


# Grafos

- 
- Es un conjunto de nodos o vértices conectados entre sí mediante arcos o bordes.
  - Los grafos se utilizan para representar relaciones entre diferentes elementos de un sistema, como por ejemplo, las relaciones entre las ciudades en un mapa de carreteras o las relaciones entre las páginas web en la World Wide Web.
  - Los grafos pueden ser dirigidos o no dirigidos, dependiendo de si los arcos tienen una dirección o no.
  - Los grafos también pueden ser ponderados, lo que significa que cada arco tiene un peso asociado.

# Terminología fundamental



- Vértice (o nodo): Es un punto o un punto de interés en el grafo.
- Arco (o arista): Es una línea que une dos vértices.
- Grado de un vértice: Es el número de arcos incidentes en un vértice. El grado de un vértice es igual al número de vecinos de ese vértice.
- Camino: Es una secuencia de arcos que conectan dos vértices.
- Ciclo: Es un camino que comienza y termina en el mismo vértice.
- Grafo dirigido (o digrafo): Es un grafo en el que cada arco tiene una dirección.
- Grafo no dirigido: Es un grafo en el que cada arco no tiene dirección.

# Terminología fundamental



- Grafo completo: Es un grafo en el que cada par de vértices están conectados por un arco.
- Grafo conexo: Es un grafo en el que hay un camino entre cualquier par de vértices.
- Grafo desconectado: Es un grafo en el que no hay un camino entre todos los pares de vértices.
- Grado de un grafo: Es el número de arcos que conectan a un vértice.
- Grafo ponderado: Es un grafo en el que cada arco tiene un peso asociado.
- Componente conexa: Es un subconjunto de vértices en un grafo desconectado que están conectados entre sí.

# Representación de los Grafos



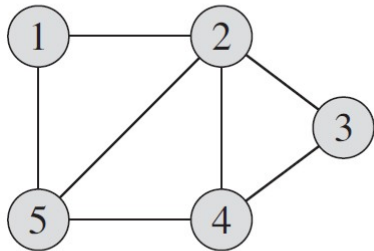
- Matriz de adyacencia: Es una matriz cuadrada en la que el elemento  $i,j$  indica si existe un arco entre los vértices  $i$  y  $j$ . Esta representación es fácil de implementar y rápida para acceder a la información de adyacencia entre dos vértices específicos, pero puede ser ineficiente en términos de espacio si el grafo es disperso (tiene pocos arcos en comparación con el número total de posibles arcos).
- Lista de adyacencia: Es una lista en la que cada vértice tiene una lista de vecinos. Esta representación es más eficiente en términos de espacio que la matriz de adyacencia para grafos dispersos, pero es más lenta para acceder a la información de adyacencia entre dos vértices específicos.

# Representación de los Grafos

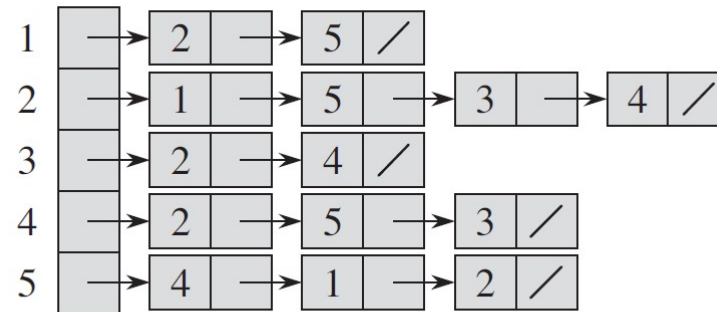


- Representación de Imagen: Los grafos pueden ser representados mediante una imagen, donde cada vértice se representa mediante un punto o un círculo y cada arco se representa mediante una línea que une dos vértices. Esta representación es fácil de visualizar y comprender, pero puede no ser adecuada para grafos muy grandes o complejos.
- Representación abstracta: Es una representación en la que se utilizan estructuras de datos abstractas para representar a los vértices y arcos. Esto puede ser útil en casos en los que los vértices o arcos tienen atributos o propiedades adicionales que deben ser almacenadas y utilizadas en algoritmos.

