Grafos

por Ariel Parra

[Γα=Ω5]

¿Qué es un grafo?

Una estructura de datos de grafo es una estructura de datos no lineal que consiste en nodos (vértices) y aristas (edges).
Aquí tienes la traducción al español junto con el grafo en Mermaid:

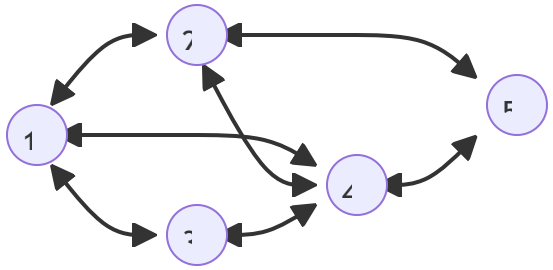
La mayoría de los problemas caen en al menos una de las siguientes dos categorías:

1. La estructura del grafo es especial (es un árbol, un camino, o un ciclo).
2. Para resolver el problema, solo necesitas iterar sobre la lista de adyacencia de cada vértice.

Algunos problemas relacionados con los grafos son:

1. ¿Está la ciudad `A` conectada con la ciudad `B`? Imagina una región como un grupo de ciudades en el cual cada ciudad en el grupo puede alcanzar cualquier otra ciudad del mismo grupo, pero no puede alcanzar ciudades de otros grupos. ¿Cuántas regiones hay en este mapa, y cuáles ciudades pertenecen a cada región?
2. ¿Cuál es la distancia más corta que debo recorrer para llegar de la ciudad `A` a la ciudad `B`?

Terminología



Camino

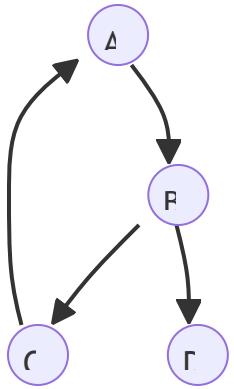
Un camino lleva de un nodo `a` a un nodo `b` a través de las aristas del grafo. La longitud de un camino es el número de aristas en él. Por ejemplo, el grafo anterior contiene un camino de longitud 3 de nodo 1 a nodo 5: `1→3→4→5`.

ciclo

Un camino es un ciclo si el primer y el último nodo son el mismo. Por ejemplo, el grafo anterior contiene un ciclo `1→3→4→1`. Un camino es simple si cada nodo aparece como máximo una vez en el camino.

Grafo dirigido

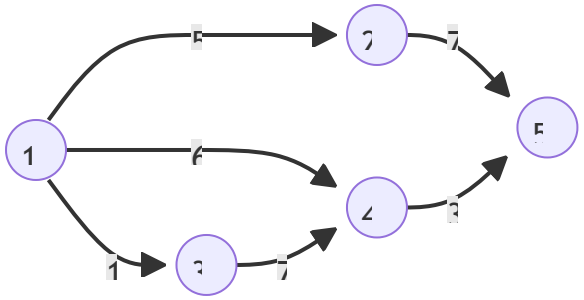
Un grafo es dirigido si las aristas solo pueden ser recorridas en una dirección. Por ejemplo, el siguiente grafo es dirigido:



El grafo anterior contiene una ruta 3 → 1 → 2 → 5 desde el nodo 3 al nodo 5, pero no hay ninguna ruta desde el nodo 5 al nodo 3.

Pesos de los aristas (weighted edges)

En un grafo ponderado, a cada arco se le asigna un peso. Los pesos a menudo se interpretan como longitudes de arcos. Por ejemplo, el siguiente grafo es ponderado:

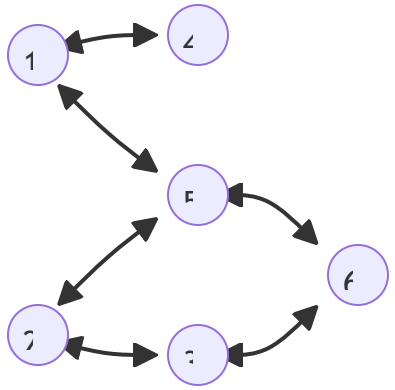


La longitud de un camino en un grafo ponderado es la suma de los pesos de los arcos en el camino. Por ejemplo, en el grafo anterior, la longitud del camino $1 \to 2 \to 5$. es 12, y la longitud del camino $1 \to 3 \to 4 \to 5$. es 11. Este último camino es el **camino más corto** desde el nodo 1 hasta el nodo 5.

