Grafos

Universidad Nacional de Rosario

Taller de Programación Competitiva

Contenidos

Qué es un algoritmo computacionalmente hablando?

A qué le llamamos algoritmo?

Un algoritmo es una serie de pasos (o instrucciones) a seguir. En computación le decimos algoritmo a los pasos que sigue una computadora al ejecutar un programa.

Le llamamos algoritmo a la idea que usamos para escribir el código y no al conjunto de instrucciones. Por ejemplo, un algoritmo para ver si un número es primo es dividirlo por los números menores o iguales a su raiz cuadrada y ver si el resto es cero, y no importa cómo lo implementemos es siempre el mismo algoritmo.

Complejidad de un algoritmo

Para decidir si un algoritmo es “bueno” o “malo” (podemos entender por bueno que sea eficiente) vamos a medir su complejidad.

Complejidad

La complejidad de un algoritmo mide la utilización de los recursos disponibles. Los recursos más importantes con los que cuenta una computadora a la hora de ejecutar un programa son dos:

- Memoria: La máxima cantidad de memoria que usa el programa al mismo tiempo. Si usa varias veces la misma memoria se cuenta una sóla vez.
- Tiempo: Cuánto tarda en correr el algoritmo. Se mide en cantidad de operaciones básicas (sumas, restas, asignaciones, etc)

Como medimos el uso de la memoria?

El uso de la memoria

El uso de la memoria lo medimos por la mayor cantidad de memoria que usa un programa al mismo tiempo. Por ejemplo, si el programa utiliza 200MB de memoria, libera 50MB y ocupa otros 150MB, la memoria usada es primero 200MB, después 150MB y por último 300MB. Por más que el programa utilize en un momento 200MB y luego ocupe otros 150MB, la memoria que utiliza el programa es 300MB y no 350MB ya que nunca utiliza 350MB al mismo tiempo.

En contexto de competencias depende del problema y la competencia pero por lo general el límite de memoria disponible es entre 1GB y 2GB.

Como medimos el uso del tiempo?

El uso del tiempo

El uso del tiempo lo medimos por la cantidad de operaciones básicas que ejecuta el programa, como pueden ser sumas, restas, asignaciones, etc. Es muy difícil calcular este número y por lo general nos interesa más tener una idea de cómo crece esta complejidad a medida que crece la entrada del problema, por eso hablamos de órdenes de complejidad.

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.

Complejidades

- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.
- El cálculo de complejidades es un análisis teórico y no tiene en cuenta la constante c que puede ser muy chica o muy grande, sin embargo es por lo general una buena referencia para saber si un algoritmo es bueno o malo.

Definición

Grafo (Wikipedia)

Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (edges en inglés) que pueden ser orientados (dirigidos) o no.

Grafo (RAE)

Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio.

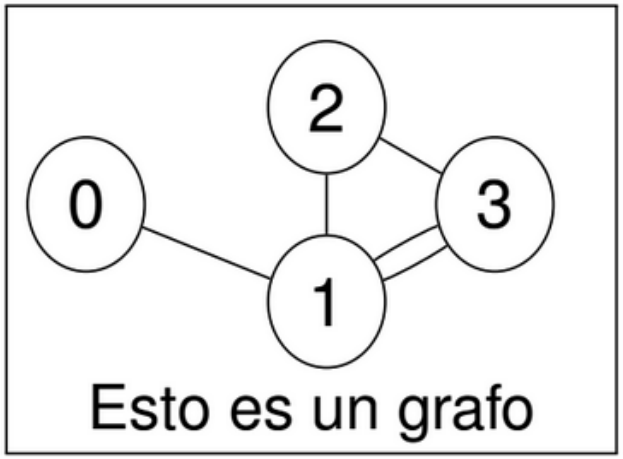
Grafo (Nuestra Definición)

Un grafo es un conjunto de puntos y líneas que unen pares de esos puntitos

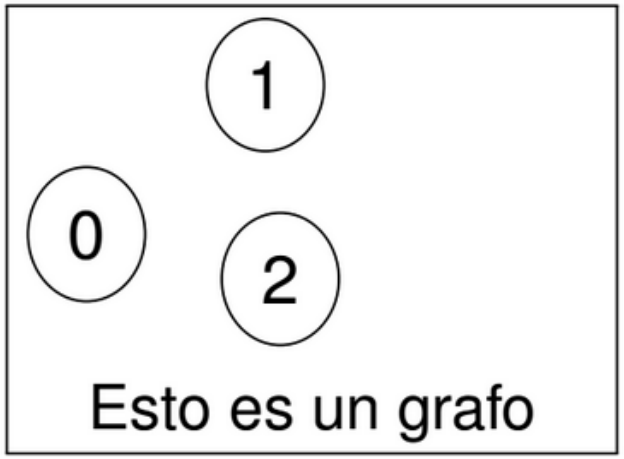
Definición (cont.)