# Representación de los Grafos



(a)



(b)

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

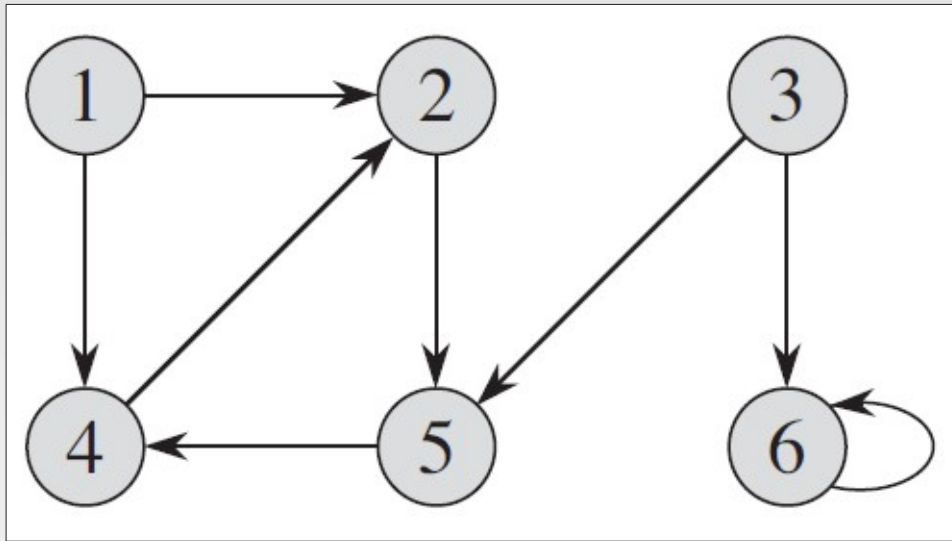
(c)

a) Representación de Imagen: grafo no dirigido de 5 vértices y 7 aristas.

b) Representación en Lista de adyacencia.

c) Representación en Matriz de adyacencia.

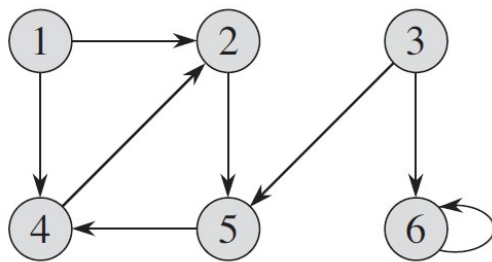
# Ejercicio



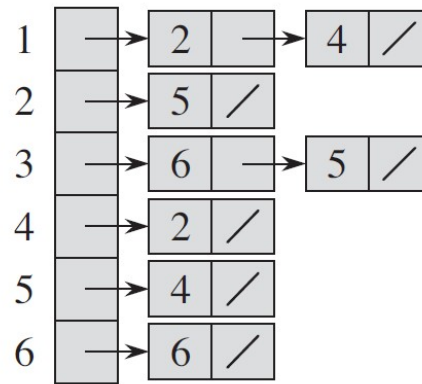
A partir de la representación en imagen del grafo dirigido, realice:

- a) La representación en Lista de adyacencia.
- b) La representación en Matriz de adyacencia.

# Ejercicio (Solución)



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)



# Representación con Matriz Adyacente



```
class GraphMatrixAdj {
private:
    int numVertices;
    vector<vector<int>> adjacencyMatrix;

public:
    GraphMatrixAdj(int numVertices) {
        this->numVertices = numVertices;
        adjacencyMatrix.resize(numVertices, vector<int>(numVertices, 0));
    }
    void addEdge(int i, int j) {
        adjacencyMatrix[i][j] = 1;
        adjacencyMatrix[j][i] = 1;
    }
    void removeEdge(int i, int j) {
        adjacencyMatrix[i][j] = 0;
        adjacencyMatrix[j][i] = 0;
    }
    bool isAdjacent(int i, int j) {
        return adjacencyMatrix[i][j];
    }
    void printGraph() {
        for (int i = 0; i < numVertices; i++) {
            cout << i << ": ";
            for (int j = 0; j < numVertices; j++) {
                cout << adjacencyMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    GraphMatrixAdj grafo(5);
    grafo.addEdge(0, 1);
    grafo.addEdge(0, 4);
    grafo.addEdge(1, 2);
    grafo.addEdge(1, 3);
    grafo.addEdge(1, 4);
    grafo.addEdge(2, 3);
    grafo.printGraph();

    return 0;
}
```

