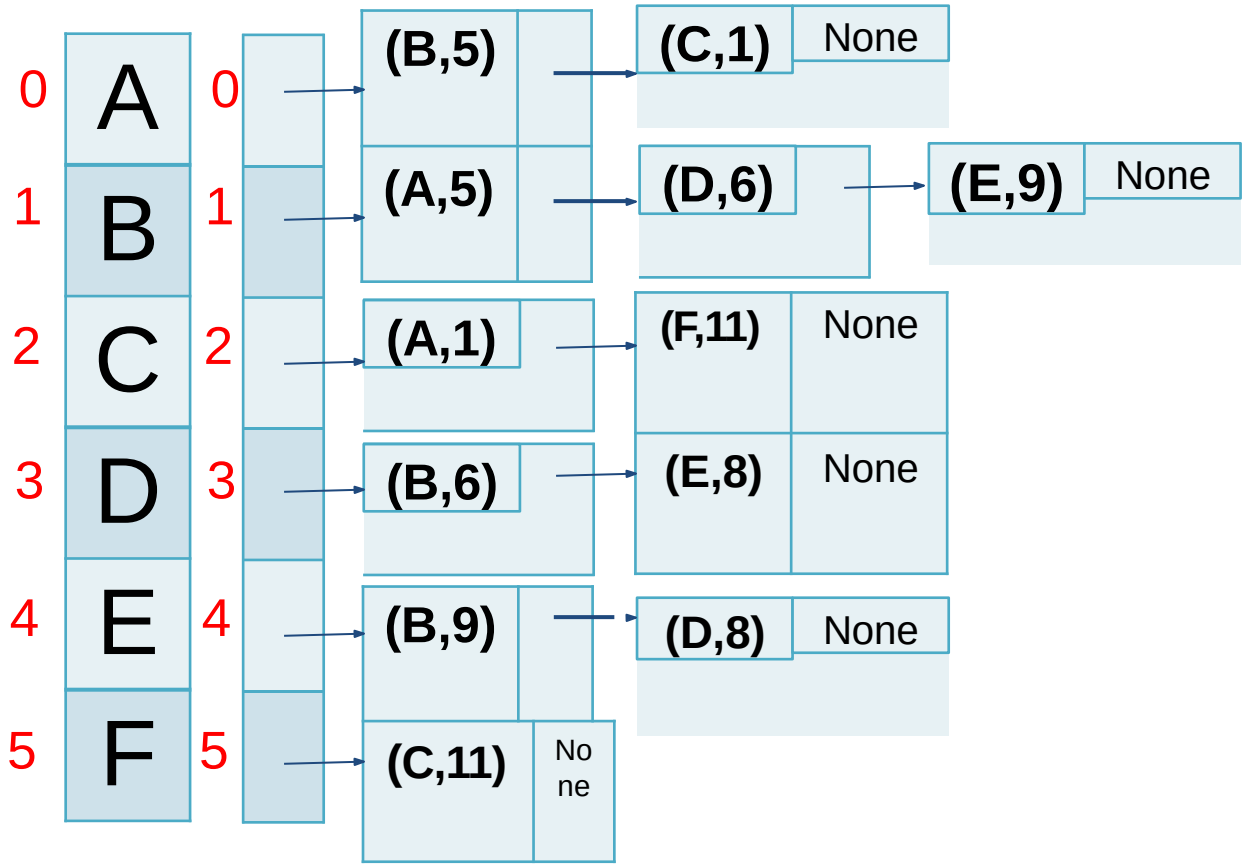




Cada vértice adyacente se representa como un **par (v,w)** donde **v es el vértice adyacente,** y **el peso de la arista.**