Grafo

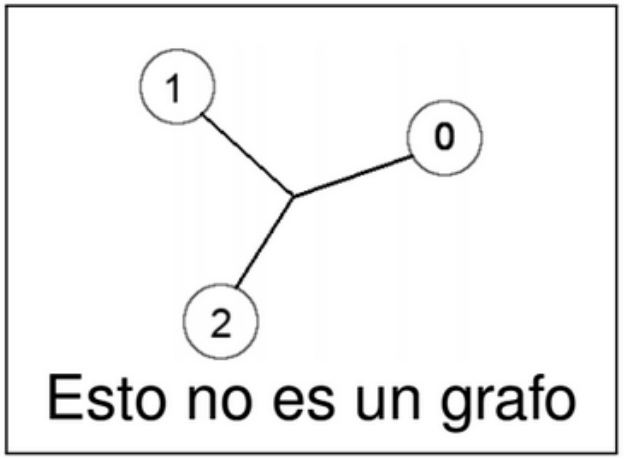
Un grafo es un conjunto de puntos y líneas que unen pares de esos puntitos



Esto es un grafo



Esto es un grafo



Esto no es un grafo

- Los grafos pueden ser **Dirigidos** o **No Dirigidos**, como en este caso.

Grafo: Problemas típicos

Ejemplo 1

Un país tiene N ciudades conectadas por M calles de diferente largo.
¿Cuál es el camino más corto desde la ciudad a hasta la ciudad b?

Ejemplo 2

Beto y Carlos tienen un árbol genealógico de sus familias.
¿Cúal es su ancestro común menor?

Entrada Típica

Como se dan los grafos en un problema tipico

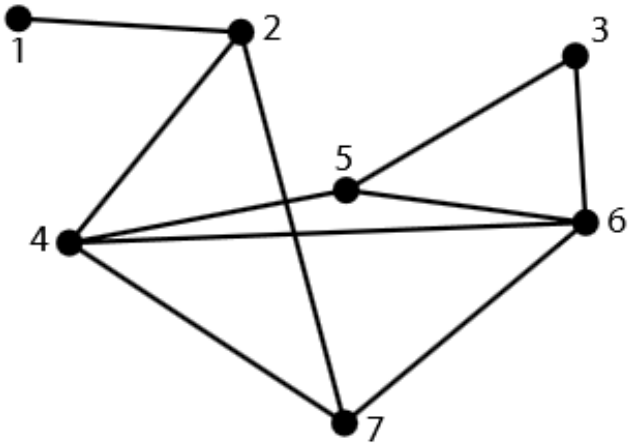
Se describe el grafo con un número N (cantidad de vértices) y M (cantidad de aristas). Luego hay M líneas donde se listan pares a b (que indica que hay una arista $a \to b$). Los vértices se los numera del 0 al $N - 1$.

Contenidos

Matriz de adyacencia

Dado un grafo G cuyos vértices están numerados de 1 a n, definimos la matriz de adyacencia de G como $M \in \{0,1\}^{n \times n}$ donde $M(i,j) = 1$ si los vértices i y j son adyacentes y 0 en otro caso.

Ej:



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Como representar grafos en memoria

Matriz de adyacencia

Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.

Ventajas

Saber si dos nodos están conectados tiene complejidad O(1) Se pueden hacer operaciones sobre esta matriz para encontrar propiedades del grafo.

Como representar grafos en memoria

Desventajas

Buscar los nodos adyacentes a otro nodo tiene complejidad O(|V|), sin importar cuantos nodos adyacentes tenga el primer nodo. Recorrer todo el grafo tiene complejidad O($|V|^2$) La matriz tiene una complejidad en memoria de O($|V|^2$) No hay ninguna manera obvia de representar multiejes (pares de nodos)

Código

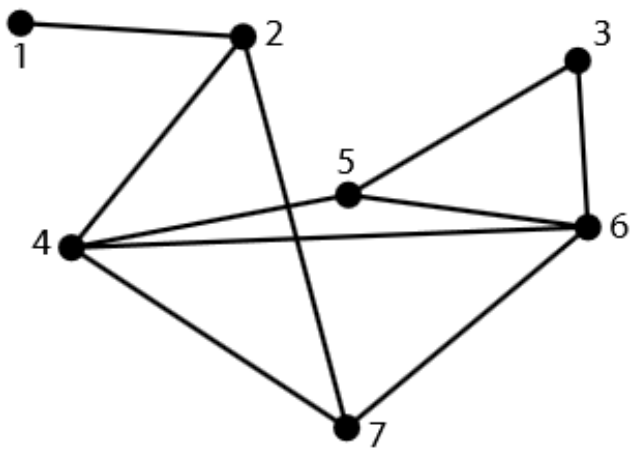
```
1
2  int N,M;
3  bool mat[1000][1000];
4  cin >> N >> M;
5  memset(mat,0,sizeof(mat))
6  for(int i=0;i<M;i++) {
7      int a, b; cin >> a >> b;
8      mat[a][b]=true;
9  }
```

Contenidos

Lista de adyacencia

Coloquialmente la llamamos *lista de vecinos* pues para cada nodo guardamos la lista de nodos para los que existe una arista que los conecta (o sea, los vecinos).

Ej:



$$L_1 : 2$$
$$L_2 : 1 \longrightarrow 4 \longrightarrow 7$$
$$L_3 : 5 \longrightarrow 6$$
$$L_4 : 2 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7$$
$$L_5 : 3 \longrightarrow 4 \longrightarrow 6$$
$$L_6 : 3 \longrightarrow 4 \longrightarrow 5 \longrightarrow 7$$
$$L_7 : 2 \longrightarrow 4 \longrightarrow 6$$

Lista de adyacencia (cont.)

Lista de Adyacencia

Una lista con |V| elementos, donde el elemento número i tiene la lista de nodos adyacentes al nodo i.

Ventajas

Tiene O(|E|) de complejidad de memoria. Encontrar la cantidad de nodos adyacentes a cierto nodo tiene complejidad lineal en el tamaño de la respuesta. Recorrer todo el grafo tiene complejidad O(|V| + |E|).