Coloreo de grafos

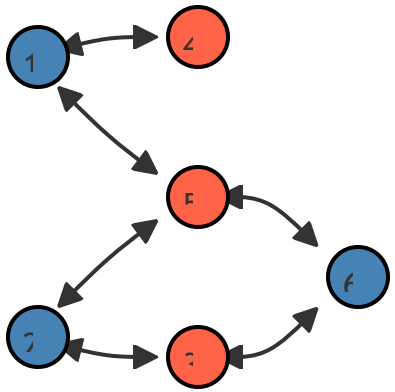
En un **coloreo** de un grafo, a cada nodo se le asigna un color de manera que ningún par de nodos adyacentes tengan el mismo color.

Un grafo es **bipartito** si es posible colorearlo usando dos colores. Resulta que un grafo es bipartito si y solo si no contiene un ciclo con un número impar de aristas. Por ejemplo, el grafo

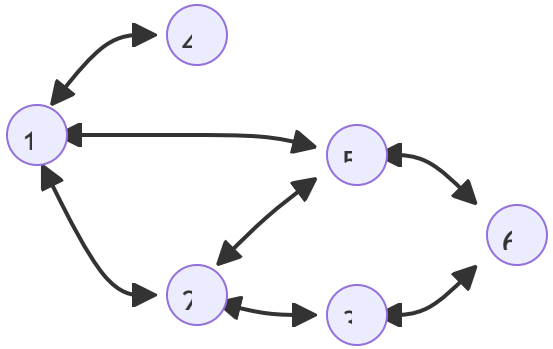


es bipartito, ya que se puede colorear de la siguiente forma:

- Nodos $1 \to 3 \to 5$. en color rojo.
- Nodos $2 \to 4 \to 6$. en color azul.



Sin embargo, el grafo



no es bipartito, ya que no es posible colorear el siguiente ciclo de tres nodos usando dos colores:

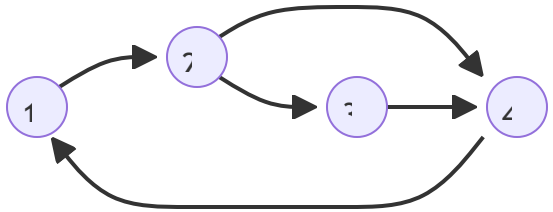
Representación de grafos con una lista de adyacencia

En la representación mediante listas de adyacencia, cada nodo `x` en el grafo tiene una lista de adyacencia que incluye los nodos a los que está conectado mediante una arista. Esta es una de las formas más populares de representar grafos, ya que permite implementar algoritmos de manera eficiente.

Para almacenar las listas de adyacencia, se puede declarar un arreglo de vectores así:

```
vector<int> adj[N];
```

Aquí, `N` es una constante que define el tamaño necesario para almacenar todas las listas de adyacencia. Por ejemplo, para el siguiente grafo:



se puede almacenar como:

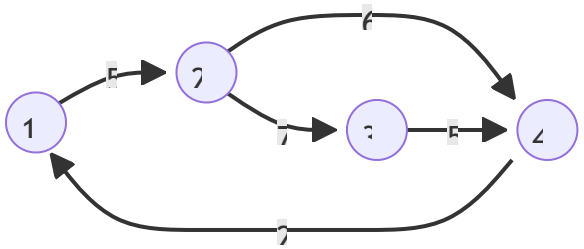
```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

> Si el grafo no está dirigido es similar, pero cada arista (edge) se agrega en ambas direcciones.

Para un grafo ponderado, la estructura es:

```
vector<par<int,int>> adj[N];
```

En este caso, la lista de adyacencia del nodo a contiene el par (b, w) siempre que exista una arista (edge) del nodo a al nodo b con peso w. Por ejemplo, el grafo:



se puede almacenar de la siguiente manera:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back(make_pair(1,2));//alternativa a {}
```