Lista de Adyacencia - Complejidad Espacial

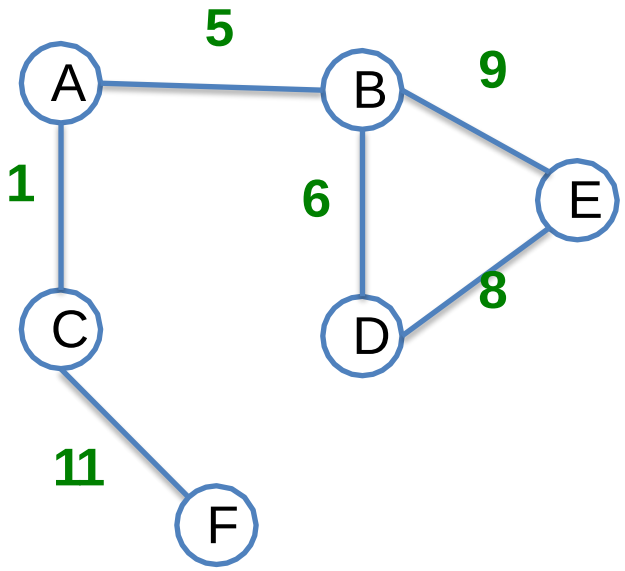


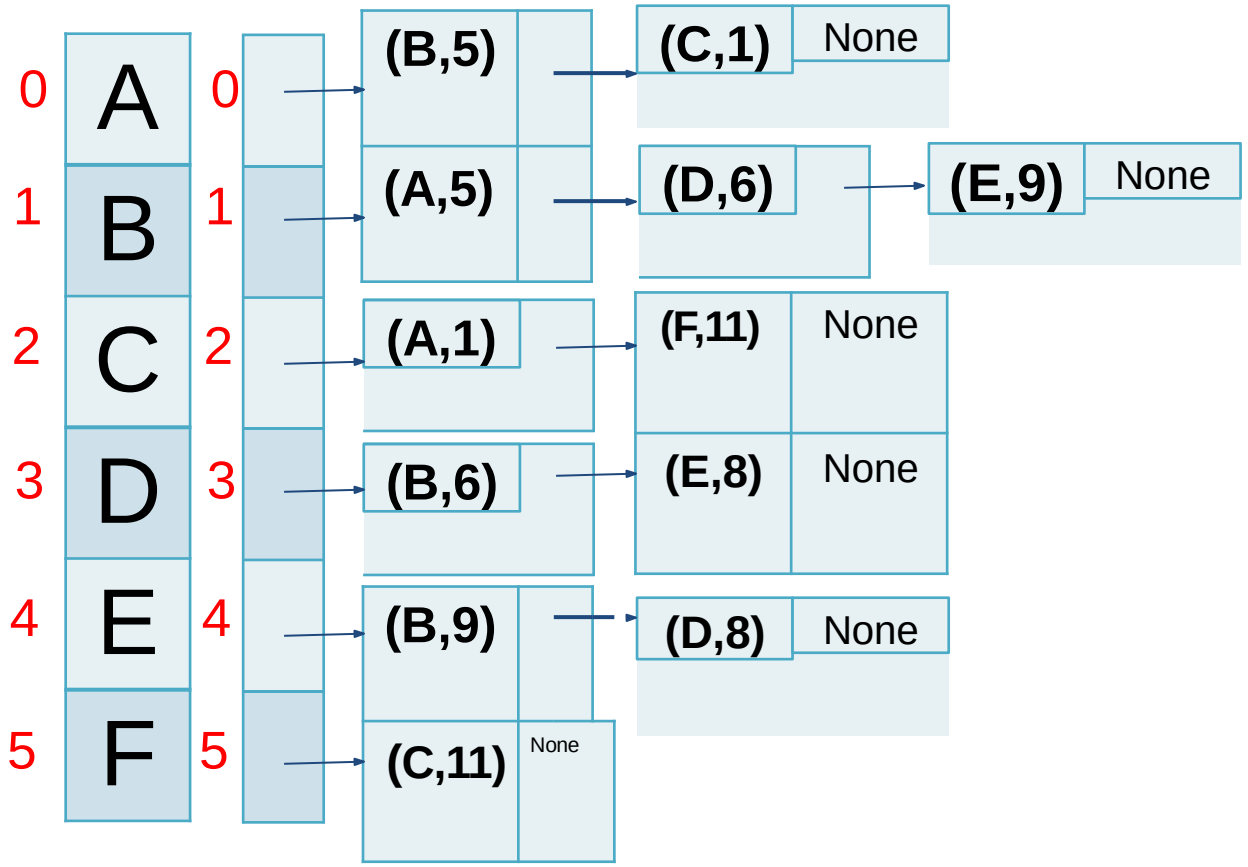


O(|A|)

