Teoría de grafos y algoritmos

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

May 13, 2025

Descripción de la unidad

En este capítulo se consideran los conceptos fundamentales de la teoría de grafos, la representación de grafos por medio de estructuras de datos y algunos de los algoritmos más importantes para resolver problemas de grafos.

Contents

1 Fundamentos de teoría de grafos (Graph Theory)

La teoría de grafos es fundamental en ciencias de la computación, ya que proporciona una poderosa herramienta para modelar y resolver problemas que involucran relaciones entre objetos. En este capítulo, aprenderemos los conceptos básicos de grafos, sus diferentes tipos, representaciones y algoritmos fundamentales que nos permitirán resolver problemas prácticos de manera eficiente.

1.1 Introducción a los Grafos

Un grafo es una estructura matemática utilizada para representar relaciones entre objetos. Formalmente, un grafo G se define como un par $G = (V, E)$, donde V es un conjunto de **vértices** (o nodos) y E es un conjunto de **aristas** (o enlaces) que conectan pares de vértices.

1.2 Modelado de Situaciones Prácticas con Grafos

Los grafos son herramientas poderosas para modelar una amplia variedad de situaciones prácticas. A continuación, se presentan algunos ejemplos comunes:

- **Redes sociales**:
  - Vértices: Personas
  - Aristas: Relaciones 'ser amigo', 'ser seguidor'
  - Aplicación: Identificar comunidades (intereses), sistemas de recomendación (publicidad)
- **Redes de transporte**:
  - Vértices: Ciudades, estaciones, aeropuertos, etc.
  - Aristas: Carreteras, rutas aéreas o conexiones de tren
  - Aplicación: Encontrar la ruta más corta, optimizar rutas de entrega
- **Redes de computadoras**:
  - Vértices: Dispositivos (routers, switches, computadoras)
  - Aristas: Conexiones de red (físicas o inalámbricas)
  - Aplicación: Optimizar el enrutamiento de paquetes, ruta más corta
- **Circuitos electrónicos**:
  - Vértices: Componentes (resistencias, capacitores, transistores)
  - Aristas: Conexiones eléctricas
  - Aplicación: Análisis de circuitos, diseño de circuitos impresos
- **Planificación de proyectos**:
  - Vértices: Tareas
  - Aristas: Dependencias entre tareas
  - Aplicación: Encontrar la ruta crítica, optimizar el tiempo de ejecución
- **Máquinas de estados finitos**:

- Vértices: Estados
- Aristas: Transiciones entre estados (eventos)
- Aplicación: Modelado de protocolos de comunicación, análisis de léxico en compiladores, diseño de sistemas de control, reconocimiento de patrones y validación de entradas.

1.3 El Grafo como Abstracción General

La belleza de los grafos radica en su capacidad para abstraer la complejidad de un sistema y enfocarse en las relaciones esenciales entre sus componentes. Al ignorar los detalles irrelevantes, los grafos permiten analizar y resolver problemas de manera eficiente.

1.4 Tipos de Grafos

1.4.1 Grafos No Dirigidos

En un grafo no dirigido, las aristas no tienen dirección. Es decir, si existe una arista entre los vértices u y v, se puede recorrer en ambas direcciones. Matemáticamente, una arista en un grafo no dirigido es un par no ordenado $\{u, v\}$.

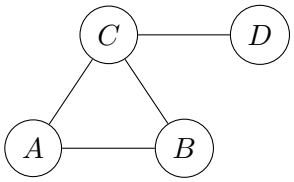


Figure 1: Grafo no dirigido

1.4.2 Grafos Dirigidos

En un grafo dirigido, las aristas tienen dirección. Es decir, si existe una arista desde el vértice u hasta el vértice v, solo se puede recorrer en esa dirección. Matemáticamente, una arista en un grafo dirigido es un par ordenado (u, v).

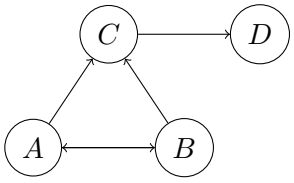


Figure 2: Grafo dirigido

1.5 Representación por Conjuntos

1.5.1 Grafo No Dirigido

Un grafo no dirigido G se define como $G = (V, E)$, donde:

- V es el conjunto de vértices.
- E es el conjunto de aristas, donde cada arista es un par no ordenado de vértices $\{u, v\}$, con $u, v \in V$.

1.5.2 Grafo Dirigido

Un grafo dirigido G se define como $G = (V, E)$, donde:

- V es el conjunto de vértices.
- E es el conjunto de aristas, donde cada arista es un par ordenado de vértices (u, v), con $u, v \in V$.