# Representación con Lista Adyacente



```
class GraphListAdjWeigth {
private:
    int numVertices;
    vector<vector<pair<int, int>>> adjList;

public:
    GraphListAdjWeigth(int numVertices) {
        this->numVertices = numVertices;
        adjList.resize(numVertices);
    }

    void addEdge(int src, int dest, int weight) {
        adjList[src].push_back({ dest, weight });
        adjList[dest].push_back({ src, weight });
    }

    void printGraph() {
        for (int i = 0; i < numVertices; i++) {
            cout << i << " --> ";
            for (auto j : adjList[i]) {
                cout << "(" << j.first << ", " << j.second << ") ";
            }
            cout << endl;
        }
    }
};
```

```
int main() {
    GraphListAdjWeigth grafo(5);
    grafo.addEdge(0, 1);
    grafo.addEdge(0, 4);
    grafo.addEdge(1, 2);
    grafo.addEdge(1, 3);
    grafo.addEdge(1, 4);
    grafo.addEdge(2, 3);
    grafo.printGraph();


    return 0;
}
```

# Representación Abstracta

```
class Vertex {  
private:  
    int data;  
    vector<Vertex*> neighbors;  
public:  
    Vertex(int data) {  
        this->data = data;  
    }  
    void addNeighbor(Vertex* v) {  
        neighbors.push_back(v);  
    }  
    int getData() {  
        return data;  
    }  
    vector<Vertex*> getNeighbors() {  
        return neighbors;  
    }  
};
```

```
class Graph {  
private:  
    vector<Vertex*> vertices;  
public:  
    void addVertex(int data) {  
        Vertex* newVertex = new Vertex(data);  
        vertices.push_back(newVertex);  
    }  
  
    Vertex* getVertex(int index) {  
        return vertices[index - 1];  
    }  
  
    void addEdge(Vertex* v1, Vertex* v2) {  
        v1->addNeighbor(v2);  
        v2->addNeighbor(v1);  
    }  
    void printGraph() {  
        for (Vertex* v : vertices) {  
            cout << v->getData() << ": ";  
            for (Vertex* neighbor : v->getNeighbors()) {  
                cout << neighbor->getData() << " ";  
            }  
            cout << endl;  
        }  
    }  
};
```

# Representación Abstracta



```
int main() {  
    Graph graph;  
    graph.addVertex(1);  
    graph.addVertex(2);  
    graph.addVertex(3);  
    graph.addVertex(4);  
    graph.addEdge(graph.getVertex(1), graph.getVertex(2));  
    graph.addEdge(graph.getVertex(1), graph.getVertex(3));  
    graph.addEdge(graph.getVertex(2), graph.getVertex(3));  
    graph.addEdge(graph.getVertex(3), graph.getVertex(4));  
    graph.printGraph();  
  
    return 0;  
}
```

# Tipos de Grafos



- 1) Grafos no dirigidos: los arcos no tienen dirección, es decir, si hay un arco entre dos vértices A y B, también hay un arco entre B y A.
- 2) Grafos dirigidos: los arcos tienen dirección, es decir, si hay un arco de A a B, no necesariamente hay un arco de B a A.
- 3) Grafos ponderados: cada arco tiene un peso asociado, que puede ser un número o un valor cualitativo.
- 4) Grafos bipartitos: un grafo es bipartito si se puede dividir en dos conjuntos de vértices de tal manera que todos los arcos conecten un vértice de un conjunto con un vértice del otro conjunto.
- 5) Grafos completos: un grafo es completo si todos los vértices están conectados entre sí.

# Tipos de Grafos



- 6) Grafos cíclicos: un grafo es cíclico si hay al menos un camino que comienza y termina en el mismo vértice.
- 7) Grafos acíclicos: un grafo es acíclico si no contiene ciclos.
- 8) Grafos conexos: un grafo es conexo si hay un camino entre cualquier par de vértices.
- 9) Grafos no conexos: un grafo no es conexo si no hay un camino entre algunos pares de vértices.
- 10) Grafos regulares: un grafo es regular si todos los vértices tienen el mismo grado (número de arcos incidentes en un vértice).
- 11) Grafos planos: un grafo es plano si se puede dibujar en un plano sin que dos arcos se crucen.