Desventajas

Saber si dos nodos son vecinos es lineal al grado de alguno de los dos nodos, que en el peor caso es O(|V|).

Representando una lista de Adyacencia en C++

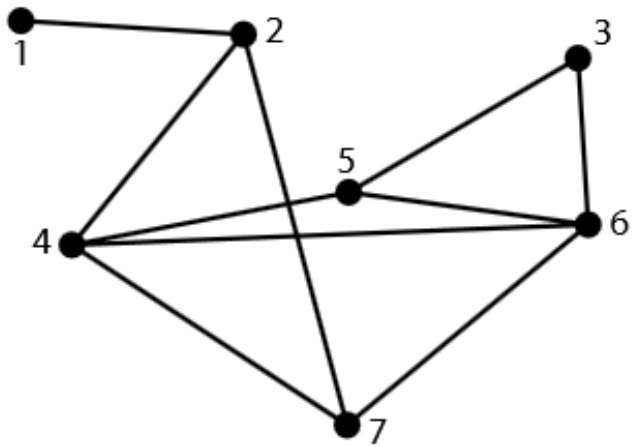
Código

```
1 int N,M;
2 vector<int> G[1000];
3 cin >> N >> M;
4 for(int i=0;i<M;i++) {
5     int a, b; cin >> a >> b;
6     G[a].push_back(b);
7 }
```

Contenidos

Lista de incidencia

Ej:



Cant. vértices: 7

Aristas: (1,2),(2,4),(2,7),(3,5),
(3,6),(4,5),(4,6),(4,7),(5,6),(6,7)

Es una estructura útil para el algoritmo de Kruskal, por ejemplo. También es la que se usa en general para almacenar un grafo en un archivo de texto.

Recorriendo un grafo

Muchas veces la solución de un problema requiere un algoritmo que recorra los nodos de un grafo en algún orden el particular.

Distancia

Comenzaremos trabajando con grafos no dirigidos y daremos algunas definiciones que tienen sentido en grafos no dirigidos, pero que luego podremos adaptar a grafos dirigidos.

- Decimos que dos vértices v_1 y v_2 son adyacentes si $(v_1, v_2) \in E$. En este caso decimos que hay una arista entre v_1 y v_2.
- Un camino de largo n entre v y w es una lista de $n+1$ vértices $v = v_0, v_1, \ldots, v_n = w$ tales que para $0 \leq i < n$ los vértices v_i y $v_i + 1$ son adyacentes. La distancia entre dos vértices v y w se define como el menor número n tal que existe un camino entre v y w de largo n. Si no existe ningún camino entre v y w decimos que la distancia entre v y w es ∞

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.
- Un problema muy frecuente es tener que encontrar la distancia entre dos vértices, este problema se resuelve encontrando un camino mínimo entre los vértices.
- Este problema es uno de los problemas más comunes de la teoría de grafos y se puede resolver de varias maneras, una de ellas es el algoritmo llamado BFS (Breadth First Search).

Contenidos

BFS

Breadth First Search

El BFS es un algoritmo que calcula las distancias de un nodo de un grafo a todos los demás. Para esto empieza en el nodo desde el cual queremos calcular la distancia a todos los demás y se mueve a todos sus vecinos, una vez que hizo esto, se mueve a los vecinos de los vecinos, y así hasta que recorrió todos los nodos del grafo a los que existe un camino desde el nodo inicial.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)
- Usaremos una cola para ir agregando los nodos por visitar. Como agregamos primero todos los vecinos del nodo inicial, los primeros nodos en entrar a la cola son los de distancia 1, luego agregamos los vecinos de esos nodos, que son los de distancia 2, y así vamos recorriendo el grafo en orden de distancia al vértice inicial.
- Cuando visitamos un nodo, sabemos cuáles de sus vecinos agregar a la cola. Tenemos que visitar los vecinos que todavía no han sido visitados.