1.6 Terminología

- **Camino**: Una secuencia de vértices conectados por aristas.
- **Camino Simple**: Un camino donde no se repiten vértices.

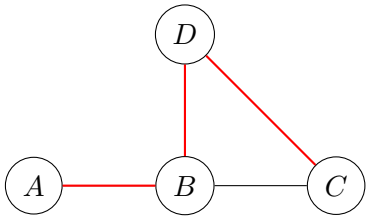


Figure 3: Ejemplo de camino simple: A-B-D-C

- **Ciclo**: Un camino que comienza y termina en el mismo vértice.
- **Ciclo Simple**: Un ciclo donde no se repiten vértices intermedios.

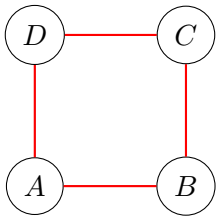


Figure 4: Ejemplo de ciclo simple: A-B-C-D-A

- **Grafo Conexo**: Un grafo donde existe un camino entre todo par de vértices. (No se exige seguir el sentido de las aristas en grafos dirigidos)

Figure 5: Ejemplo de grafo conexo (izquierda) y no conexo (derecha)

- **Grafo Fuertemente Conexo**: Un grafo dirigido donde existe un camino entre todo par de vértices respetando el sentido de las aristas.

Figure 6: Ejemplo de grafo fuertemente conexo (izquierda) y no fuertemente conexo (derecha)

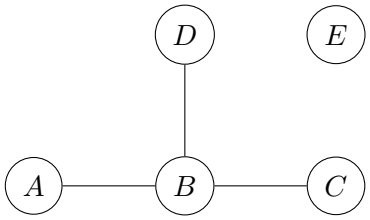


Figure 7: Ejemplo de grafo acíclico

- **Grafo Acíclico**: Un grafo que no contiene ciclos.
- **Árbol**: Un grafo conexo acíclico.

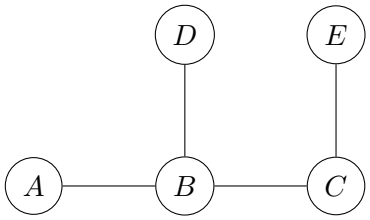


Figure 8: Ejemplo de árbol

- **Grafo Acíclico Dirigido (Directed Acyclic Graph (DAG))**: Un grafo dirigido que no contiene ciclos.

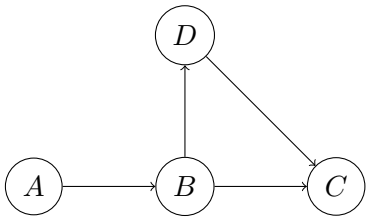


Figure 9: Ejemplo de DAG

- **Grado de un Nodo**: El número de aristas incidentes a un nodo. En grafos dirigidos, se distingue entre grado de entrada (aristas que llegan al nodo) y grado de salida (aristas que salen del nodo).
- **Clique**: Un grafo (o subgrafo de un grafo mayor) en el que todo par de vértices está conectado por una arista.

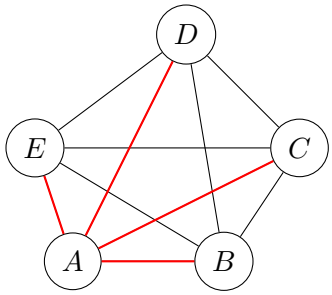


Figure 10: Ejemplo de clique de tamaño 5

Theorem 1 (Cota superior del número de aristas) *El máximo número de aristas de un grafo es $O(V^2)$, siendo el clique es máximo posible.*

- ***Grafo no dirigido:*** *En todo grafo no dirigido con V vértices, el número de aristas E satisface la desigualdad:*
$$E \leq \frac{(V-1)V}{2}$$
- ***Grafo dirigido:*** *En todo grafo dirigido con V vértices, el número de aristas E satisface la desigualdad:*
$$E \leq (V-1)V$$

- **Densidad de un Grafo**: La proporción de aristas presentes en un grafo en relación con el número máximo posible de aristas. Si el número de aristas en el grafo es pequeño con respecto al máximo posible, se dice que el grafo es *no denso*. Un criterio, que es útil para el análisis, es tomar como no densos los grafos con $E \leq cV$ (el número de aristas es lineal con respecto al número de vértices).
- **Grafo Bipartito**: Un grafo cuyos vértices pueden dividirse en dos conjuntos disjuntos de tal manera que cada arista conecta un vértice de un conjunto con un vértice del otro conjunto. No se permiten aristas entre vértices del mismo conjunto.

