Resumen del Cormen, Introduction to Algorithms, temas primer parcial 2025 primer cuatrimestre.

Backtracking/DFS

1. Representations of graphs

Los grafos son estructuras fundamentales en ciencias de la computación. La representación de un grafo tiene un impacto directo en la eficiencia de los algoritmos que lo manipulan. En el capítulo 22.1 del libro *Introduction to Algorithms* de Cormen, se presentan dos formas principales de representar grafos: **listas de adyacencia** y **matrices de adyacencia**. Cada una tiene ventajas particulares según el tipo de grafo (dirigido o no dirigido) y las operaciones requeridas.

Definiciones Básicas

Un grafo $G = (V, E)$ está compuesto por un conjunto de **vértices** V y un conjunto de **aristas** E. Si el grafo es dirigido, cada arista es un par ordenado (u, v). Si es no dirigido, se representa como un conjunto no ordenado $\{u, v\}$.

Lista de Adyacencia

En esta representación:

- Para cada vértice $u \in V$, se mantiene una lista `Adj[`u`]` que contiene todos los vértices v tales que $(u, v) \in E$.
- Se utiliza comúnmente con grafos dispersos (es decir, con pocas aristas en comparación con $|V|^2$).
- El espacio total utilizado es $\mathcal{O}(V + E)$.
- Es eficiente para iterar sobre los vecinos de un vértice.

Ventajas

- Menor uso de memoria en grafos dispersos.
- Iterar sobre vecinos de un nodo u es eficiente: $\mathcal{O}(\text{grado}(u))$.

Desventajas

- Consultar si una arista (u, v) existe puede costar $\mathcal{O}(\text{grado}(u))$.

Matriz de Adyacencia

Esta representación consiste en una matriz A de dimensión $|V| \times |V|$ tal que:

$$A[u][v] = \begin{cases} 1 & \text{si } (u, v) \in E \\ 0 & \text{en otro caso} \end{cases}$$

Ventajas

- Consultar si una arista (u, v) existe es una operación en tiempo constante: $\mathcal{O}(1)$.
- Es útil para grafos densos o cuando se requiere acceso rápido a cualquier par de vértices.

Desventajas

- Usa $\mathcal{O}(V^2)$ espacio, incluso si hay pocas aristas.
- Iterar sobre los vecinos de un vértice cuesta $\mathcal{O}(V)$.

Conclusión

La elección de la representación adecuada depende del tipo de grafo y las operaciones que se deseen realizar. Para grafos dispersos, se recomienda usar listas de adyacencia por su eficiencia espacial. Para operaciones frecuentes de consulta de existencia de aristas, una matriz de adyacencia puede ser más apropiada, especialmente en grafos densos.

2. Breadth-first search (BFS)

Introducción

La **búsqueda en anchura (BFS)** es un algoritmo fundamental para recorrer grafos. Dado un vértice fuente s, BFS explora el grafo por çapas ": primero todos los vértices a una distancia de 1, luego los de distancia 2, y así sucesivamente. Este algoritmo es especialmente útil en grafos no ponderados para encontrar caminos más cortos desde una fuente.

Aplicaciones Clave

- Calcular la distancia mínima desde un nodo fuente a todos los demás vértices.
- Verificar si un grafo es conexo.
- Construcción de árboles de expansión.
- Resolver problemas en grafos no ponderados como caminos mínimos, componentes conectados, entre otros.

Conceptos y Estructuras

Para implementar BFS se utilizan:

- Una **cola** (FIFO) para mantener el orden de exploración.
- Un arreglo o mapa de `color` para marcar el estado de cada vértice:
  - Blanco: no visitado.
  - Gris: visitado, pero con vecinos aún sin explorar.
  - Negro: totalmente explorado.
- Un arreglo de `distancia` para guardar la distancia desde s.
- Un arreglo de `predecesores` para reconstruir caminos.

Algoritmo

Inicialización: Todos los vértices comienzan blancos, con distancia infinita y sin predecesor. Solo el nodo fuente s comienza gris, distancia cero.

Pseudocódigo de BFS (versión simplificada):

```
BFS(G, s):
  for cada vértice u en V:
    color[u] = blanco
    distancia[u] =
    predecesor[u] = NIL

  color[s] = gris
  distancia[s] = 0
  predecesor[s] = NIL

  Q = cola vacía
  ENCOLAR(Q, s)

  while Q no esté vacía:
    u = DESENCOLAR(Q)
    for cada v en Adj[u]:
      if color[v] == blanco:
        color[v] = gris
        distancia[v] = distancia[u] + 1
        predecesor[v] = u
        ENCOLAR(Q, v)
    color[u] = negro
```