Recuerda que el número máximo de aristas en un grafo simple es n(n-1)/2 (no dirigido) y n(n-1) (dirigido)

Lista de Adyacencia - Complejidad Espacial

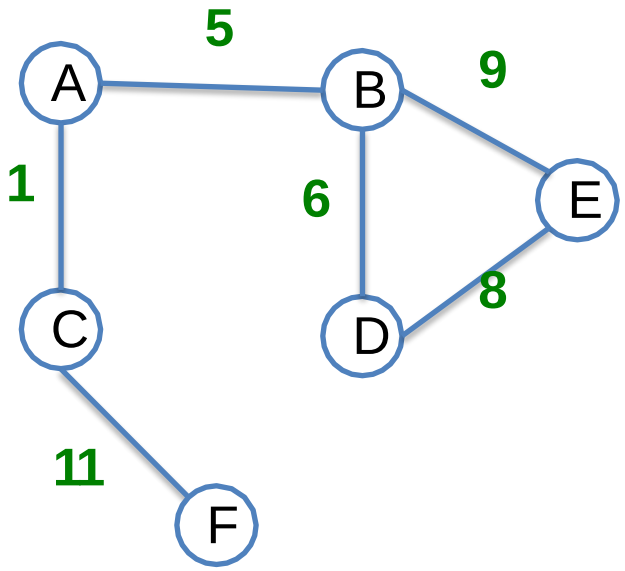


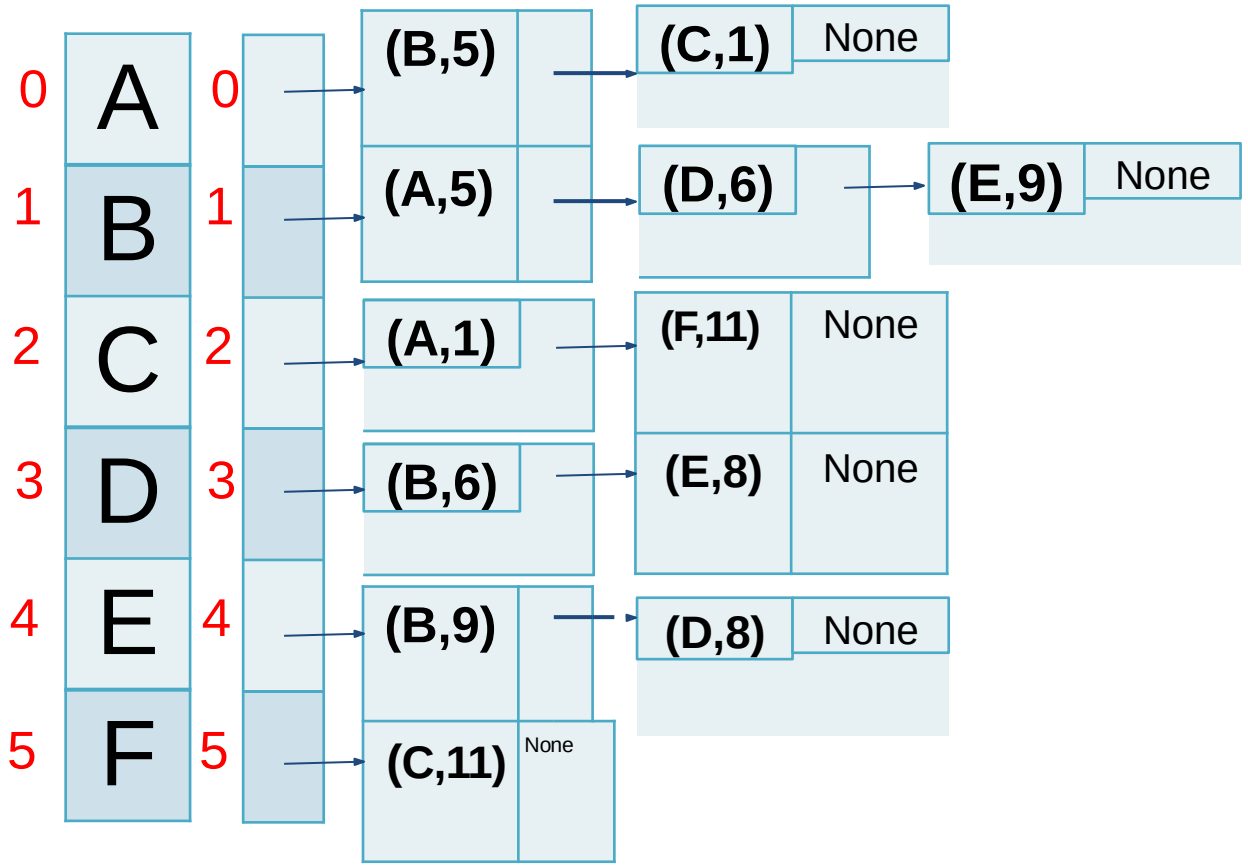


Si el grafo es denso: O(|A|) -> n^2

Si el grafo es disperso: O(|A|) -> n

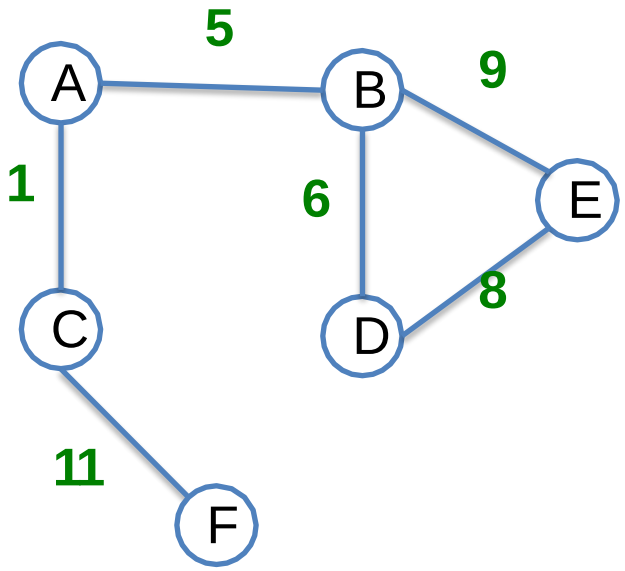
Lista de Adyacencia - Complejidad Temporal