Código de ejemplo del BFS

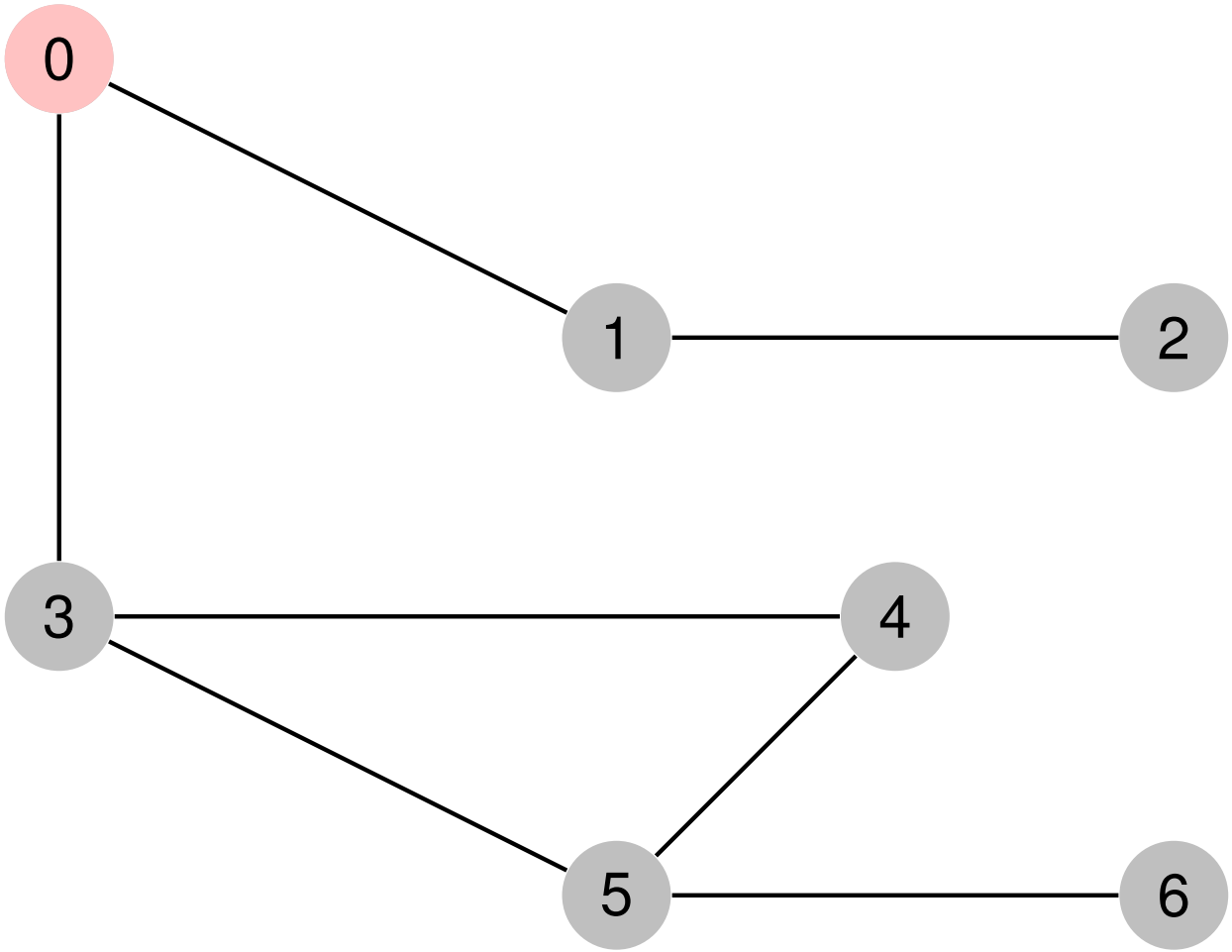
```
1  vector<int> G[1000];
2  int n;
3  int dist[1000];
4
5  void bfs(int root){
6    queue<int> q;
7    memset(dist,-1,sizeof(dist));
8    q.push(root); dist[root]=0;
9
10   dist[root] = 0;
11   while(!q.empty()) {
12     int v = q.front(); q.pop();
13     for(vector<int>::iterator it = G[v].begin() ; it!=G[v].end() ; ++it)
14       if (d[*it]==-1) {
15         d[*it]=d[v]+1;
16         q.push(*it);
17       }
18   }
19 }
```

Formas de recorrer un grafo

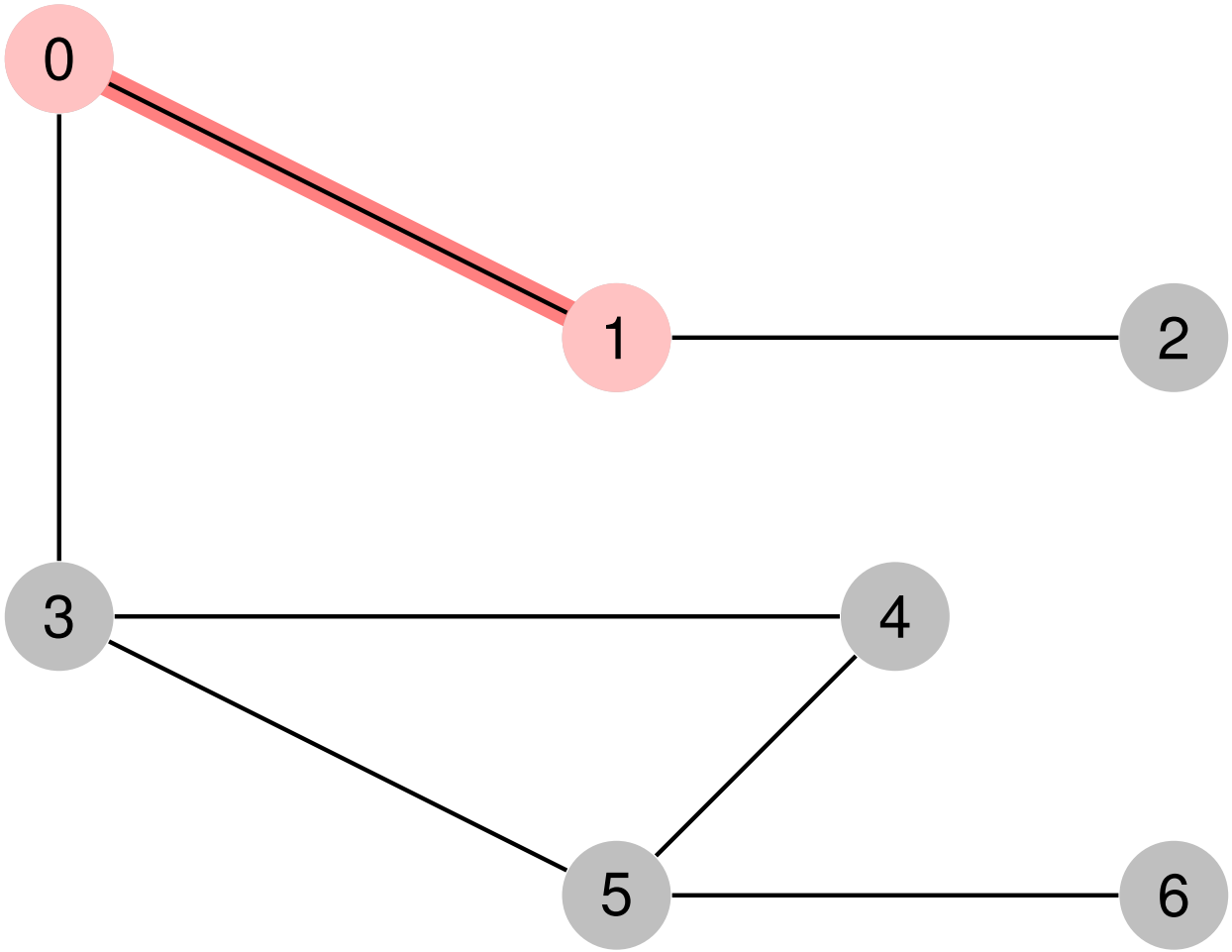
- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. El BFS recorre los nodos en orden de distancia a un nodo.
- Otra forma de recorrer un grafo (sustancialmente más sencilla) es un DFS (Depth First Search), que recorre el grafo en profundidad, es decir, empieza por el nodo inicial y en cada paso visita un nodo no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.