Complejidad

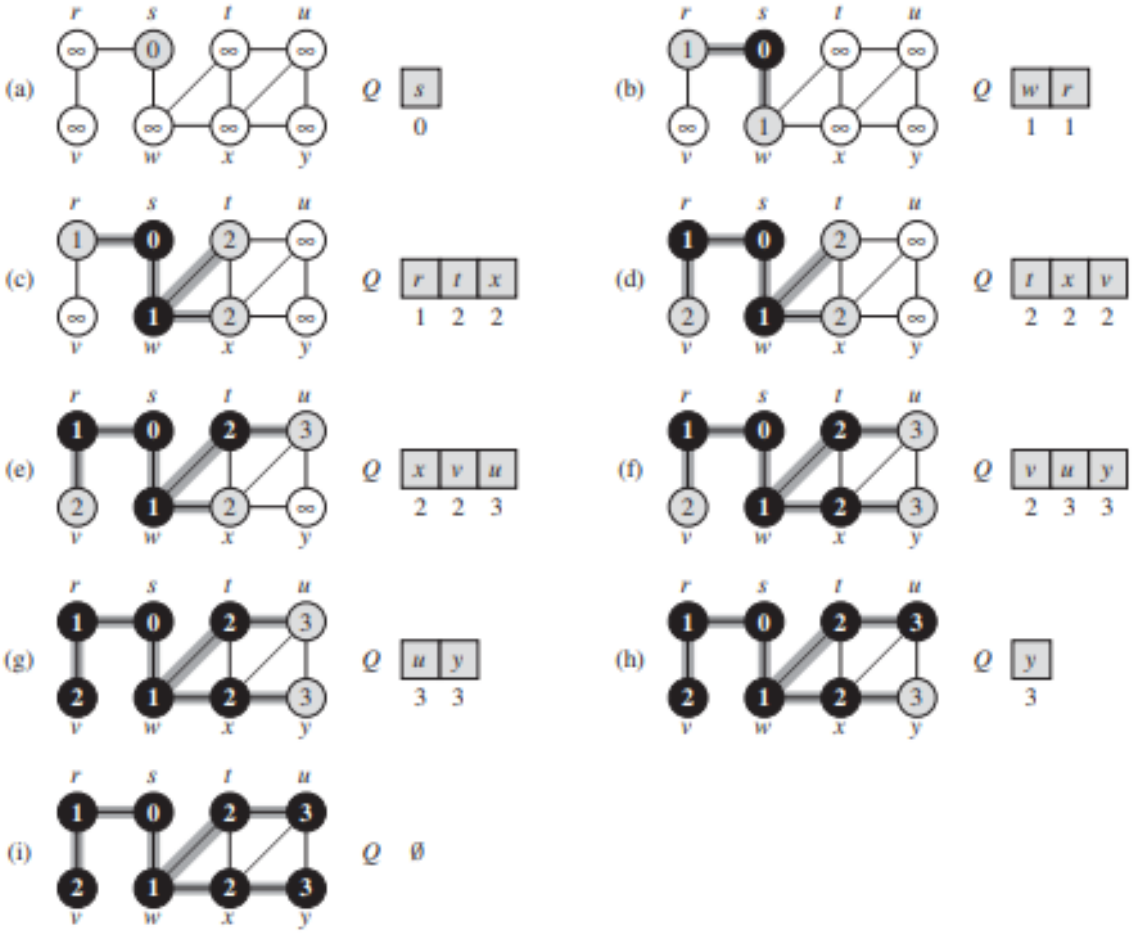
- Tiempo: $\mathcal{O}(V + E)$ para una representación con listas de adyacencia.
- Espacio: $\mathcal{O}(V)$ para las estructuras auxiliares.

Propiedades Importantes

- El árbol generado por BFS es un **árbol de expansión en anchura** que contiene los caminos más cortos desde s a cada nodo alcanzable.
- La distancia almacenada en cada vértice v es igual al número mínimo de aristas entre s y v.
- Si se sigue el arreglo de predecesores, se puede reconstruir el camino más corto.

Ejemplo Visual

"*Ejecución de BFS*"



Notas Finales

BFS es un algoritmo simple pero poderoso. Su capacidad para encontrar caminos más cortos lo hace útil en redes, juegos, inteligencia artificial y más. Se diferencia de DFS (Búsqueda en Profundidad) no solo en el orden de exploración, sino también en sus propiedades matemáticas.

3. Búsqueda en Profundidad (DFS)

Introducción

La **búsqueda en profundidad** (*Depth-First Search*, DFS) es un algoritmo clásico para recorrer o explorar grafos. A diferencia de BFS, que explora por niveles, DFS se sumerge lo más posible a lo largo de un camino antes de retroceder. DFS se utiliza como base para algoritmos complejos como el ordenamiento topológico, la detección de componentes fuertemente conexos y la identificación de ciclos.

Idea General

DFS comienza desde un vértice fuente y explora recursivamente a sus vecinos no visitados. Cada vértice es marcado con:

- `d[u]`: Tiempo de descubrimiento (cuando se visita por primera vez).
- `f[u]`: Tiempo de finalización (cuando se ha explorado completamente).