# Tipos de Grafos



- 12) Grafos espaciales: un grafo es espacial si los vértices y arcos tienen posiciones geométricas en un espacio tridimensional.
- 13) Grafos de flujo: un grafo es de flujo si tiene una capacidad asociada a cada arco y un flujo asociado a cada arco que cumple ciertas restricciones.
- 14) Grafos de costo mínimo: un grafo es de costo mínimo si tiene un costo asociado a cada arco y se busca encontrar un camino entre dos vértices con el costo mínimo posible.
- 15) Grafos de camino más corto: un grafo es de camino más corto si tiene un costo asociado a cada arco y se busca encontrar el camino más corto entre dos vértices.

# Tipos de Grafos



- 16) Grafos de redes: un grafo es de red si representa un sistema de conexiones, como las redes de comunicaciones, las redes de transporte o las redes de suministro. Estos grafos suelen tener atributos adicionales como capacidades, costos, flujos, etc.
- 17) Grafos de procesamiento de tareas: es un grafo dirigido acíclico que representa un conjunto de tareas y sus dependencias, donde las tareas son los vértices y las dependencias son los arcos.
- 18) Grafos de decisión: es un grafo dirigido acíclico que representa un conjunto de decisiones y sus consecuencias, donde las decisiones son los vértices y las consecuencias son los arcos.