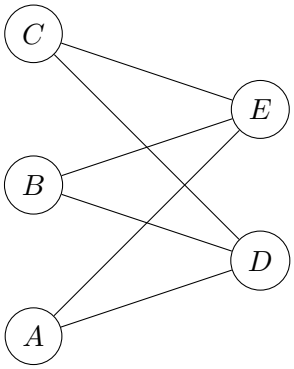


Figure 11: Ejemplo de grafo bipartito

Aplicaciones de grafos bipartitos: Se utilizan frecuentemente para modelar relaciones de asignación de recursos. Por ejemplo la asignación de estudiantes a cursos, o de empleados a tareas.

1.7 ADTs de Grafos

Para definir el ADT de los grafos dirigidos y no dirigidos, vamos a asumir sin pérdida de generalidad que los vértices del grafo se numeran $0 \dots V-1$.

1.7.1 ADT Grafo No Dirigido

Método	Descripción
`int V()`	Retorna número de vértices
`int E()`	Retorna número de aristas
`void addEdge(int u, int v)`	Agrega la arista $\{u, v\}$
`Iterable<Integer> adj(int v)`	Adyacentes de v
`int degree(int v)`	Retorna el grado de v
`String toString()`	Retorna la representación textual del grafo

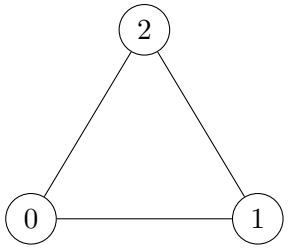
1.7.2 ADT Grafo Dirigido

Método	Descripción
`int V()`	Retorna número de vértices
`int E()`	Retorna número de aristas
`void addEdge(int u, int v)`	Agrega la arista (u, v)
`Iterable<Integer> adj(int v)`	Adyacentes de v
`int indegree(int v)`	Retorna el grado entrante de v
`int outdegree(int v)`	Retorna el grado saliente de v
`String toString()`	Retorna la representación textual del grafo
`DiGraph reverse()`	Retorna el grafo reverso

1.8 Representación de Grafos

1.8.1 Matrices de Adyacencia

Una matriz de adyacencia es una matriz cuadrada de tamaño $|V| \times |V|$, donde $|V|$ es el número de vértices en el grafo. El elemento (i, j) de la matriz es 1 si existe una arista entre el vértice i y el vértice j, y 0 en caso contrario.

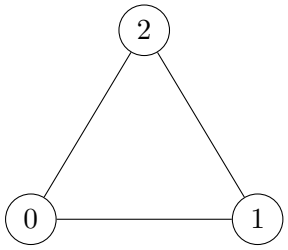


	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

Figure 12: Ejemplo de grafo y su matriz de adyacencia

1.8.2 Listas de Adyacencia

Una lista de adyacencia es una lista donde cada elemento representa un vértice en el grafo. Cada elemento de la lista contiene una lista de los vértices adyacentes a ese vértice.



Vértice	Adyacentes
0	1, 2
1	0, 2
2	0, 1

Figure 13: Ejemplo de grafo y su lista de adyacencia

1.9 Bibliotecas para representación y manipulación de grafos

Algunos ejemplos de herramientas disponibles para distintos lenguajes:

- **Python**: NetworkX, PyGraphviz, GraphViz.
- **Java**: JUNG, JGraphT, Guava, Apache commons graph.
- **C#**: QuickGraph.
- **C++**: Boost Graph Library.

1.10 Algoritmos Fundamentales de Grafos

Los algoritmos de grafos son herramientas esenciales para resolver problemas prácticos. A continuación, se presentan algunos de los algoritmos más importantes:

- **Recorrido en Profundidad (DFS)**:
  - Propósito: Explorar todos los vértices de un grafo siguiendo un camino hasta el final antes de retroceder.
  - Aplicaciones: Detección de ciclos, ordenamiento topológico, componentes conexas.
  - Complejidad: $O(|V| + |E|)$, donde $|V|$ es el número de vértices y $|E|$ es el número de aristas.
- **Recorrido en Anchura (BFS)**:
  - Propósito: Explorar todos los vértices nivel por nivel, comenzando desde un vértice inicial.
  - Aplicaciones: Encontrar el camino más corto en grafos no ponderados, verificar si un grafo es bipartito.
  - Complejidad: $O(|V| + |E|)$.
- **Ordenamiento Topológico**:
  - Propósito: Ordenar los vértices de un grafo dirigido acíclico (DAG) de tal manera que para cada arista dirigida (u, v), el vértice u aparece antes que el vértice v en el ordenamiento.
  - Aplicaciones: Planificación de tareas, análisis de dependencias.
  - Complejidad: $O(|V| + |E|)$.