Pseudocódigo de DFS

```
DFS(G):
  for cada vértice u en V:
    color[u] = blanco
    predecesor[u] = NIL
  tiempo = 0
  for cada vértice u en V:
    if color[u] == blanco:
      DFS-Visitar(G, u)

DFS-Visitar(G, u):
  tiempo = tiempo + 1
  d[u] = tiempo
  color[u] = gris
  for cada v en Adj[u]:
    if color[v] == blanco:
      predecesor[v] = u
      DFS-Visitar(G, v)
  color[u] = negro
  tiempo = tiempo + 1
  f[u] = tiempo
```

Clasificación de Aristas

Durante la ejecución de DFS, las aristas pueden clasificarse como:

- **Aristas de árbol:** Conectan un vértice a un nuevo vértice descubierto.
- **Aristas de retroceso:** Conectan un vértice a un ancestro en el árbol DFS.

- **Aristas directas:** (solo en grafos dirigidos) conectan un vértice con un descendiente que ya ha sido finalizado.
- **Aristas cruzadas:** Conectan vértices en ramas separadas del árbol DFS.

Teorema 1 (Propiedad del tiempo de descubrimiento y finalización)**.** *Para cada vértice u del grafo, durante la ejecución de DFS se asignan dos valores: $d[u]$ (tiempo de descubrimiento) y $f[u]$ (tiempo de finalización), de modo que:*

$$d[u] < f[u]$$

Demostración. Por construcción del algoritmo DFS, cuando se visita un vértice u, se incrementa el contador global de tiempo y se guarda en $d[u]$. Luego, tras recorrer todos sus vecinos recursivamente, se vuelve a incrementar el tiempo y se guarda en $f[u]$. Como estas dos operaciones son secuenciales y no se reinicia el tiempo, se tiene que $d[u] < f[u]$ para todo u. □

Teorema 2 (Teorema de los intervalos de tiempo)**.** *Sean u y v dos vértices del grafo. Entonces, uno de los siguientes es cierto para sus intervalos $[d[u], f[u]]$ y $[d[v], f[v]]$:*

- *Los intervalos son disjuntos.*
- *Uno está contenido en el otro.*

Demostración. La ejecución de DFS genera un árbol (o un bosque). Cuando se descubre un vértice u, se marcan $d[u]$ y luego se comienza a recorrer sus vecinos. Si desde u se llega a v, entonces toda la exploración de v (y sus descendientes) ocurre antes de completar la de u. Por lo tanto:

$$d[u] < d[v] < f[v] < f[u]$$

lo que implica que el intervalo $[d[v], f[v]]$ está contenido dentro de $[d[u], f[u]]$.

Si u y v no están en la misma rama del árbol DFS, sus exploraciones no se intercalan y los intervalos son disjuntos. □

Teorema 3 (Clasificación de aristas en grafos dirigidos)**.** *Durante la ejecución de DFS en un grafo dirigido, cada arista $(u, v) \in E$ puede clasificarse en una de las siguientes categorías:*

- ***Arista de árbol:*** *Si v fue descubierto por primera vez por la llamada recursiva a partir de u.*
- ***Arista de retroceso:*** *Si v es un ancestro de u en el árbol DFS (es decir, v aún no ha terminado cuando se explora (u, v)).*
- ***Arista directa:*** *Si v ya fue visitado y es descendiente de u (solo en grafos dirigidos).*
- ***Arista cruzada:*** *Si conecta dos ramas distintas del bosque DFS o conecta a un vértice que ya ha sido finalizado.*

Demostración. El algoritmo DFS asigna a cada vértice tiempos de descubrimiento y finalización. Sea (u, v) una arista. Al momento de explorarla, dependiendo de los valores de $d[v]$ y $f[v]$ respecto a $d[u]$ y $f[u]$, se puede determinar su clasificación:

- Si v es blanco al explorarla, entonces es una arista de árbol.
- Si v es gris, se está regresando a un ancestro: arista de retroceso.
- Si v es negro y $d[u] < d[v]$, entonces es una arista directa.
- Si v es negro y $d[u] > d[v]$, es una arista cruzada.

Estas reglas derivan directamente del modo en que DFS establece el orden temporal de visitas y finalizaciones. □

Corolario 1. *Un grafo dirigido contiene un ciclo si y solo si durante la ejecución de DFS se encuentra al menos una arista de retroceso.*

Demostración. **(ida)** Si hay un ciclo, existe una secuencia de vértices $u_1 \to u_2 \to \cdots \to u_k \to u_1$. Durante DFS, al llegar a u_k y explorar (u_k, u_1), u_1 ya estará en el proceso de ser explorado (gris), lo cual es la definición de arista de retroceso.