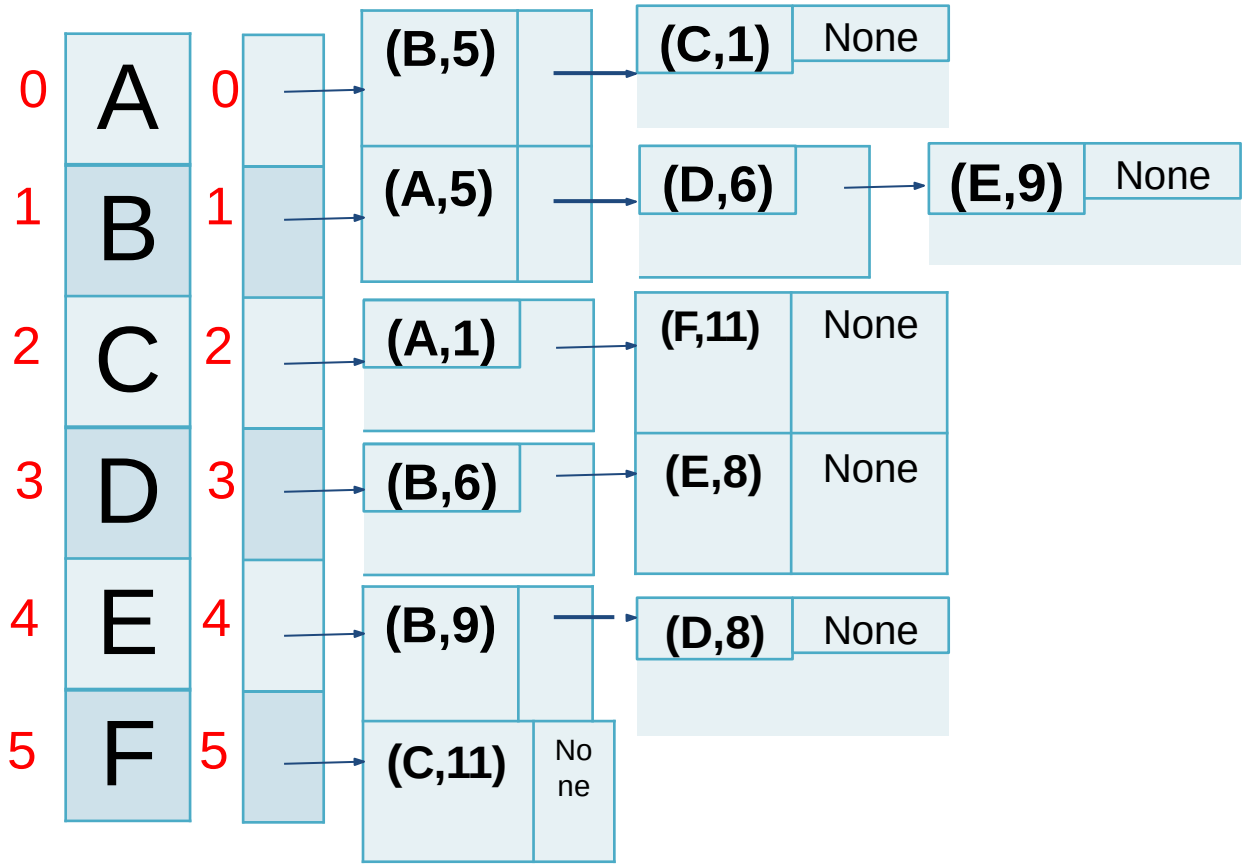




Obtener vértices adyacentes a E?
O(n) (obtener su índice=4) y O(1) devolver la lista de adyacencia asociada al índice

Lista de Adyacencia - Complejidad Temporal





Comprobar si E y F son vecinos?

O(n) (obtener el índice de E) y O(n) comprobar si F está en la lista de adyacencia de E

Lista de Adyacencia - Conclusiones

- En términos de **complejidad temporal**, la lista de adyacencia y la matriz de adyacencia son similares. Para la mayorías de las operaciones, su complejidad es **O(n) donde n=|V|.**

- Sin embargo, en términos de **complejidad espacial**, la lista de adyacencia es una estructura más eficiente **O(|A|).**

- La mayoría de los **grafos reales son escasos**, y por tanto, **|A|≈|V|=n.** Por tanto, la **complejidad espacial será O(n).**

Implementación – Lista de adyacencias

```
import dlist.DListVertex;

public class GraphLAFull implements IGraph {

    int numVertices;
    int maxVertices;

    DListVertex[] vertices;
    boolean directed;
```

Implementación – Lista de adyacencias

```
public GraphLAFull(int n, int max, boolean d) {
    if (max<=0)
        throw new IllegalArgumentException("Negative maximum number of vertices!!!");
    if (n<=0)
        throw new IllegalArgumentException("Negative number of vertices!!!.");
    if (n>max)
        throw new IllegalArgumentException("number of vertices can never "
                + "be greater than the maximum.");

    maxVertices=max;
    vertices=new DListVertex[maxVertices];
    numVertices=n;
    //creates each list
    for (int i=0; i<numVertices;i++) {
        vertices[i]=new DListVertex();
    }
    directed=d;