La ventaja de usar listas de adyacencia es que podemos encontrar de manera eficiente los nodos a los que podemos movernos desde un nodo dado a través de una arista. Por ejemplo, el siguiente `for` pasa por todos los nodos a los que podemos movernos desde el nodo s:

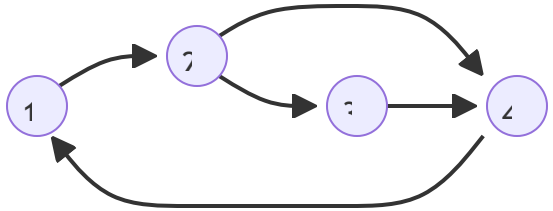
```
for (auto u : adj[s]) {
  // procesar nodo u
}
```

Representación de grafos con la matriz de adyacencia

Una **matriz de adyacencia** es un arreglo bidimensional que indica qué aristas están presentes en el grafo. Esta representación permite verificar de manera eficiente si existe una arista entre dos nodos.

```
int adj[N][N]; // declaracion simple de C
vector<vector<int>> adj(N, vector<int>(N, 0));//declaracion con std::vector de c++
```

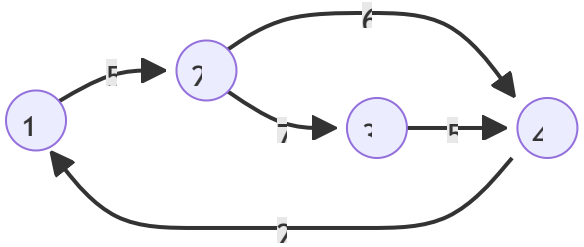
Cada valor `adj[a][b]` indica si existe una arista desde el nodo `a` al nodo `b`. Si la arista está en el grafo, entonces `adj[a][b] = 1`; si no, `adj[a][b] = 0`.



esta matriz se ve como la siguiente tabla:

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

y si el grafo es ponderado cambiamos los 1 por sus pesos



dicha matriz ponderada se ve como la siguiente tabla:

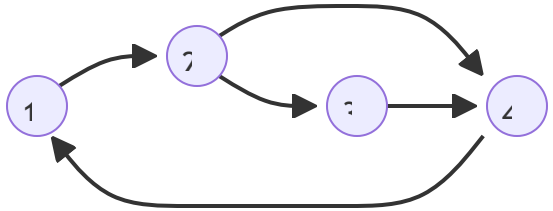
	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

> La desventaja de la representación de la matriz de adyacencia es que la matriz contiene n^2 elementos y la mayoría de ellos son cero. Por este motivo, la representación no se puede utilizar si el grafo es grande.

Representación de grafos con lista de aristas (Edge list)

An edge list contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node.

```
vector<pair<int,int>> edges;//aristas
```




```
edges.push_back({1,2});    edges.push_back({2,3});
edges.push_back({2,4});    edges.push_back({3,4});    edges.push_back({4,1});
```

Search (traverse) algorithms (for unweighted graphs)

Búsqueda en profundidad / Depth-first search (DFS)

La búsqueda en profundidad (DFS) es una técnica sencilla de recorrido de grafos. El algoritmo comienza en un nodo inicial y se extiende a todos los demás nodos accesibles desde ese nodo inicial, utilizando los bordes del grafo.

La búsqueda en profundidad sigue siempre un único camino en el grafo mientras encuentra nuevos nodos. Una vez que llega a un punto sin salida (sin más nodos nuevos), regresa a los nodos anteriores y comienza a explorar otras partes del grafo. El algoritmo mantiene un registro de los nodos visitados para procesar cada nodo solo una vez.

Características de DFS

- Utiliza el principio de “último en entrar, primero en salir” (pila).
- No es óptimo y tiende a ser errático.

Implementacion de DFS

```
vector<int> adj[N];
bool visited[N];

void dfs(int s) {
  if (visited[s]) return;
  visited[s] = true;
  // process node s
  for (auto u: adj[s])
    dfs(u);
}//O(V + E)  (Vertices + Edges)
```

Implementacion de DFS con contador de profundidad

```
void dfs(int s) {
    static int depth = 0;
    if (visited[s]) return;
    visited[s] = true;
    cout << "Node: " << s << ", depth: " << depth << endl;
    ++depth;
    for (auto u : adj[s])
        dfs(u);
    --depth;
}
```

Búsqueda en amplitud/ Breadth-first search (BFS)

La búsqueda en amplitud (BFS) visita los nodos en orden creciente de su distancia desde el nodo inicial. De esta manera, es posible calcular la distancia desde el nodo inicial hasta todos los demás nodos usando la búsqueda en amplitud. Sin embargo, la búsqueda en amplitud es más complicada de implementar que la búsqueda en profundidad.