(vuelta) Si existe una arista de retroceso (u, v), entonces v es ancestro de u en el árbol DFS. Por lo tanto, existe un camino de llamadas recursivas de v a u, más la arista (u, v), que forma un ciclo. □

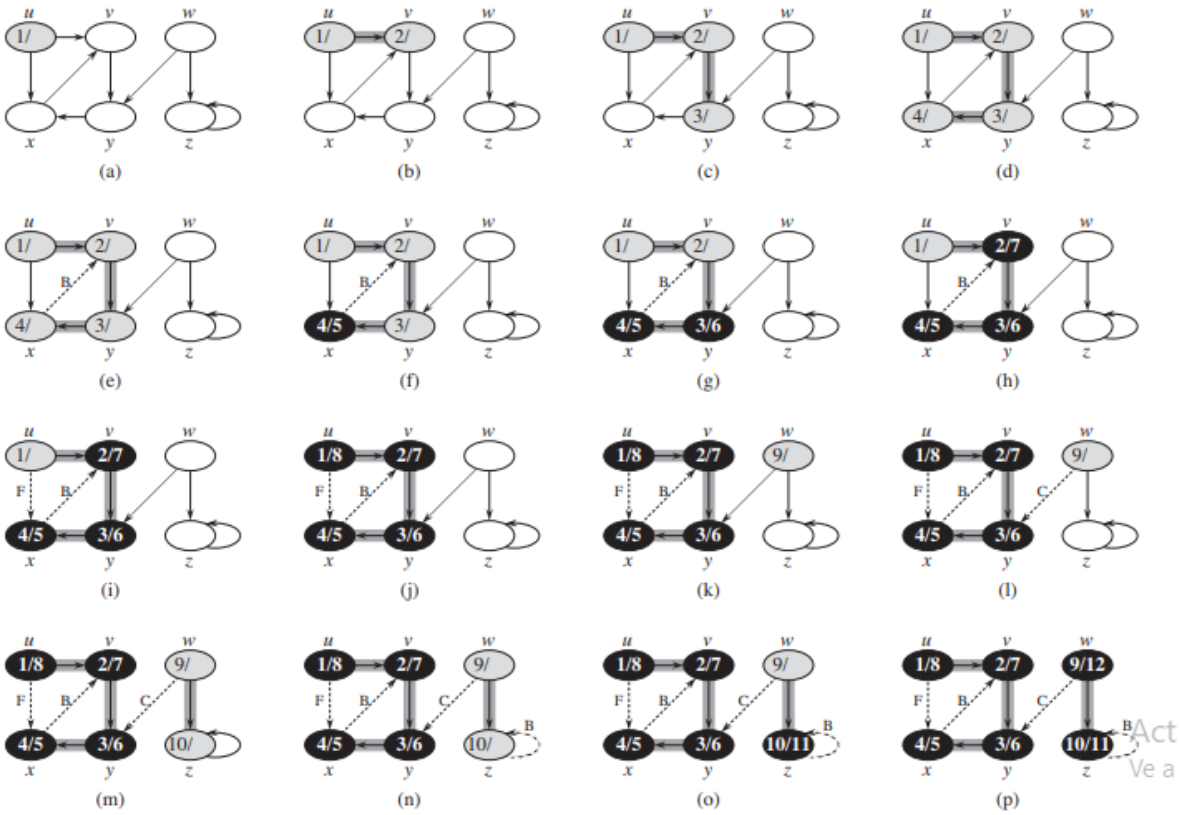
Aplicaciones de DFS

- **Detección de ciclos:** Si se encuentra una arista de retroceso, el grafo contiene un ciclo.
- **Ordenamiento topológico:** Aplicable a grafos dirigidos acíclicos (DAGs). Se obtiene ordenando los vértices por tiempos de finalización en orden decreciente.
- **Componentes conexos (no dirigidos):** Cada árbol del bosque DFS representa un componente conexo.
- **Componentes fuertemente conexos (dirigidos):** Base del algoritmo de Kosaraju.

Complejidad

- Tiempo: $\mathcal{O}(V + E)$ en grafos representados con listas de adyacencia.
- Espacio: $\mathcal{O}(V)$ para almacenar colores, tiempos y predecesores.

"*Ejecución de DFS y clasificación de aristas*"



Conclusión

DFS es una herramienta poderosa que revela la estructura profunda de un grafo. Su uso va más allá de la simple exploración, permitiendo resolver problemas complejos como ciclos, ordenamientos y particiones.

Dynamic Programming

4. Dynamic Programming

¿Qué es la Programación Dinámica?

La programación dinámica es una metodología de diseño de algoritmos que se basa en resolver subproblemas más pequeños y combinar sus soluciones para abordar problemas más grandes. Es especialmente útil cuando un problema exhibe las siguientes propiedades:

- **Subproblemas superpuestos**: El problema puede descomponerse en subproblemas que se repiten múltiples veces.
- **Óptima subestructura**: La solución óptima del problema puede construirse a partir de soluciones óptimas de sus subproblemas.