- **Algoritmo de Dijkstra**:
  - Propósito: Encontrar el camino más corto desde un vértice origen a todos los demás vértices en un grafo con pesos no negativos.
  - Aplicaciones: Enrutamiento en redes, sistemas de navegación GPS.
  - Complejidad: $O((|V| + |E|) \log |V|)$ usando una cola de prioridad.
- **Algoritmo de Bellman-Ford**:
  - Propósito: Encontrar el camino más corto desde un vértice origen a todos los demás vértices, permitiendo pesos negativos.
  - Aplicaciones: Detección de ciclos negativos, enrutamiento en redes con costos negativos.
  - Complejidad: $O(|V||E|)$.
- **Algoritmo de Floyd-Warshall**:
  - Propósito: Encontrar los caminos más cortos entre todos los pares de vértices.
  - Aplicaciones: Análisis de redes, cálculo de distancias entre todas las ciudades.
  - Complejidad: $O(|V|^3)$.
- **Algoritmo de Kruskal**:
  - Propósito: Encontrar el árbol de expansión mínima de un grafo.
  - Aplicaciones: Diseño de redes de comunicación, planificación de rutas de cableado.
  - Complejidad: $O(|E| \log |E|)$.
- **Algoritmo de Prim**:
  - Propósito: Encontrar el árbol de expansión mínima de un grafo.
  - Aplicaciones: Similar a Kruskal, pero puede ser más eficiente en grafos densos.
  - Complejidad: $O((|V| + |E|) \log |V|)$ usando una cola de prioridad.
- **Algoritmo de Ford-Fulkerson**:
  - Propósito: Encontrar el flujo máximo en una red de flujo.
  - Aplicaciones: Optimización de redes de distribución, asignación de recursos.
  - Complejidad: $O(|E|f^*)$, donde f^* es el valor del flujo máximo.

2 Algoritmos de recorrido de grafos

Los algoritmos de recorrido de grafos son fundamentales para explorar y analizar la estructura de un grafo. Permiten visitar cada vértice del grafo de manera sistemática, lo que es esencial para resolver una variedad de problemas, como la búsqueda de caminos, la detección de ciclos y la identificación de componentes conexas.

2.1 Aplicaciones de los Recorridos de Grafos

Los recorridos de grafos tienen numerosas aplicaciones en informática y otras disciplinas, incluyendo:

- **Búsqueda de caminos**:
  - Encontrar la ruta más corta entre dos puntos en un mapa.
  - Planificar rutas de entrega optimizando tiempo y distancia.
  - Navegación GPS y sistemas de recomendación de rutas.
- **Detección de ciclos**:
  - Verificar la integridad de datos en bases de datos.
  - Detectar dependencias circulares en código fuente.
  - Validar la estructura de redes de comunicación.
- **Análisis de redes sociales**:
  - Identificar comunidades y grupos de usuarios.
  - Recomendar conexiones basadas en intereses comunes.
  - Analizar patrones de interacción y difusión de información.
- **Planificación de tareas**:
  - Determinar el orden óptimo de ejecución de tareas.
  - Identificar dependencias críticas en proyectos.
  - Optimizar la asignación de recursos.
- **Resolución de laberintos**:
  - Encontrar la salida más corta en un laberinto.
  - Generar soluciones para puzzles y juegos.
  - Diseñar algoritmos de navegación para robots.

2.2 Recorrido en profundidad (DFS)

El recorrido en profundidad (Depth-first Search (DFS)) es un algoritmo para recorrer un grafo que explora tan lejos como sea posible a lo largo de cada rama antes de retroceder. Es como explorar un laberinto siguiendo un camino hasta llegar a un callejón sin salida, y luego retroceder para explorar caminos alternativos.

2.2.1 Características de DFS

- Explora un camino completo antes de retroceder.
- Utiliza una estructura de pila (stack) implícita o explícita.
- Es útil para detectar ciclos y componentes conexas.
- Puede implementarse de forma recursiva o iterativa.

2.2.2 Ejemplo Visual de DFS

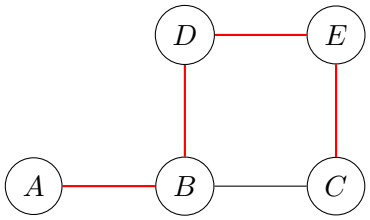


Figure 14: Ejemplo de recorrido DFS: A-B-D-E-C

2.2.3 Pseudo-código para DFS

Algorithm 1 DFS(grafo, vértice)

```
1: procedure DFS(grafo, vértice)
2:     Marcar vértice como visitado
3:     for cada vecino del vértice en el grafo do
4:         if vecino no está visitado then
5:             DFS(grafo, vecino)
6:         end if
7:     end for
8: end procedure
```

2.2.4 Complejidad de DFS

La complejidad temporal de DFS es $O(V + E)$, donde V es el número de vértices y E es el número de aristas en el grafo. Esto se debe a que DFS visita cada vértice y cada arista una vez. La complejidad espacial de DFS es $O(V)$ en el peor de los casos, debido a la profundidad de la pila de recursión.

