Módulo 5 – Grafos y algoritmos de recorrido

Curso de Introducción a Estructuras de Datos y Algoritmos

21 de agosto de 2025

Contents

1 Módulo 5 – Grafos: Modelando un Mundo Conectado

1.1 Motivación: La Estructura de Datos Universal

Hasta ahora, hemos visto estructuras con reglas claras: las listas son secuenciales, los árboles son jerárquicos. Pero, ¿cómo modelamos un sistema donde las conexiones son libres y complejas? * En una **red social**, tú estás conectado a tus amigos, pero tus amigos también están conectados entre sí, formando una red intrincada. * En la **World Wide Web**, una página enlaza a otra, que a su vez enlaza a muchas más, creando una telaraña global. * En un **mapa de carreteras**, las ciudades están conectadas por múltiples rutas, con intersecciones y posibles caminos circulares.

Las estructuras lineales y jerárquicas son insuficientes para representar esta realidad. Los **grafos** son la respuesta. Son la estructura de datos más general y poderosa, capaz de

modelar cualquier sistema de “entidades” y las “relaciones” entre ellas.

Pasar de los árboles a los grafos es un salto conceptual fundamental: abandonamos la seguridad de la jerarquía para abrazar la flexibilidad del caos conectado. Aprender a navegar y analizar estas estructuras nos da las herramientas para resolver algunos de los problemas más interesantes y complejos de la computación.

1.2 Conceptos Fundamentales de los Grafos

Un **grafo** $G = (V, E)$ es un par compuesto por un conjunto de **vértices** (o nodos) V y un conjunto de **aristas** (o arcos) E que conectan pares de vértices.

1.2.1 La Taxonomía de los Grafos

- **Grafo No Dirigido**: Las aristas son bidireccionales. Si existe una arista $\{u, v\}$, se puede ir de u a v y de v a u. Modela relaciones simétricas, como la amistad en Facebook.
- **Grafo Dirigido (Dígrafo)**: Las aristas son flechas con una sola dirección. Una arista (u, v) va de u hacia v, pero no necesariamente al revés. Modela relaciones asimétricas, como “seguir” a alguien en Twitter o los hipervínculos en la web.
- **Grafo Ponderado**: A cada arista se le asigna un “peso” o “coste” numérico. Este peso puede representar distancia, tiempo, capacidad, etc. Son esenciales para problemas de optimización, como encontrar la ruta más corta en un mapa.
- **Grafo Disperso vs. Denso**: Un grafo es **disperso** si tiene pocas aristas en comparación con el número máximo posible ($|E| \ll |V|^2$). Un grafo es **denso** si se acerca a ese máximo. Esta distinción es clave para elegir la representación correcta.

1.2.2 Representaciones: El Mapa del Grafo

Elegir cómo almacenar un grafo en memoria es la primera decisión crucial, con un gran impacto en el rendimiento.

Característica	Matriz de Adyacencia ($O(\|V\|^2)$)	Lista de Adyacencia ($O(\|V\| + \|E\|)$)
Uso de Memoria	Alto y fijo. Ineficiente para grafos dispersos.	Proporcional al número de aristas. Ideal para grafos dispersos.
Añadir Vértice	Costoso ($O(\|V\|^2)$), requiere reconstruir la matriz.	Fácil ($O(1)$).
Añadir Arista	Muy rápido ($O(1)$).	Rápido ($O(1)$).

Característica	Matriz de Adyacencia ($O(\|V\|^2)$)	Lista de Adyacencia ($O(\|V\| + \|E\|)$)
Verificar si existe $\{u, v\}$	Muy rápido ($O(1)$), acceso directo a la celda `M[u][v]`.	Lento ($O(k)$ donde k es el grado del vértice u).
Iterar sobre vecinos de u	Lento ($O(\|V\|)$), hay que recorrer toda la fila.	Óptimo ($O(k)$ donde k es el grado de u).
Ideal para	Grafos **densos** y problemas donde la verificación de aristas es constante.	Grafos **dispersos** (la mayoría de los casos reales).