}
```

Implementación – Lista de adyacencias

Comprobamos que los índices son correctos.

```
public void addEdge(int i, int j, float w) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    vertices[i].addLast(j,w);
    //if it is a non-directed graph
    if (!directed) vertices[j].addLast(i,w);
}
```

Tenemos que añadir el vértice j a la lista de vértices adyacentes del vértice i (que está almacenada en vertices[i]).

Si el grafo no es dirigido, deberemos también almacenar el vértice i como adyacente del vértice j

Implementación – Lista de adyacencias

```
public void removeEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    int index=vertices[i].getIndexOf(j);
    vertices[i].removeAt(index);

    if (!directed) {
            index=vertices[j].getIndexOf(i);
            vertices[j].removeAt(index);
    }
}
```

Si el grafo no es dirigido, deberemos también borrar el vértice i como adyacente del vértice j

Implementación – Lista de adyacencias

```
public boolean isEdge(int i, int j) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    if (!checkVertex(j))
        throw new IllegalArgumentException("Nonexistent vertex  " + j);

    boolean result=vertices[i].contains(j);
    return result;
}
```

Implementación – Lista de adyacencias

```
public int getOutDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);

    int outdegree=0;
    outdegree=vertices[i].getSize();
    return outdegree;
}

public int getInDegree(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    int indegree=0;
    for (int j=0; j<numVertices;j++) {
        if (vertices[j].contains(i)) indegree++;
    }
    return indegree;
}
```