2.3 Recorrido en anchura (BFS)

El recorrido en anchura (Breadth-first Search (BFS)) es un algoritmo para recorrer un grafo que explora todos los vecinos del vértice actual antes de pasar a los vecinos de los vecinos. Es como explorar un laberinto nivel por nivel, asegurándose de visitar todos los caminos posibles a una distancia dada antes de avanzar.

2.3.1 Características de BFS

- Explora todos los vértices a una distancia k antes de pasar a distancia k+1.
- Utiliza una estructura de cola (queue).
- Encuentra el camino más corto en grafos no ponderados.
- Es útil para verificar si un grafo es bipartito.

2.3.2 Ejemplo de BFS

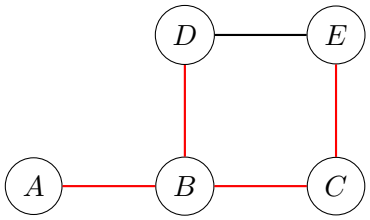


Figure 15: Ejemplo de recorrido BFS: A, B, C-D, E

2.3.3 Pseudo-código para BFS

Algorithm 2 BFS(grafo, vértice)

```
1:  procedure BFS(grafo, vértice)
2:      Crear una cola Q
3:      Encolar vértice en Q
4:      Marcar vértice como visitado
5:      while Q no está vacía do
6:          vértice = desencolar de Q
7:          for cada vecino del vértice en el grafo do
8:              if vecino no está visitado then
9:                  Encolar vecino en Q
10:                 Marcar vecino como visitado
11:             end if
12:         end for
13:     end while
14: end procedure
```

2.3.4 Complejidad de BFS

La complejidad temporal de BFS es $O(V + E)$, donde V es el número de vértices y E es el número de aristas en el grafo. Esto se debe a que BFS visita cada vértice y cada arista una vez. La complejidad espacial de BFS es $O(V)$, ya que en el peor de los casos, la cola puede contener todos los vértices del grafo.

2.4 Ordenamiento topológico (Topological Sort)

El ordenamiento topológico es un ordenamiento lineal de los vértices en un grafo dirigido acíclico (DAG) tal que para cada arista dirigida (u, v), el vértice u viene antes del vértice v en el ordenamiento. En otras palabras, el ordenamiento topológico respeta las dependencias entre los vértices.

2.4.1 Ejemplo de DAG con Dependencias

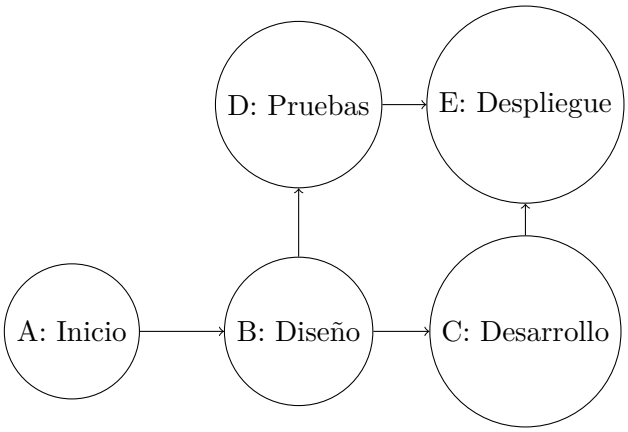


Figure 16: Ejemplo de DAG representando dependencias de tareas en un proyecto

2.4.2 Ordenamiento topológico de Tareas

Para el DAG mostrado en la Figura 16, un ordenamiento topológico válido sería:

1. A: Inicio
2. B: Diseño
3. C: Desarrollo
4. D: Pruebas
5. E: Despliegue

Este ordenamiento respeta todas las dependencias:

- El diseño (B) debe completarse antes del desarrollo (C) y las pruebas (D)
- Tanto el desarrollo (C) como las pruebas (D) deben completarse antes del despliegue (E)
- El inicio (A) debe ser la primera tarea

2.4.3 Aplicaciones del Ordenamiento Topológico

El ordenamiento topológico tiene varias aplicaciones prácticas, incluyendo:

- **Planificación de tareas**: Determinar el orden en que se deben realizar las tareas en un proyecto, teniendo en cuenta las dependencias entre ellas.