Ejemplo Clásico: Números de Fibonacci

Un ejemplo clásico para ilustrar la programación dinámica es el cálculo de los números de Fibonacci. La definición recursiva es:

$$F(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ F(n-1) + F(n-2) & \text{si } n > 1. \end{cases}$$

Una implementación recursiva directa puede ser ineficiente debido a la recalculación de los mismos valores. A continuación, se muestra una implementación utilizando programación dinámica con memoización:

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```

Listing 1: Cálculo de Fibonacci con memoización

Enfoques en Programación Dinámica

Existen dos enfoques principales en programación dinámica:

Memoización (Top-Down)

Este enfoque utiliza recursión y almacena los resultados de los subproblemas en una estructura de datos (como un diccionario) para evitar cálculos redundantes.

Tabulación (Bottom-Up)

Este enfoque construye una tabla de soluciones desde los casos base hasta el problema original, evitando la recursión.

Aplicaciones Comunes

La programación dinámica se aplica en diversos problemas, como:

- **Problema de la mochila**: Seleccionar elementos con pesos y valores para maximizar el valor total sin exceder la capacidad.
- **Edición de cadenas**: Calcular la distancia mínima de edición entre dos cadenas.
- **Cadenas de matrices**: Determinar la forma más eficiente de multiplicar una cadena de matrices.

Conclusión

La programación dinámica es una herramienta esencial en el diseño de algoritmos eficientes. Al identificar subproblemas repetitivos y almacenar sus soluciones, podemos mejorar significativamente el rendimiento de nuestros programas. Aunque puede requerir un esfuerzo adicional para identificar la estructura adecuada del problema, los beneficios en términos de eficiencia y claridad del código son indiscutibles.

5. Enfoque Top-Down (con Memoización)

Introducción

El enfoque **Top-Down**, también conocido como *memoización*, es una técnica que combina recursión con almacenamiento de resultados intermedios. Se comienza resolviendo el problema original y, a medida que se realizan llamadas recursivas, se almacenan las soluciones de los subproblemas para evitar cálculos redundantes.

Características

- Utiliza recursión para descomponer el problema.
- Almacena los resultados de subproblemas ya resueltos.
- Evita la recalculación de subproblemas.
- Puede tener un mayor consumo de memoria debido al uso de la pila de llamadas.

Ejemplo: Cálculo de Fibonacci

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```

Listing 2: Implementación Top-Down de Fibonacci

Ventajas y Desventajas

Ventajas:

- Fácil de implementar y entender.
- Evita cálculos redundantes.

Desventajas:

- Puede consumir más memoria debido a la recursión.
- Riesgo de desbordamiento de pila para entradas grandes.

6. Enfoque Bottom-Up (Tabulación)

Introducción

El enfoque **Bottom-Up**, también conocido como *tabulación*, resuelve primero los subproblemas más pequeños y utiliza sus soluciones para construir las soluciones de subproblemas más grandes, hasta llegar al problema original. Este enfoque es iterativo y no utiliza recursión.

Características

- Utiliza una estructura de datos (como una tabla) para almacenar soluciones.
- Resuelve los subproblemas en orden ascendente.
- Evita el uso de la pila de llamadas, reduciendo el riesgo de desbordamiento.
- Puede ser más eficiente en términos de tiempo y espacio.

Ejemplo: Cálculo de Fibonacci

```
def fibonacci(n):
    if n <= 1:
        return n
    fib = [0, 1]
    for i in range(2, n+1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]
```

Listing 3: Implementación Bottom-Up de Fibonacci

Ventajas y Desventajas

Ventajas:

- Mayor eficiencia en tiempo y espacio.
- No utiliza recursión, evitando desbordamientos de pila.

Desventajas:

- Puede ser menos intuitivo de implementar.
- Requiere conocer el orden de resolución de los subproblemas.

7. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford es una herramienta fundamental en teoría de grafos, utilizada para encontrar los caminos más cortos desde un vértice fuente a todos los demás en un grafo dirigido con pesos, incluso si algunos de estos pesos son negativos. A diferencia del algoritmo de Dijkstra, Bellman-Ford puede manejar aristas con pesos negativos y detectar ciclos de peso negativo, lo que lo hace especialmente útil en diversas aplicaciones, como enrutamiento de redes y análisis financiero.

Descripción del Algoritmo

Dado un grafo dirigido $G = (V, E)$ con función de peso $w : E \to \mathbb{R}$ y un vértice fuente $s \in V$, el objetivo es encontrar el camino más corto desde s a cada vértice $v \in V$.

El algoritmo inicializa las distancias desde s a todos los demás vértices como infinito, y la distancia desde s a sí mismo como cero. Luego, realiza $|V| - 1$ iteraciones, relajando todas las aristas en cada iteración. Finalmente, verifica la existencia de ciclos de peso negativo.