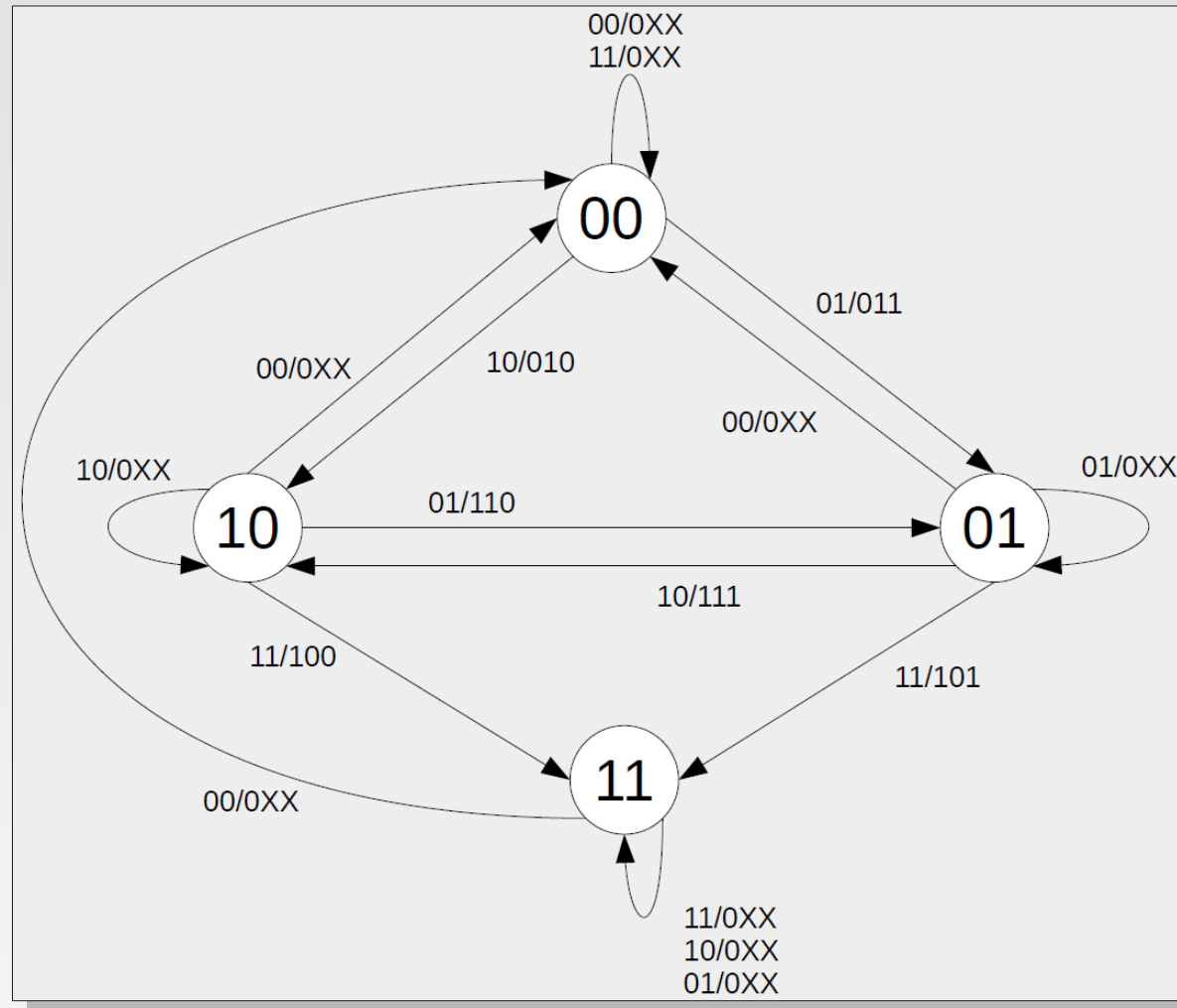


# Tipos de Grafos



- 19) Grafos de coloreado: es un grafo no dirigido en el que se busca asignar colores a los vértices de tal manera que dos vértices adyacentes no tengan el mismo color.
- 20) Grafos de clique: es un grafo no dirigido en el que se busca encontrar un conjunto de vértices en el que todos los vértices estén conectados entre sí.

# Aplicaciones de Grafos



# Algoritmos utilizados en Grafos



- 1) BFS (Breadth-First Search): El algoritmo BFS recorre el grafo en amplitud, es decir, visita todos los vértices de un nivel antes de pasar al siguiente nivel. El algoritmo BFS utiliza una cola para almacenar los vértices a visitar, y se utiliza para encontrar el camino más corto entre dos vértices.
- 2) DFS (Depth-First Search): El algoritmo DFS recorre el grafo en profundidad, es decir, visita un vértice y luego sigue recorriendo sus vértices adyacentes. El algoritmo DFS utiliza una pila para almacenar los vértices a visitar, y se utiliza para encontrar todos los caminos entre dos vértices.

# Pseudocódigo BFS iterativo



```
BFS(G, s):
    // G es el grafo, s es el vértice origen
    // Inicializar estado de vértices
    for each vertex v in G:
        visited[v] = false
    // Crear cola y agregar vértice origen
    queue Q
    Q.add(s)
    visited[s] = true
    // Mientras la cola no esté vacía
    while Q is not empty:
        // Extraer vértice de la cola
        u = Q.pop()
        // Revisar cada vecino
        for each neighbor v of u:
            if visited[v] == false:
                // Marcar como visitado y agregar
                visited[v] = true
                Q.add(v)
    // No hay valor de retorno, se utiliza la
    matriz "visited" para verificar si un vértice fue
    visitado o no
```

En resumen, el algoritmo inicializa el estado de los vértices, crea una cola y agrega el vértice origen. Luego, mientras la cola no esté vacía, se extrae un vértice de la cola, se marca como visitado y se agrega a la cola todos sus vecinos no visitados.

El algoritmo termina cuando la cola esta vacía, y no tiene retorno, pero se utiliza la matriz visited para verificar si un vértice ha sido visitado o no.

# Pseudocódigo BFS recursivo



```
BFS_recursive(G, s):
    // G es el grafo, s es el vértice
    origen
    // Inicializar estado de vértices
    for each vertex v in G:
        visited[v] = false
    visited[s] = true
    // Llamar función recursiva
    BFS_visit(s)
```

```
BFS_visit(u):
    // Revisar cada vecino
    for each neighbor v of u:
        if visited[v] == false:
            // Marcar como visitado y
            llamar a la función recursiva
            visited[v] = true
            BFS_visit(v)
```

En resumen, el algoritmo se divide en dos funciones: BFS\_recursive y BFS\_visit. La función BFS\_recursive inicializa el estado de los vértices, marca el vértice origen como visitado y llama a la función BFS\_visit con el vértice origen como parámetro. La función BFS\_visit, a su vez, recorre todos los vecinos no visitados de un vértice, los marca como visitados y llama a sí misma con cada uno de ellos como parámetro.