Contenidos

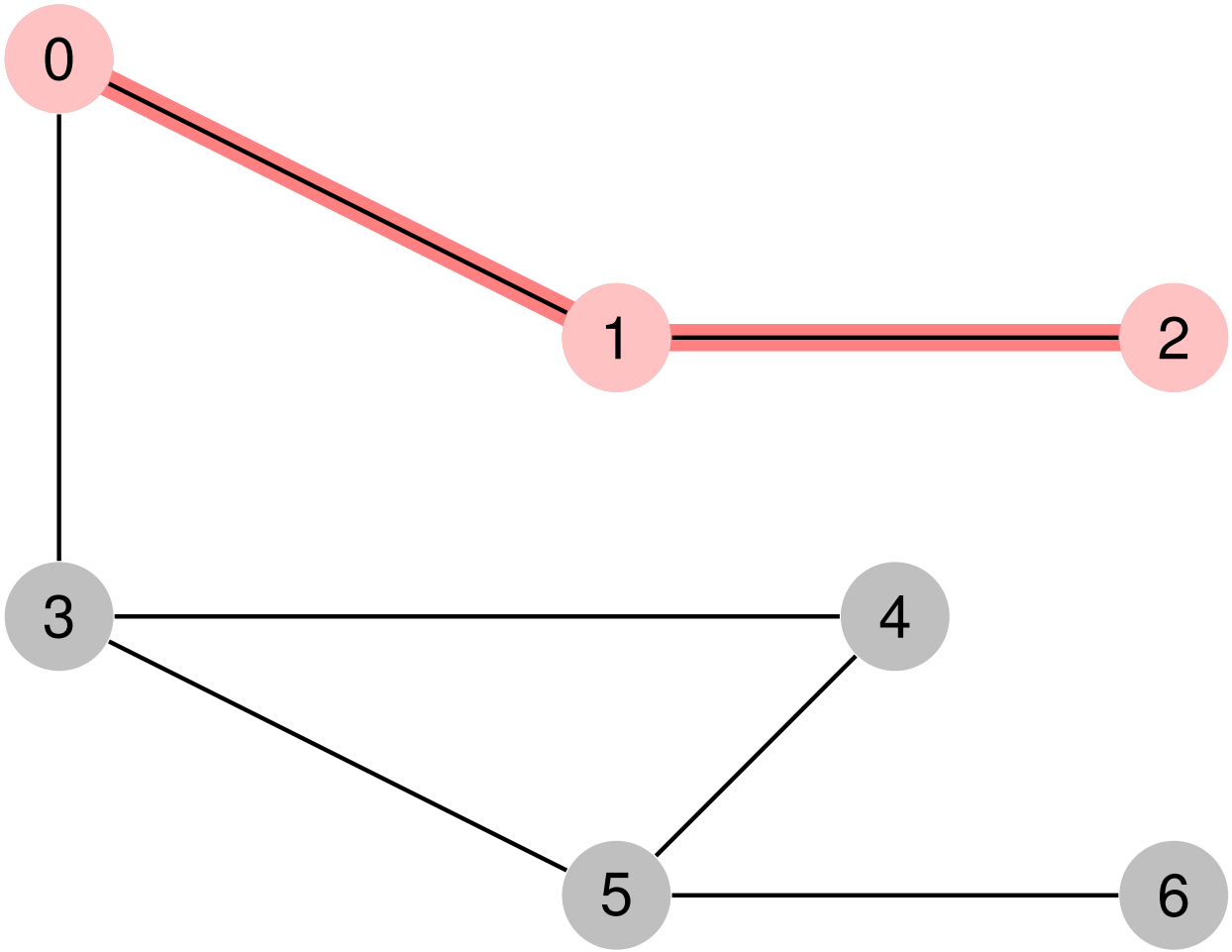
Ejemplo Gráfico del recorrido de un DFS



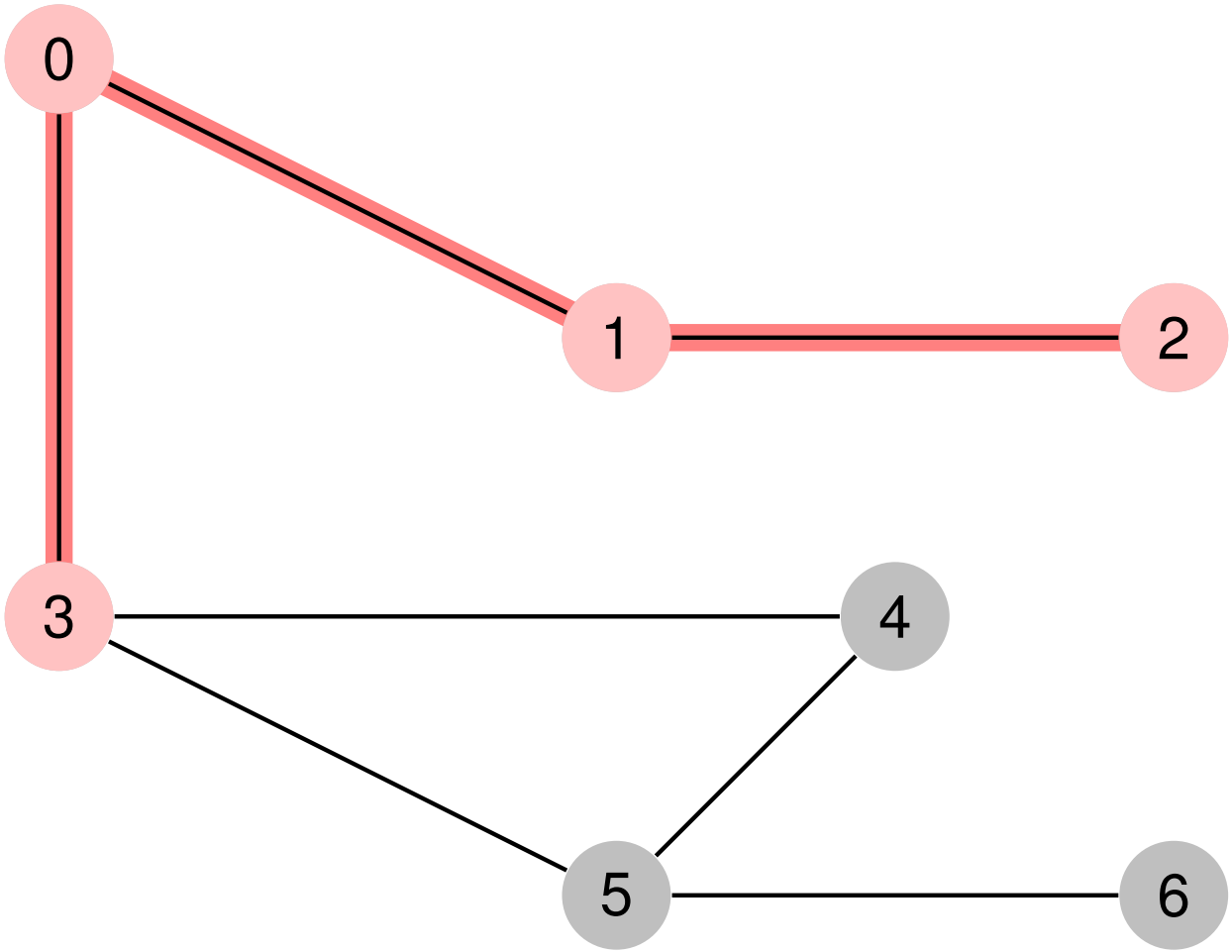
Ejemplo Gráfico del recorrido de un DFS



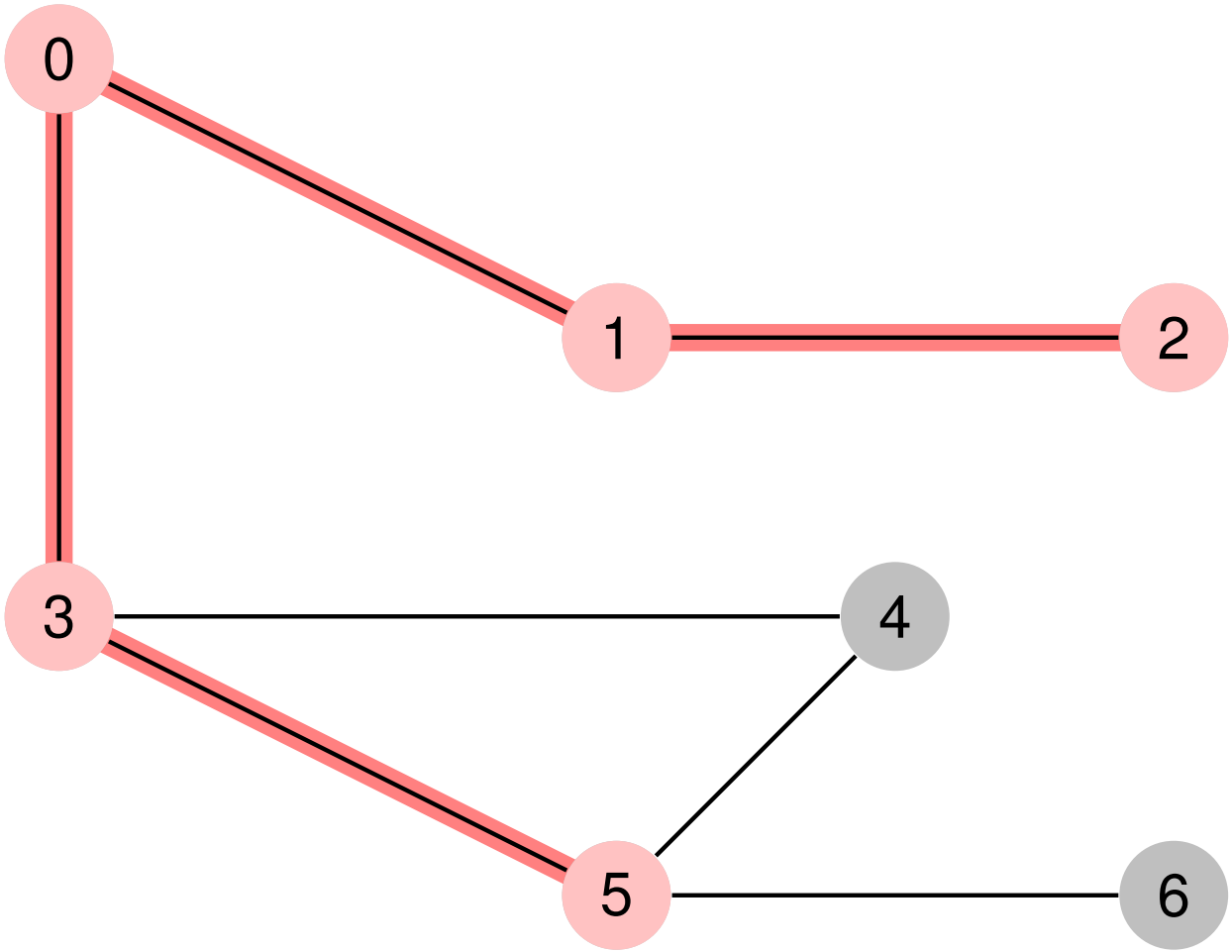
Ejemplo Gráfico del recorrido de un DFS



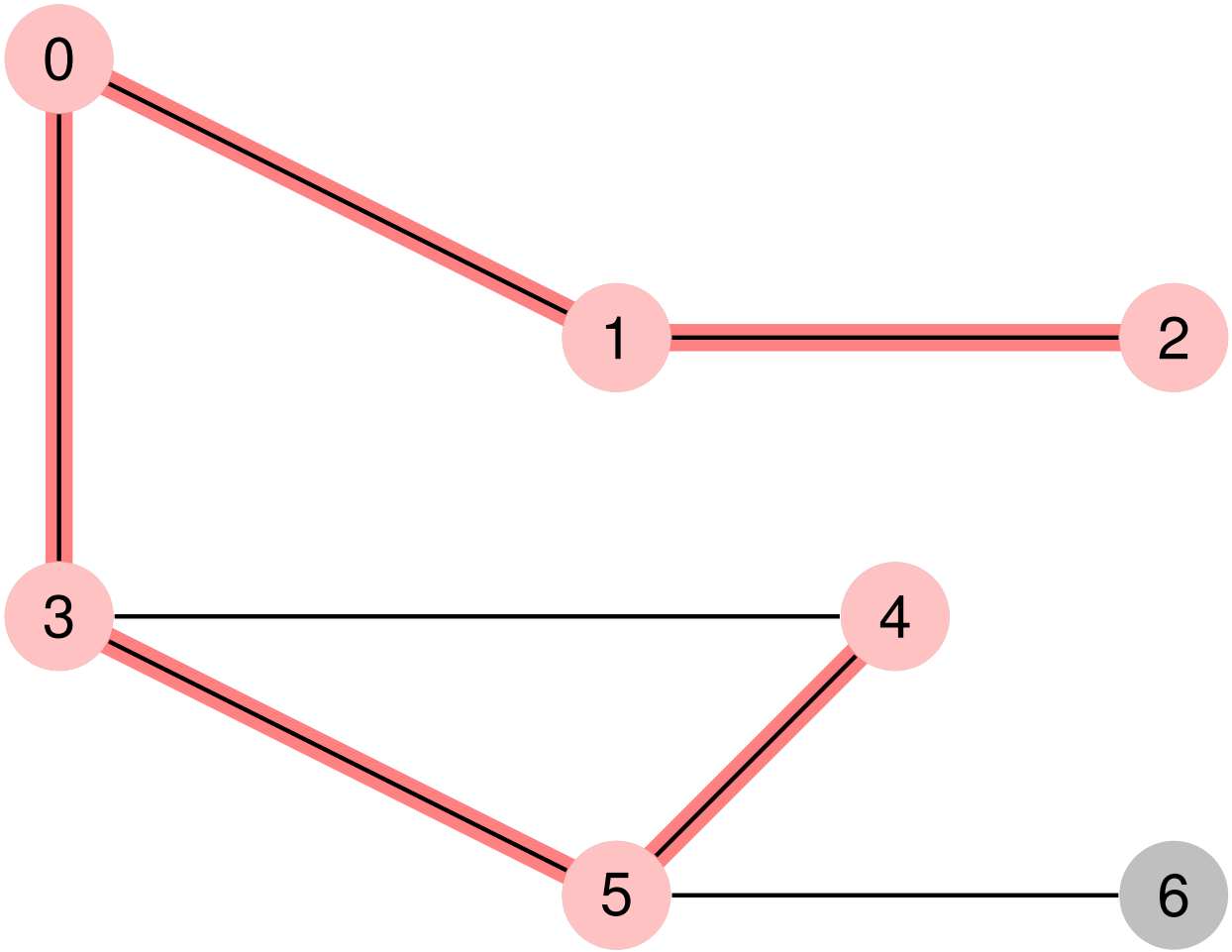
Ejemplo Gráfico del recorrido de un DFS



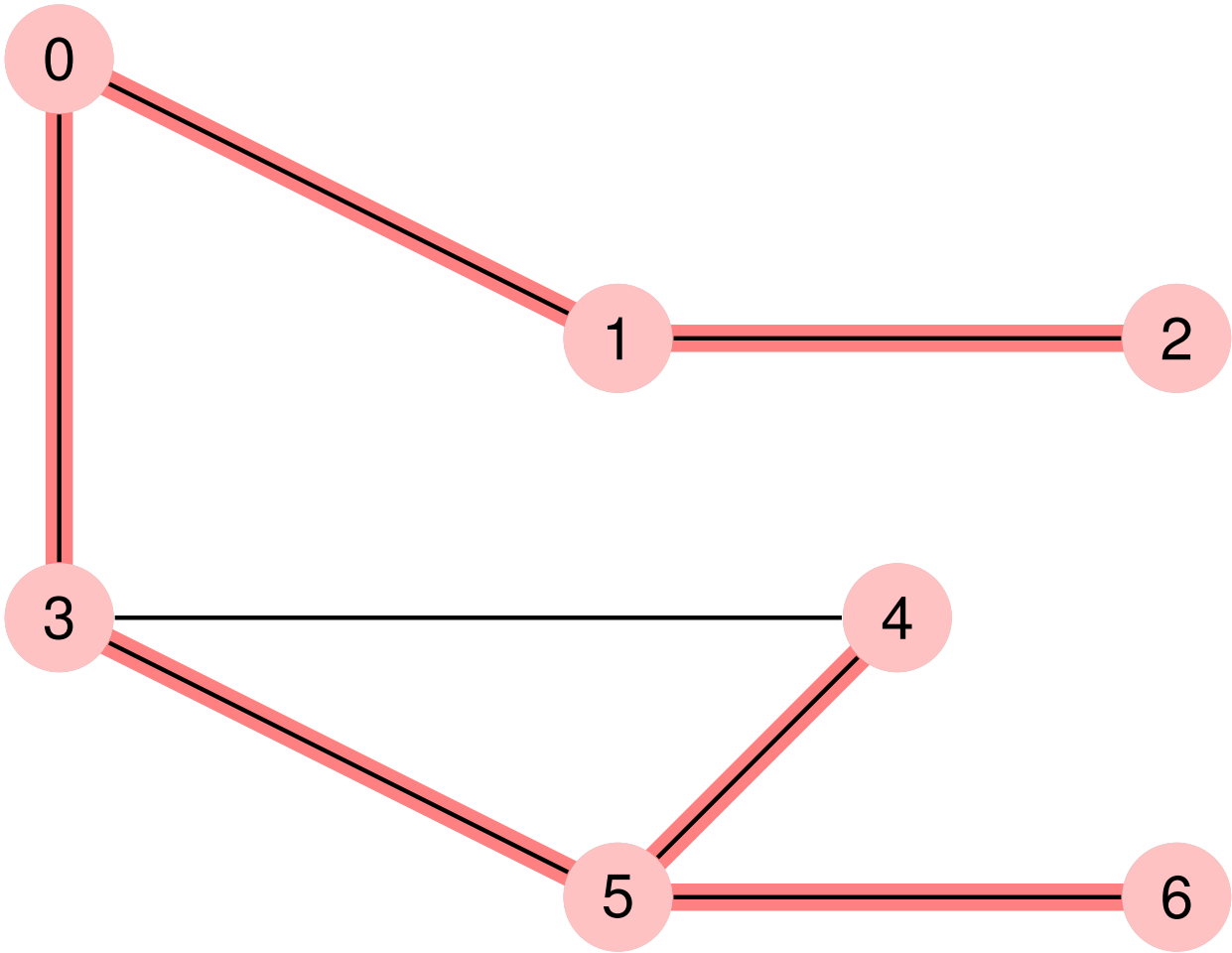
Ejemplo Gráfico del recorrido de un DFS