- **Análisis de dependencias**: Identificar las dependencias entre módulos de software o componentes de hardware.
- **Compilación de código**: Determinar el orden en que se deben compilar los archivos de código fuente, teniendo en cuenta las dependencias entre ellos.
- **Resolución de dependencias**: Resolver las dependencias entre paquetes de software o bibliotecas.

2.4.4 Pseudo-código para el Ordenamiento Topológico

Un algoritmo para ordenamiento topológico se basa en una búsqueda en profundidad (DFS) para calcular el post-orden de los vértices. El ordenamiento topológico se obtiene listando los vértices en el orden inverso del post-orden.

Algorithm 3 OrdenamientoTopológicoDFS(grafo)

```
1: procedure OrdenamientoTopológicoDFS(grafo)
2:     Crear una lista L para almacenar el ordenamiento topológico
3:     Crear un conjunto de vértices visitados
4:     for cada vértice en el grafo do
5:         if vértice no está visitado then
6:             DFS(grafo, vértice, L, visitados)
7:         end if
8:     end for
9:     Devolver L (en orden inverso)
10: end procedure
11: procedure DFS(grafo, vértice, L, visitados)
12:     Marcar vértice como visitado
13:     for cada vecino del vértice en el grafo do
14:         if vecino no está visitado then
15:             DFS(grafo, vecino, L, visitados)
16:         end if
17:     end for
18:     Agregar vértice al inicio de L
19: end procedure
```

Este algoritmo primero realiza una búsqueda en profundidad en el grafo. Cuando la búsqueda en profundidad completa un vértice (es decir, después de visitar todos sus vecinos), el vértice se agrega al inicio de una lista. La lista resultante, cuando se invierte, da el ordenamiento topológico.

Ejemplo : Analizar la ejecución de OrdenamientoTopológicoDFS en el grafo de la figura 17

Otro algoritmo común para el ordenamiento topológico es el algoritmo de Kahn, que utiliza una cola para mantener un registro de los vértices con grado de entrada cero.

2.4.5 Complejidad del Ordenamiento Topológico

La complejidad temporal del algoritmo de Kahn es $O(V + E)$, donde V es el número de vértices y E es el número de aristas en el grafo. Esto se debe a que el algoritmo visita cada vértice y cada

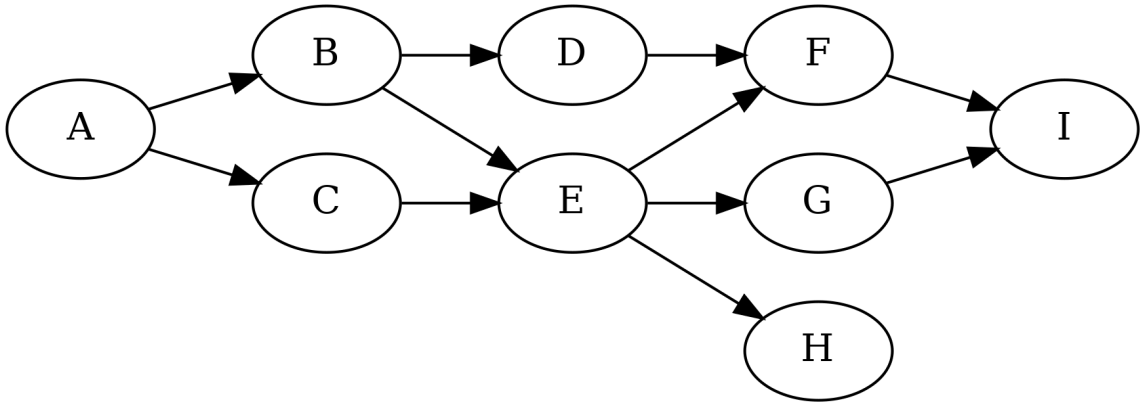


Figure 17: Ejemplo para análisis del ordenamiento topológico

Algorithm 4 OrdenamientoTopológicoKahn(grafo)

```
1:  procedure OrdenamientoTopológicoKahn(grafo)
2:      Crear una cola Q
3:      Encolar todos los vértices con grado de entrada 0 en Q
4:      Crear una lista L para almacenar el ordenamiento topológico
5:      while Q no está vacía do
6:          u = desencolar de Q
7:          Agregar u a L
8:          for cada vecino v de u en el grafo do
9:              Reducir el grado de entrada de v en 1
10:             if el grado de entrada de v es 0 then
11:                 Encolar v en Q
12:             end if
13:         end for
14:     end while
15:     if el grafo tiene aristas then
16:         Imprimir "Error: el grafo tiene un ciclo"
17:     else
18:         Devolver L
19:     end if
20: end procedure
```

arista una vez. La complejidad espacial del algoritmo es $O(V)$, ya que en el peor de los casos, la cola puede contener todos los vértices del grafo.