1.2.3 Orígenes Históricos: Los Puentes de Königsberg

La teoría de grafos nació en 1736, cuando el gran matemático **Leonhard Euler** resolvió el problema de los **Siete Puentes de Königsberg**. La ciudad tenía siete puentes que conectaban dos islas y las dos orillas de un río. El acertijo era: ¿es posible dar un paseo que cruce cada puente exactamente una vez y regrese al punto de partida?

Euler abstrajo el problema: representó las zonas de tierra como **vértices** y los puentes como **aristas**. Demostró que tal paseo era imposible, sentando las bases de la teoría de grafos y demostrando el poder de la abstracción matemática.

1.3 Recorrido en Anchura (BFS): La Exploración por Ondas

El **BFS (Breadth-First Search)** explora el grafo de manera expansiva, como las ondas que se forman al lanzar una piedra al agua. Partiendo de un nodo origen, visita primero a todos sus vecinos directos, luego a los vecinos de sus vecinos, y así sucesivamente, nivel por nivel.

La clave de su comportamiento es el uso de una **cola (FIFO)**, que garantiza que los nodos se procesen en el orden en que fueron descubiertos.

1.3.1 BFS Paso a Paso: Un Ejemplo Guiado

Consideremos un grafo simple y apliquemos BFS desde el vértice **A**.

Estado Inicial: * Cola: `[]` * Visitados: `{}` * Orden de visita: `[]`

Paso 1: * Se visita **A**. * Cola: `[A]` * Visitados: `{A}`

Paso 2: * Se desencola **A**. Orden de visita: `[A]`. * Se encolan los vecinos no visitados de A (B y C). * Cola: `[B, C]` * Visitados: `{A, B, C}`

Paso 3: * Se desencola **B**. Orden de visita: `[A, B]`. * Se encola el vecino no visitado de B (D). * Cola: `[C, D]` * Visitados: `{A, B, C, D}`

Paso 4: * Se desencola **C**. Orden de visita: `[A, B, C]`. * Los vecinos de C (A y D) ya han sido visitados. No se encola nada. * Cola: `[D]` * Visitados: `{A, B, C, D}`

Paso 5: * Se desencola **D**. Orden de visita: `[A, B, C, D]`. * No tiene vecinos no visitados. * Cola: `[]`

La cola está vacía. El algoritmo termina. El recorrido es **A -> B -> C -> D**.

1.3.2 La "Superpotencia" de BFS: El Camino Más Corto

La propiedad más importante de BFS es que, en un **grafo no ponderado**, encuentra el camino más corto (en número de aristas) desde el nodo origen a todos los demás nodos alcanzables. Como explora por niveles, la primera vez que llega a un nodo, lo hace necesariamente por el camino más corto posible.

1.4 Recorrido en Profundidad (DFS): El Laberinto y el Retroceso

El **DFS (Depth-First Search)** explora el grafo de una manera completamente distinta. Es como un explorador en un laberinto: elige un camino y lo sigue hasta el final. Si llega a un callejón sin salida (o un nodo ya visitado), **retrocede (backtracking)** hasta la última bifurcación y prueba el siguiente camino disponible.

Este comportamiento se logra naturalmente con **recursión** (usando la pila de llamadas del sistema) o con una **pila (LIFO)** explícita.

1.4.1 DFS Paso a Paso: Un Ejemplo Guiado

Usando el mismo grafo y partiendo de **A**, un posible recorrido DFS recursivo sería:

1. **DFS(A)**: Marcar A como visitado. Orden: `[A]`. * Explorar el primer vecino de A: **B**. * Llamar a **DFS(B)**.
2. **DFS(B)**: Marcar B como visitado. Orden: `[A, B]`. * Explorar el primer vecino de B: **D**. * Llamar a **DFS(D)**.
3. **DFS(D)**: Marcar D como visitado. Orden: `[A, B, D]`. * Explorar el primer vecino de D: **C**. * Llamar a **DFS(C)**.
4. **DFS(C)**: Marcar C como visitado. Orden: `[A, B, D, C]`. * El vecino de C, A, ya está visitado. No hay más caminos. * **Retornar** a D.
5. (En D): No hay más vecinos por explorar. **Retornar** a B.