Ejemplo Gráfico del recorrido de un DFS



Ejemplo Gráfico del recorrido de un DFS



Ejemplo

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.
- Cuando llega a un nodo que ya fue visitado se da cuenta de que pudo acceder a ese nodo por dos caminos, eso quiere decir que si recorremos uno de los caminos en un sentido y el otro camino en sentido opuesto formamos un ciclo.
- Así como el BFS se implementa con una cola, el DFS se implementa con una pila.

Código de un DFS

Pseudocódigo

```
visitar(v):
    marcar v como visitado
    por cada vecino u no visitado de v:
            visitar(u)
```

Código ejemplo de DFS

```
vector<int> G[1000];
int n;
bool vis[1000];
void dfs(int v) {
    vis[v]=true;
    for(vector<int>::iterator it = G[v].begin(); it!=G[v].end(); ++it)
      if (!vis[*it]) dfs(*it);
}
...
memset(vis,0,sizeof(vis))
for(int i=0;i<n;i++) if(!vis[i]) dfs(i);
```

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Para detectar un ciclo podemos guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.
- La forma de chequear si un grafo es conexo es, empezando por algún nodo, chequear que hayamos tocado todos los nodos, es decir, que todas las posiciones de *vis* terminen en true.
- La complejidad del BFS y del DFS es $O(n+m) = O(m)$

Preguntas??