# Dato curioso del BFS



- La implementación del algoritmo BFS en iterativo (con una cola) suele ser considerada como más eficiente que la implementación recursiva.
- La implementación iterativa del algoritmo BFS utiliza una cola para almacenar los vértices a visitar, lo que significa que el algoritmo siempre está trabajando con el vértice más próximo en la cola. Esto ayuda a reducir la cantidad de memoria utilizada, ya que solo se necesita almacenar los vértices en la cola y no todas las llamadas recursivas.
- Por otro lado, la implementación recursiva del algoritmo BFS utiliza una pila de llamadas recursivas para almacenar los vértices a visitar, lo que significa que el algoritmo puede llegar a utilizar más memoria que la implementación iterativa si el grafo es muy grande. Además, puede llegar a tener problemas de stack overflow en caso de un grafo muy grande o profundo.
- En conclusión, la implementación iterativa del algoritmo BFS suele ser considerada como la mejor opción en términos de eficiencia y escalabilidad. Sin embargo, la implementación recursiva puede ser útil en casos específicos o para fines educativos.

# Algoritmos utilizados en Grafos



- 3) Dijkstra: Es un algoritmo de búsqueda de camino más corto en grafos ponderados, donde se asigna un peso a cada arco. El algoritmo utiliza una cola de prioridades para determinar el camino más corto desde un vértice de origen hasta todos los demás vértices.
- 4) Bellman-Ford: Es similar al algoritmo de Dijkstra, pero se utiliza para grafos ponderados con arcos negativos, es decir, arcos con pesos negativos. El algoritmo se basa en la relajación de los arcos, es decir, en la actualización de los pesos de los arcos para encontrar el camino más corto.

# Pseudocódigo Dijkstra



```
Dijkstra(G, s):
// G es el grafo, s es el vértice origen
// Inicializar distancias y estado de vértices
for each vertex v in G:
    dist[v] = infinito
    prev[v] = null
    visited[v] = false
dist[s] = 0
// Crear cola de prioridades y agregar vértice origen
priority_queue Q
Q.add(s, 0)
// Mientras la cola no esté vacía
while Q is not empty:
    // Extraer vértice con distancia mínima
    u = Q.extract_min()
    visited[u] = true
    // Revisar cada arco (u, v)
    for each neighbor v of u:
        if visited[v] == false:
            // Actualizar distancia si es mejor
            if dist[u] + weight(u, v) < dist[v]:
                dist[v] = dist[u] + weight(u, v)
                prev[v] = u
                Q.add(v, dist[v])
// Devolver distancias y predecesores
return dist, prev
```

En resumen, el algoritmo inicializa las distancias y el estado de los vértices, crea una cola de prioridades y agrega el vértice origen con distancia 0. Luego, mientras la cola no esté vacía, se extrae el vértice con distancia mínima, se marca como visitado y se revisan todos sus vecinos. Si un vecino no ha sido visitado y la distancia a través del vértice actual es menor que la distancia actual, se actualiza la distancia y se agrega el vecino a la cola de prioridades.

Finalmente, el algoritmo devuelve las distancias y los predecesores de cada vértice. Con esta información, se pueden reconstruir los caminos más cortos desde el vértice origen a cualquier otro vértice.



# Algoritmos utilizados en Grafos



- 6)Prim: Es un algoritmo para encontrar un árbol recubridor mínimo (MST, por sus siglas en inglés) en un grafo ponderado no dirigido. El algoritmo comienza con un vértice cualquiera y agrega sucesivamente vértices al MST eligiendo el arco de menor peso que conecta un vértice fuera del MST con uno dentro del mismo.
- 7)Kruskal: Es otro algoritmo para encontrar un MST en un grafo ponderado no dirigido. El algoritmo comienza ordenando todos los arcos por peso y luego va agregando arcos al MST siempre y cuando no formen ciclo. El algoritmo utiliza una estructura de conjuntos disjuntos para determinar si un arco forma un ciclo o no.

# Algoritmos utilizados en Grafos



- 8) Floyd-Warshall: Es un algoritmo para encontrar el camino más corto entre todos los pares de vértices en un grafo ponderado con arcos negativos. El algoritmo utiliza una matriz de distancias para almacenar el camino más corto entre cada par de vértices, y se basa en la relajación de los arcos para determinar los caminos más cortos.
- 9) Algoritmo de Johnson: Es una variante del algoritmo de Floyd-Warshall que se utiliza para encontrar caminos más cortos en grafos ponderados con arcos negativos. Es un algoritmo combinado que utiliza el algoritmo Bellman-Ford y Dijkstra.