3 Caminos más cortos (Shortest paths)

3.1 Grafos con Peso o Costo

En un *grafo con peso* (también llamado grafo ponderado), cada arista tiene asociado un valor numérico, que representa su *peso* o *costo*. Este peso puede representar diferentes magnitudes dependiendo de la aplicación, como por ejemplo:

- Distancia física entre dos puntos (en kilómetros, metros, etc.).
- Costo monetario de viajar entre dos ciudades.
- Tiempo necesario para ir de un lugar a otro.
- Ancho de banda disponible en una conexión de red.

Formalmente, un grafo con peso se define como $G = (V, E, w)$, donde V es el conjunto de vértices, E es el conjunto de aristas, y $w : E \to \mathbb{R}$ es una función que asigna un peso a cada arista.

3.2 El Problema del Camino Más Corto

El *problema del camino más corto* consiste en encontrar el camino entre dos vértices de un grafo, tal que la suma de los pesos de las aristas que lo componen sea mínima. Este problema tiene muchas aplicaciones prácticas, entre las que se destacan:

- **Navegación GPS:** Encontrar la ruta más rápida entre dos ubicaciones en un mapa.
- **Planificación de rutas:** Determinar la secuencia óptima de entregas para un camión de reparto.
- **Redes de comunicación:** Encontrar la ruta con menor latencia para enviar datos entre dos nodos.
- **Logística:** Optimizar el flujo de materiales en una cadena de suministro.

3.3 Algoritmo de Dijkstra

El *algoritmo de Dijkstra* es un algoritmo eficiente para resolver el problema del camino más corto en grafos con pesos no negativos en las aristas. Es decir, no funciona correctamente si alguna arista tiene un peso negativo.

La idea principal del algoritmo es mantener un conjunto de vértices ya visitados, y para cada vértice, la distancia más corta conocida desde el vértice de origen. Inicialmente, la distancia al vértice de origen es 0, y la distancia a todos los demás vértices es infinito.

El algoritmo itera seleccionando el vértice no visitado con la menor distancia conocida, y actualiza las distancias a sus vecinos. Este proceso se repite hasta que todos los vértices hayan sido visitados, o hasta que se haya encontrado el camino más corto al vértice destino.

3.3.1 Pseudo-código del Algoritmo de Dijkstra

Algorithm 5 Dijkstra(grafo, origen)

```
1:  procedure DIJKSTRA(grafo, origen)
2:      Inicialización
3:      distancia[origen] ← 0
4:      for cada vértice v en grafo do
5:          if v ≠ origen then
6:              distancia[v] ← infinito
7:          end if
8:          visitado[v] ← falso
9:      end for
10:     Iteración principal
11:     while existan vértices no visitados do
12:         u ← vértice no visitado con la menor distancia conocida
13:         visitado[u] ← verdadero
14:         for cada vecino v de u do
15:             if distancia[u] + peso(u, v) < distancia[v] then
16:                 distancia[v] ← distancia[u] + peso(u, v)
17:                 predecesor[v] ← u
18:             end if
19:         end for
20:     end while
21:     return distancia, predecesor
22: end procedure
```

3.3.2 Ejemplo de Ejecución del Algoritmo de Dijkstra

Considerara el grafo de la figura 18. Analizar el proceso de ejecución de Dijsktra y obtener el camino mas corto de A hacia los demás vértices.

3.4 Eficiencia del algoritmo de Dijkstra

La eficiencia del algoritmo de Dijkstra depende de la implementación utilizada para la cola de prioridad (priority queue). La cola de prioridad se utiliza para seleccionar el vértice no visitado con la menor distancia conocida.

- **Arreglo (Array):** La implementación más simple utiliza un arreglo para almacenar las distancias a todos los vértices. En cada iteración, se busca el vértice con la menor distancia en el arreglo, lo que toma $O(V)$ tiempo, donde V es el número de vértices. La actualización de las distancias toma $O(1)$ tiempo. Por lo tanto, la complejidad total es $O(V^2+E) = O(V^2)$, donde E es el número de aristas.
- **Lista enlazada (Linked List):** Similar al arreglo, la búsqueda del vértice con la menor distancia toma $O(V)$ tiempo. La actualización de las distancias toma $O(1)$ tiempo. La complejidad total es $O(V^2)$.
- **Montículo Binario (Binary Heap):** Un montículo binario es una estructura de datos que permite encontrar el elemento mínimo en $O(1)$ tiempo y eliminarlo en $O(\log V)$ tiempo. La