Teorema de Correctitud

Teorema 4. *Después de realizar $|V| - 1$ iteraciones de relajación de todas las aristas, si no existen ciclos de peso negativo alcanzables desde s, entonces las distancias calculadas representan los pesos de los caminos más cortos desde s a cada vértice $v \in V$.*

Demostración. La demostración se basa en la inducción sobre el número de aristas en el camino más corto. En cada iteración, el algoritmo encuentra los caminos más cortos que utilizan a lo sumo i aristas. Dado que el camino más corto entre dos vértices no contiene ciclos, su longitud máxima es $|V| - 1$ aristas. Por lo tanto, después de $|V| - 1$ iteraciones, se han encontrado todos los caminos más cortos. □

Corolario: Detección de Ciclos de Peso Negativo

Corolario 2. *Si después de las $|V| - 1$ iteraciones, existe una arista $(u, v) \in E$ tal que $d[v] > d[u] + w(u, v)$, entonces el grafo contiene un ciclo de peso negativo alcanzable desde s.*

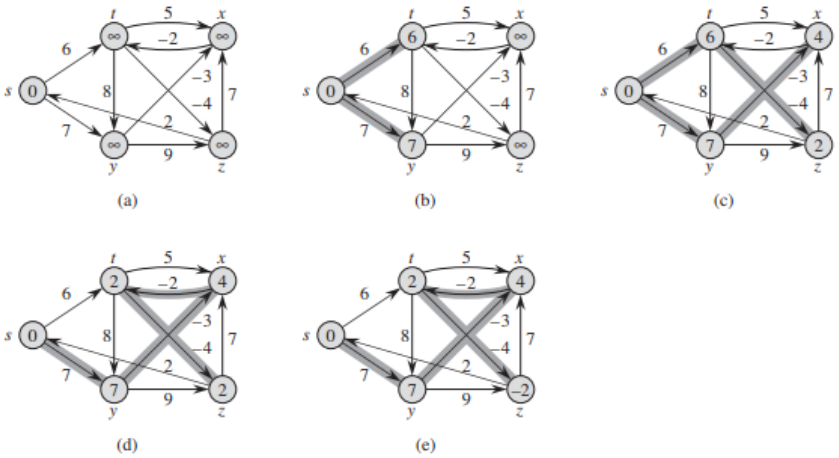
Demostración. Si tal arista existe, significa que aún es posible reducir la distancia a v, lo que implica la existencia de un ciclo que permite disminuir indefinidamente la distancia, es decir, un ciclo de peso negativo. □

Implementación en Python

```python
def bellman_ford(G, w, s):
    # Inicializacion
    d = {v: float('inf') for v in G}
    d[s] = 0
    pred = {v: None for v in G}

    # Relajacion de aristas
    for _ in range(len(G) - 1):
        for u in G:
            for v in G[u]:
                if d[v] > d[u] + w(u, v):
                    d[v] = d[u] + w(u, v)
                    pred[v] = u

    # Deteccion de ciclos de peso negativo
```

```
    for u in G:
        for v in G[u]:
            if d[v] > d[u] + w(u, v):
                raise ValueError("El grafo contiene un ciclo de
                    peso negativo")

    return d, pred
```

Listing 4: Implementación del algoritmo de Bellman-Ford



Conclusión

El algoritmo de Bellman-Ford es una herramienta poderosa para encontrar caminos más cortos en grafos con pesos negativos y para detectar ciclos de peso negativo. Aunque es menos eficiente que el algoritmo de Dijkstra en grafos sin pesos negativos, su capacidad para manejar una gama más amplia de grafos lo hace indispensable en muchas aplicaciones prácticas.

8. Algoritmo de Floyd-Warshall

En el mundo de la informática y las redes, encontrar las rutas más cortas entre todos los pares de nodos en un grafo es una tarea fundamental. El algoritmo de Floyd-Warshall nos proporciona una solución eficiente y elegante a este problema, utilizando programación dinámica para calcular las distancias mínimas entre todos los pares de vértices en un grafo ponderado. Este algoritmo es especialmente útil en aplicaciones donde se requiere conocer las distancias más cortas entre múltiples puntos, como en sistemas de navegación y análisis de redes.

Descripción del Algoritmo

Dado un grafo dirigido $G = (V, E)$ con una función de peso $w : E \to \mathbb{R}$, el objetivo es encontrar el camino más corto entre todos los pares de vértices. El algoritmo de Floyd-Warshall utiliza una matriz de distancias D y una matriz de predecesores Π para almacenar las distancias mínimas y reconstruir los caminos más cortos, respectivamente.