Sólo para grafos dirigidos

Implementación – Lista de adyacencias

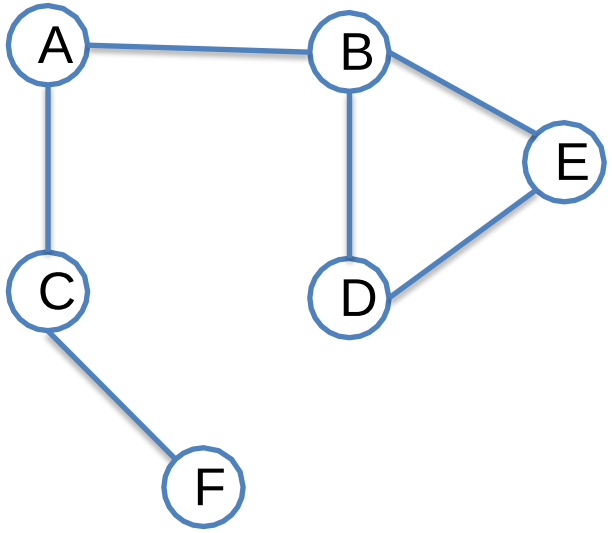
```
public int getDegree(int i) {
    int degree=0;
    if (directed) {
        degree=getOutDegree(i)+getInDegree(i);
    } else degree=vertices[i].getSize();
    return degree;
}
```

El grado de un vértice en un grafo dirigido es igual a la suma de su grado de entrada y de su grado de salida.
En un grafo no dirigido, es suficiente con obtener el número de vértices adyacentes a dicho vértice.

Implementación – Lista de adyacencias

```
public int[] getAdjacents(int i) {
    if (!checkVertex(i))
        throw new IllegalArgumentException("Nonexistent vertex  " + i);
    //gets the number of adjacent vertices
    int numAdj=vertices[i].getSize();
    //creates the array
    int[] adjVertices=new int[numAdj];
    //saves the adjacent vertices into the array
    for (int j=0; j<numAdj; j++) {
        adjVertices[j]=vertices[i].getVertexAt(j);
    }
    //return the array with the adjacent vertices of i
    return adjVertices;
}
```

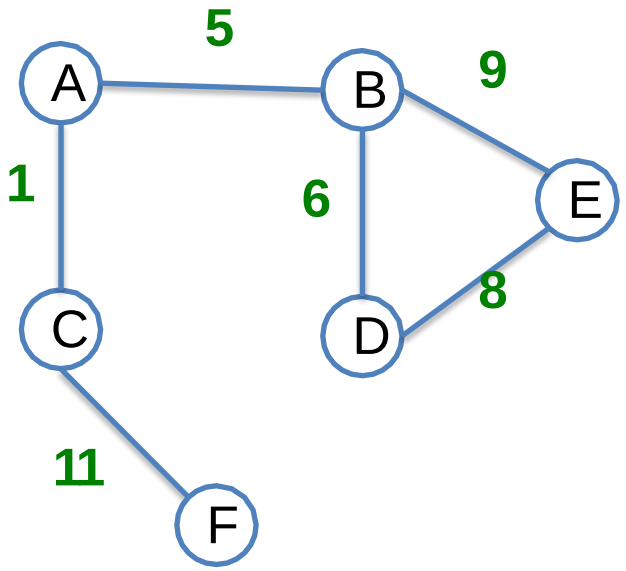
Diccionarios




```
graph ={
'A':['B','C'],

'B':['A','D','E'],

'C':['A','F'],

'D':['B','E'],

'E':['B','D'],

'F':['C']}
```

Los vértices del grafo se van a representar como las claves (keys)del diccionario.
En el diccionario, asociado a cada clave (vértice), se guarda la lista de sus vértices adyacentes

Diccionarios (grafos ponderados)




```
graph = {
'A':[('B',5),('C',1)],

'B':[('A',5),('D',6),('E',9)],

'C':[('A',1),('F',11)],

'D':[('B',6),('E',8)],

'E':[('B',9),('D',8)],

'F':[('C',11)]    }
```

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

Implementar el TAD grafo