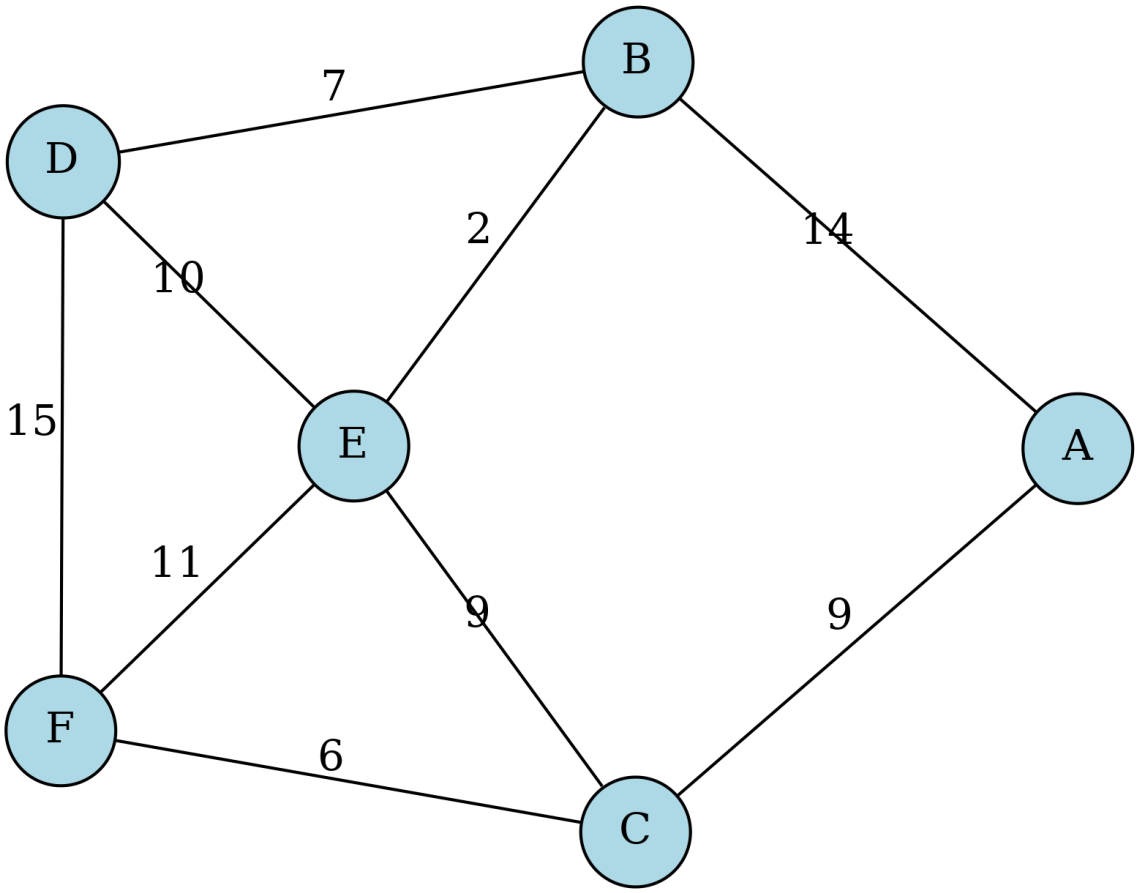


Figure 18: Grafo ponderado para buscar caminos más cortos

actualización de las distancias toma $O(\log V)$ tiempo. Por lo tanto, la complejidad total es $O((V + E) \log V)$. En grafos densos, donde $E \approx V^2$, la complejidad se acerca a $O(V^2 \log V)$.

- **Montículo de Fibonacci (Fibonacci Heap):** Un montículo de Fibonacci es una estructura de datos más avanzada que permite encontrar el elemento mínimo en $O(1)$ tiempo (amortizado) y eliminarlo en $O(\log V)$ tiempo (amortizado). La actualización de las distancias toma $O(1)$ tiempo (amortizado). Por lo tanto, la complejidad total es $O(V \log V + E)$. Esta es la implementación más eficiente para grafos grandes y dispersos.

En resumen, la complejidad del algoritmo de Dijkstra depende de la implementación de la cola de prioridad:

- Arreglo: $O(V^2)$
- Lista enlazada: $O(V^2)$
- Montículo Binario: $O((V + E) \log V)$
- Montículo de Fibonacci: $O(V \log V + E)$

La elección de la implementación de la cola de prioridad depende del tamaño y la densidad del grafo. Para grafos pequeños, un arreglo o una lista enlazada pueden ser suficientes. Para grafos

grandes y dispersos, un montículo de Fibonacci es la mejor opción. Para grafos grandes y densos, un montículo binario puede ser una buena opción.