6. (En B): No hay más vecinos por explorar. **Retornar** a A.
7. (En A): Explorar el siguiente vecino: **C**. Ya está visitado. No hay más caminos.
8. El algoritmo termina. El recorrido es **A -> B -> D -> C**.

Nota importante: El orden exacto de un DFS puede variar dependiendo del orden en que se exploran los vecinos.

1.4.2 Aplicaciones Clave de DFS

- **Detección de Ciclos**: Durante un recorrido DFS, si encontramos un vértice que ya está en la pila de recursión actual (un ancestro), hemos encontrado un "back edge", lo que significa que hay un ciclo.
- **Ordenación Topológica**: En un Grafo Dirigido Acíclico (DAG), una ordenación topológica es una secuencia lineal de vértices tal que para cada arista (u, v), u aparece antes que v. Es fundamental para planificar tareas con dependencias (ej. compilar un proyecto). Un recorrido DFS postorden produce el inverso de una ordenación topológica.

1.5 Comparación: ¿BFS o DFS?

La elección entre BFS y DFS no es una cuestión de "cuál es mejor", sino de "**cuál es la herramienta adecuada para el trabajo**".

Característica	BFS (Anchura)	DFS (Profundidad)
Estructura de Datos	**Cola (FIFO)**	**Pila (LIFO) o Recursión**
Estrategia	Explora por niveles, de forma expansiva y uniforme.	Se sumerge en un camino hasta el final antes de retroceder.
Camino más corto	**Garantizado** en grafos no ponderados.	No lo garantiza. Puede encontrar un camino mucho más largo primero.
Uso de Memoria	Puede ser muy alto si el grafo es ancho (muchos nodos en un nivel).	Generalmente menor, proporcional a la profundidad máxima del grafo.

Característica	BFS (Anchura)	DFS (Profundidad)
Ideal para...	Encontrar el camino más corto, "rastreadores" web que exploran por cercanía, análisis de redes sociales (amigos de amigos).	Detección de ciclos, ordenación topológica, resolución de laberintos, búsqueda exhaustiva en árboles de decisión (IA, juegos).

1.6 Ejercicios Ampliados

1.6.0.1 Ejercicios Teóricos y de Diseño

1. Dado un grafo no dirigido y conexo, ¿es posible que el orden de visita de BFS y DFS sea el mismo? Si es así, dibuja un ejemplo. Si no, explica por qué.
2. Explica con detalle cómo modificarías el pseudocódigo de DFS para detectar un ciclo en un grafo **dirigido**. (Pista: necesitas tres estados para cada nodo: no visitado, visitando, visitado).
3. Diseña un algoritmo que, dado un grafo no dirigido, cuente el número de **componentes conexas** que tiene. ¿Usarías BFS o DFS? ¿Por qué?
4. Un grafo bipartito es aquel cuyos vértices se pueden dividir en dos conjuntos disjuntos tal que toda arista conecta un vértice de un conjunto con uno del otro. Diseña un algoritmo para determinar si un grafo es bipartito. (Pista: BFS y colores).

1.6.0.2 Ejercicios Prácticos de Programación

5. Implementa una clase `Grafo` que permita añadir vértices y aristas, y que pueda ser representado tanto por matriz como por lista de adyacencia.
6. Implementa las funciones `recorridoBFS` y `recorridoDFS` para tu clase Grafo.
7. Usando tu implementación de BFS, escribe una función que encuentre la distancia (número de aristas) entre dos nodos dados. Si no hay camino, debe devolver -1.
8. Escribe un programa que resuelva un laberinto simple representado por una matriz de caracteres ('#' para paredes, '.' para caminos, 'E' para entrada, 'S' para salida). Utiliza DFS para encontrar un camino desde la entrada hasta la salida.

1.7 Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms.* MIT Press.
- Sedgewick, R., & Wayne, K. *Algorithms.* Addison-Wesley.

- Gross, J. L., & Yellen, J. *Graph Theory and Its Applications*. Chapman & Hall/CRC.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. *Data Structures and Algorithms in Java*. Wiley.