Inicialización

Se inicializa la matriz de distancias D de la siguiente manera:

$$D^{(0)}[i][j] = \begin{cases} 0 & \text{si } i = j \\ w(i,j) & \text{si } (i,j) \in E \\ \infty & \text{si } (i,j) \notin E \end{cases}$$

Iteración

Para cada vértice intermedio k de 1 a n, se actualiza la matriz de distancias:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

Teorema de Correctitud

Teorema 5. *Después de la k-ésima iteración, $D^{(k)}[i][j]$ contiene la longitud del camino más corto desde i hasta j que utiliza únicamente los vértices intermedios del conjunto $\{1, 2, ..., k\}$.*

Demostración. La demostración se basa en la inducción sobre k. Para $k = 0$, $D^{(0)}[i][j]$ es igual al peso directo de la arista o infinito si no existe. Suponiendo que $D^{(k-1)}[i][j]$ es correcto, al considerar el vértice k como intermedio, se actualiza $D^{(k)}[i][j]$ si se encuentra un camino más corto pasando por k. □

Corolario: Detección de Ciclos de Peso Negativo

Corolario 3. *Si existe un ciclo de peso negativo en el grafo, entonces al menos uno de los elementos diagonales de la matriz de distancias final $D^{(n)}[i][i]$ será negativo.*

Demostración. Un ciclo de peso negativo implica que existe un camino desde un vértice i a sí mismo con peso negativo. Durante las iteraciones del algoritmo, este valor se reflejará en la diagonal de la matriz de distancias. □

Implementación en Python

```python
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
```

```
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

Listing 5: Implementación del algoritmo de Floyd-Warshall

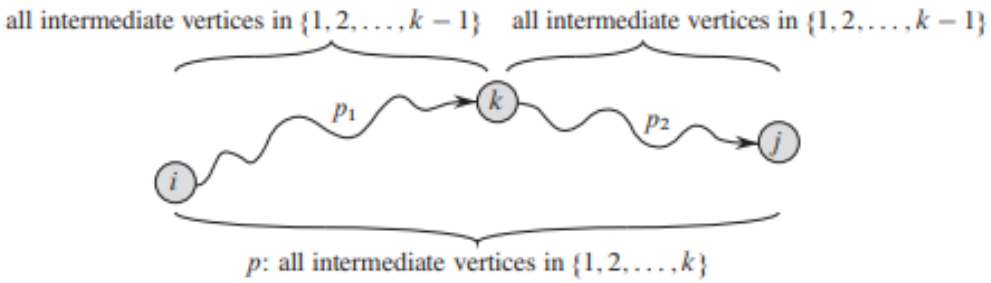


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

Aplicaciones

El algoritmo de Floyd-Warshall tiene diversas aplicaciones, entre ellas:

- Cálculo de rutas más cortas en redes de transporte y comunicación.
- Análisis de accesibilidad en grafos.
- Detección de ciclos de peso negativo en grafos.
- Resolución de problemas en teoría de grafos y optimización.

Análisis de Complejidad

La complejidad temporal del algoritmo de Floyd-Warshall es:

$$\mathcal{O}(V^3)$$

dado que hay tres bucles anidados sobre los vértices del grafo. Esto lo hace eficiente para grafos densos y con número de vértices moderado, pero inadecuado para grafos grandes y dispersos.

La complejidad espacial es:

$$\mathcal{O}(V^2)$$

ya que se necesita almacenar una matriz de distancias y, opcionalmente, una matriz de predecesores para reconstruir los caminos.

Reconstrucción de Caminos

Además de conocer la distancia mínima entre los nodos, frecuentemente se desea conocer el camino concreto. Para esto, se utiliza una matriz de predecesores $\Pi[i][j]$ que indica el vértice anterior a j en el camino más corto desde i.

```
for i in range(n):
    for j in range(n):
        if i == j or graph[i][j] == float('inf'):
            pred[i][j] = None
        else:
            pred[i][j] = i
```

Listing 6: Inicialización de matriz de predecesores

Función para reconstruir el camino:

```
def reconstruir_camino(pred, i, j):
    if pred[i][j] is None:
        return []
    camino = [j]
    while i != j:
        j = pred[i][j]
        camino.insert(0, j)
    return camino
```

Ejemplo Práctico

Consideremos el siguiente grafo dirigido ponderado:

$$\begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

Después de ejecutar Floyd-Warshall, obtendremos la matriz de distancias mínimas entre todos los pares. Puedes representar este ejemplo con una imagen como:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Ventajas y Limitaciones

Ventajas

- Fácil de implementar.
- Maneja pesos negativos (sin ciclos negativos).

- Proporciona caminos más cortos entre todos los pares de nodos.

Limitaciones

- No es eficiente en grafos dispersos con muchos vértices.
- No permite pesos negativos en ciclos.

Extensiones

Floyd-Warshall puede extenderse para:

- **Cierre transitivo**: reemplazar suma y mínimo por disyunción y conjunción lógica.
- **Contar caminos más cortos**: llevar un contador en cada celda.
- **Floyd-Warshall para matrices booleanas**: útil en teoría de lenguajes y análisis de accesibilidad.

Conclusión

El algoritmo de Floyd-Warshall es una joya de la programación dinámica. Aunque su complejidad cúbica puede ser un obstáculo en grafos grandes, su simplicidad y poder lo hacen muy útil en contextos donde se necesita conocer todas las distancias posibles. Más allá de la teoría, el algoritmo nos enseña cómo descomponer un problema grande en subproblemas manejables, y cómo iterar hacia la solución óptima.

Entender Floyd-Warshall no es sólo saber programarlo: es entender la belleza de resolver todos los caminos con una sola estructura coherente. Y eso es, sin dudas, una muestra del arte del algoritmo.

9. DAG

En muchos problemas—como la resolución de dependencias de módulos, la planificación de tareas o la ordenación de cursos con prerequisitos—surge la necesidad de linealizar un grafo dirigido acíclico (DAG) respetando sus aristas. El *ordenamiento topológico* brinda precisamente esa herramienta: una forma sistemática de listar todos los vértices de un DAG de modo que cada arista vaya siempre de un vértice anterior a uno posterior. En esta sección exploraremos el algoritmo basado en *Depth-First Search* (DFS), sus garantías de corrección y sus propiedades teóricas más relevantes.

Definición y Propiedades Básicas

- Un **ordenamiento topológico** de un DAG $G = (V, E)$ es una permutación σ de V tal que, para cada arista $(u, v) \in E$, u aparece antes que v en σ.
- Sólo los DAG admiten ordenamientos topológicos; si G contiene ciclos, no existe secuencia que satisfaga la condición anterior.

Lema Fundamental

[Caracterización de DAGs] Un grafo dirigido G es acíclico si y sólo si una búsqueda en profundidad (DFS) de G no revela *back edges*.

Esquema de demostración. ($\Rightarrow$) Si existe un *back edge* (u, v), entonces v es ancestro de u en el árbol DFS, lo que cierra un ciclo.
($\Leftarrow$) Si G contiene un ciclo C, sea v el primer vértice descubierto en C; el recorrido DFS seguirá la arista anterior en C antes de cerrar el ciclo, generando un *back edge*. □

Algoritmo TOPOLOGICAL-SORT

```
TOPOLOGICAL-SORT(G):
  marcar todos los v rtices como \texttt{no visitado}
  L = lista vac a
  for cada v rtice u in V:
    if u no visitado:
      DFS-Visit(u)
  return L

DFS-Visit(u):
  marcar u como \texttt{visitando}
  for cada v en Adj[u]:
    if v no visitado:
      DFS-Visit(v)
  marcar u como \texttt{terminado}
  insertar u al frente de L
```

Listing 7: Pseudocódigo de TOPOLOGICAL-SORT

Correctitud y Garantías

Teorema de Correctitud

Teorema 6. *Si G es un DAG, entonces `TOPOLOGICAL-SORT(G)` produce un ordenamiento topológico válido.*

Esquema de demostración. Cuando DFS explora (u, v), si v no ha sido terminado aún, se invoca recursivamente. Por la inserción al frente de la lista `L`, al finalizar u, éste queda delante de todos sus descendientes. Así, para cada arista (u, v), u aparece antes que v en la lista final. Dado que no hay *back edges* (por ser DAG), no se viola la propiedad. □

Corolario: Unicidad Parcial

Corolario 4. *El ordenamiento topológico de un DAG no es único en general. Es único si y sólo si el DAG contiene un camino Hamiltoniano, es decir, para cada par de vértices consecutivos en la ordenación existe una arista que los conecta.*

Complejidad

- **Tiempo:** $O(|V| + |E|)$ — coste de una DFS más $O(1)$ por inserción en lista.
- **Espacio:** $O(|V|)$ — para marcar estados y pila de recursión.

Ejemplo Ilustrativo

Supongamos el DAG con aristas:

$$A \to C, \quad B \to C, \quad B \to D, \quad C \to E, \quad D \to F, \quad E \to F.$$

Un posible ordenamiento topológico es:

$$[A, B, C, D, E, F].$$

Al correr TOPOLOGICAL-SORT, los tiempos de descubrimiento/finalización dictan la inserción en lista que respeta todas estas precedencias.

Extensiones y Aplicaciones

Caminos Mínimos en DAGs

Dado un DAG ponderado, un orden topológico permite resolver caminos más cortos en tiempo lineal, relajando cada arista una sola vez en el orden obtenido.

Detección de Componentes Fuertemente Conexas

Al aplicar TOPOLOGICAL-SORT sobre el grafo de componentes (contraportado tras identificar SCCs con dos DFS), se garantiza procesar las componentes en orden topológico.

Conclusión

El ordenamiento topológico demuestra el poder de la búsqueda en profundidad para estructurar información sobre dependencias. A través de técnicas tan sencillas como pintar vértices y apilar al terminar, obtenemos una herramienta versátil que alimenta desde la compilación de proyectos hasta la optimización de rutas en DAGs ponderados. Entender su corrección y sus propiedades —como la no unicidad y la relación con el ciclo— refuerza la apreciación de cómo algoritmos simples construyen la base de soluciones más complejas en teoría de grafos y algoritmia avanzada.