# ¿Qué es un algoritmo de relajación de arcos?

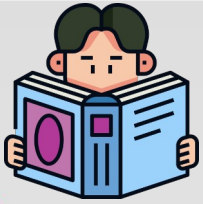
- Es un algoritmo utilizado para encontrar el camino más corto entre un vértice de origen y todos los demás vértices en un grafo ponderado. Este algoritmo se basa en la idea de ir actualizando los pesos de los vértices a medida que se recorren los arcos del grafo.
- La relajación de un arco es el proceso de comparar el peso actual de un vértice destino con el peso del vértice origen más el peso del arco que conecta ambos vértices. Si el peso del vértice destino es mayor que la suma del peso del vértice origen y del arco, se actualiza el peso del vértice destino con la suma mencionada. Este proceso se repite para cada arco del grafo varias veces hasta que no se realicen más actualizaciones.



# Ejercicios



- 1)Escribir un programa para encontrar el camino más corto entre dos nodos en un grafo utilizando el algoritmo de Dijkstra.
- 2)Escribir un programa para encontrar el camino más largo entre dos nodos en un grafo utilizando el algoritmo de Bellman-Ford.
- 3)Escribir un programa para determinar si un grafo dado es conexo utilizando el algoritmo de búsqueda en profundidad (DFS).
- 4)Escribir un programa para encontrar el recorrido en profundidad de un grafo utilizando recursión.
- 5)Escribir un programa para encontrar el recorrido en anchura de un grafo utilizando una cola.
- 6)Escribir un programa para determinar si un grafo es bipartito utilizando el algoritmo de búsqueda en profundidad.



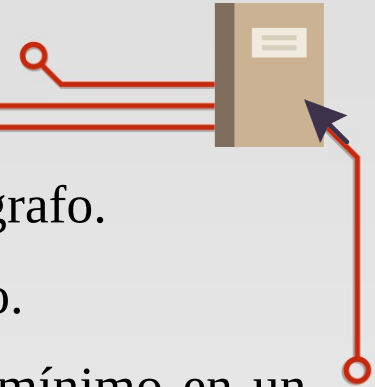
# Ejercicios



- 7) Escribir un programa para encontrar el árbol de expansión mínimo de un grafo utilizando el algoritmo de Prim.
- 8) Escribir un programa para encontrar el árbol de expansión mínimo de un grafo utilizando el algoritmo de Kruskal.
- 9) Escribir un programa para encontrar el número de caminos más cortos entre dos nodos en un grafo utilizando el algoritmo de Floyd-Warshall.
- 10) Escribir un programa para encontrar el camino de costo mínimo entre dos nodos en un grafo ponderado utilizando el algoritmo de Dijkstra.
- 11) Escribir un programa para encontrar el camino de costo mínimo entre dos nodos en un grafo ponderado utilizando el algoritmo de Bellman-Ford.
- 12) Escribir un programa que detecte ciclos en un grafo no dirigido.
- 13) Escribir un programa que detecte si un grafo es conexo o no.



# Ejercicios



- 14) Escribir un programa que encuentre un camino hamiltoniano en un grafo.
- 15) Escribir un programa que encuentre un camino euleriano en un grafo.
- 16) Escribir un programa que encuentre un recubrimiento de vértices mínimo en un grafo.
- 17) Escribir un programa que encuentre un recubrimiento de arista mínimo en un grafo.
- 18) Escribir un programa que encuentre la componente fuertemente conectada más grande en un grafo dirigido.
- 19) Escribir un programa que encuentre el camino más corto entre dos nodos en un grafo con pesos negativos utilizando el algoritmo de Bellman-Ford.
- 20) Escribir un programa que encuentre la longitud del camino más corto desde un nodo a todos los demás nodos en un grafo utilizando BFS.

# Bibliografía



- Cairó, O & Guardati, S. Estructuras de Datos. Tercera Edición. McGraw Hill.
- Aho, A, Hopcroft, J & Ullman, J. 1988. Estructuras de datos y algoritmos. Addison-Wesley Iberoamericana. ISBN 968-6048-19-7
- Cormen, T, et al. Introduction to Algorithms. Tercera Edición. The MIT Press Cambridge, Massachusetts.