La búsqueda en amplitud recorre los nodos nivel por nivel. Primero explora los nodos cuya distancia desde el nodo inicial es 1, luego los nodos a distancia 2, y así sucesivamente. Este proceso continúa hasta que se hayan visitado todos los nodos.

Características de BFS

- Utiliza el principio de “primero en entrar, primero en salir” (cola).
- Es óptima, pero lenta.

Implementacion de BFS

```
queue<int> q;
bool visited[N];
int distance[N];

visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
  int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
      if (visited[u]) continue;
      visited[u] = true;
      distance[u] = distance[s]+1;
      q.push(u);
  }
}//O(V + E)  (Vertices + Edges)
```

DFS vs BFS

Característica	DFS (Depth-First Search)	BFS (Breadth-First Search)
Orden de exploración	Profundidad primero, explora completamente una rama antes de retroceder	Anchura primero, explora todos los nodos en el nivel actual antes de pasar al siguiente nivel
Recorrido	Recursivo, sigue cada camino hasta su fin antes de retroceder	Utiliza una cola para recorrer nivel por nivel
Aplicaciones	Ideal para explorar caminos completos y laberintos, útil en algoritmos de backtracking	Ideal para encontrar la ruta más corta en gráficos no ponderados y niveles de árbol
Uso típico	Problemas de profundidad máxima, laberintos, problemas de conexión	Búsqueda de ruta más corta, recorrido por niveles en árboles
Ejemplo de uso	Cálculo de profundidad máxima en un árbol	Nivel de recorrido en árboles, distancia mínima en redes sin peso
Implementación	Recursiva o con pila explícita	Con una cola

Caminos más cortos (para grafos ponderados)

El algoritmo de Dijkstra encuentra los caminos más cortos desde el nodo inicial hasta todos los nodos del grafo, al igual que el algoritmo de Bellman–Ford. La ventaja del algoritmo de Dijkstra es que es más eficiente y se puede utilizar para procesar grafos grandes. Sin embargo, este algoritmo requiere que el grafo no contenga aristas con pesos negativos.

Al igual que el algoritmo de Bellman–Ford, el algoritmo de Dijkstra mantiene las distancias a los nodos y las reduce a medida que realiza la búsqueda. El algoritmo de Dijkstra es eficiente porque solo procesa cada arista del grafo una vez, aprovechando la condición de que no hay aristas de peso negativo.

Características del algoritmo de Dijkstra

- Explora en orden de costo creciente.
- Procesa cada arista una única vez (sin aristas de peso negativo).
- Es óptimo y adecuado para grafos grandes, aunque puede ser lento.

Implementacion de Dijkstra para una de lista de adyacencia

```
const int INF = numeric_limits<int>::max();
vector<int> distance(n + 1, INF);
vector<bool> processed(n + 1, false);
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
distance[x] = 0;
q.push({0, x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a] + w < distance[b]) {
            distance[b] = distance[a] + w;
            q.push({distance[b], b});
        }
    }
}//O(V^2)
```

Dijkstra's algorithm in 3 minutes

Problema

- **1360E** Polygon ⮭

Referencias

- GeeksforGeeks. (2024). *Introduction to Graph Data Structure*. Recuperado de https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/
- Huang, S. et al (s.f.). *Graph Traversal*. Recuperado de https://usaco.guide/silver/graph-traversal?lang=cpp
- Laaksonen, A. (2018). *Competitive Programmer's Handbook*. Recuperado de https://cses.fi/book/book.pdf
- Mussmann, S. & See, A. (s.f.). *Graph Search Algorithms*. Recuperado de https://cs.stanford.edu/people/abisee/gs.pdf
- Yao, D. & Qi, B. (s.f.). *Introduction to Graphs*. Recuperado de https://usaco.guide/bronze/intro-graphs