Grafos: una no tan breve introducción

Licenciada Melanie Sclar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Nacional OIA 2015

¿Qué es un grafo?

> “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.”
>
> — Wikipedia

¿Qué es un grafo?

> “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.”
>
> — Wikipedia

> “Un grafo es un conjunto de puntos y líneas que unen pares de esos puntos”
>
> — La Posta

Grafos dirigidos y no dirigidos

En los grafos no dirigidos las aristas son doble mano (se puede ir en ambos sentidos).

En los dirigidos en cambio, las aristas se recorren en un único sentido: desde el origen al destino de la flecha. Si queremos representar que la calle que une las esquinas u y v es doble mano deberemos poner dos aristas: una que vaya de u a v y otra que vaya de v a u.

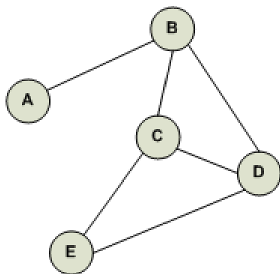


Fig 1. Grafo no dirigido

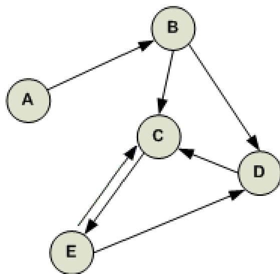


Fig 2. Grafo dirigido

¿Para qué podemos usar los grafos?

Mediante un grafo podemos representar, por ejemplo, una ciudad. Las esquinas serían los vértices y las conexiones por medio de una calle entre dos esquinas serían los ejes. Si todas las calles son doble mano, el grafo es no dirigido. Si algunas calles son mano única el grafo es dirigido: ¿cómo modelamos una calle doble mano aquí?

A medida que los problemas se dificultan, puede suceder que sea difícil que a uno se le ocurra modelar el problema con un grafo, pero que una vez que lo hayamos hecho, el problema se vuelva sencillo utilizando los algoritmos que veremos hoy.

De ahora en más trabajaremos con grafos no dirigidos por simplicidad, pero también podrían aplicarse a grafos dirigidos.

Vecindad

Diremos que dos nodos son *adyacentes* o *vecinos* si existe una arista que los une.

Distancia

La distancia entre dos nodos u y v es la mínima cantidad de ejes por los que me tengo que mover para salir de u y llegar a v. Si hay múltiples formas de llegar de u a v, me quedo con la más corta.

Formas de representar un grafo

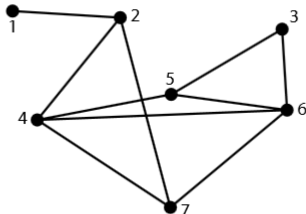
Existen varias maneras de guardar un grafo en memoria para poder luego consultar cosas (por ejemplo, recorrer el grafo, que es lo que haremos hoy).
Veamos las dos más populares.

Matriz de adyacencia

Matriz de adyacencia

La matriz de adyacencia es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j y 0 (o false) si no.

Ej:



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

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

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

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

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

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

Desventajas

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

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

Desventajas

- La complejidad espacial: se necesitan n^2 casillas para representar un grafo de n nodos.

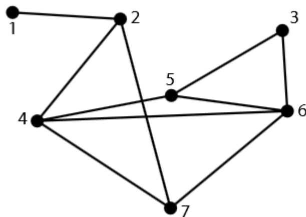
Lista de adyacencia

Lista de adyacencia

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j.

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:



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

Lista de adyacencia

Nuevamente, con la misma idea también se pueden modelar grafos dirigidos y con pesos.

La complejidad espacial de esta representación será posiblemente mucho menor. ¿Cuánta memoria necesitaremos para un grafo de n nodos y m aristas?

Lista de adyacencia

Nuevamente, con la misma idea también se pueden modelar grafos dirigidos y con pesos.

La complejidad espacial de esta representación será posiblemente mucho menor. ¿Cuánta memoria necesitaremos para un grafo de n nodos y m aristas? **O(m+n)**

Recorrer un grafo

A continuación vamos a ver dos algoritmos utilizados para recorrer grafos. Luego, podremos utilizar estos algoritmos para calcular lo que necesitemos (por ejemplo, distancias a un nodo en particular) o para encontrar un nodo en particular, que tenga una propiedad.

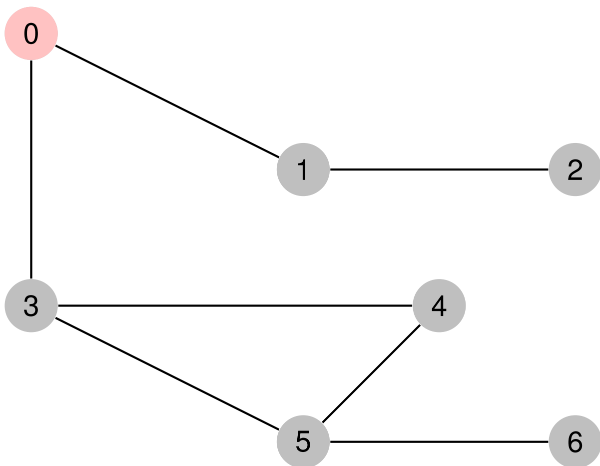
Ambos algoritmos tendrán una complejidad de $O(n+m)$, si n es la cantidad de nodos y m la de aristas, pero cada uno será útil en situaciones específicas.

Una forma intuitiva de recorrer el grafo: DFS

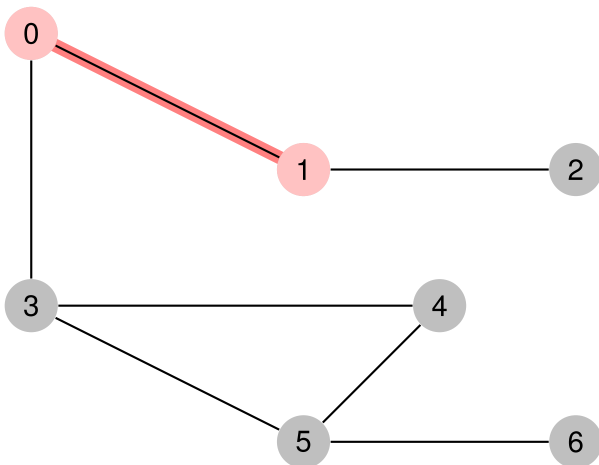
El DFS (Depth First Search) es un tipo de recorrido del grafo. Se dice que lo recorre *en profundidad*, es decir, empieza por el nodo inicial y en cada paso visita un nodo vecino no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.

Veamos un ejemplo visual de cómo se recorre un grafo con DFS.

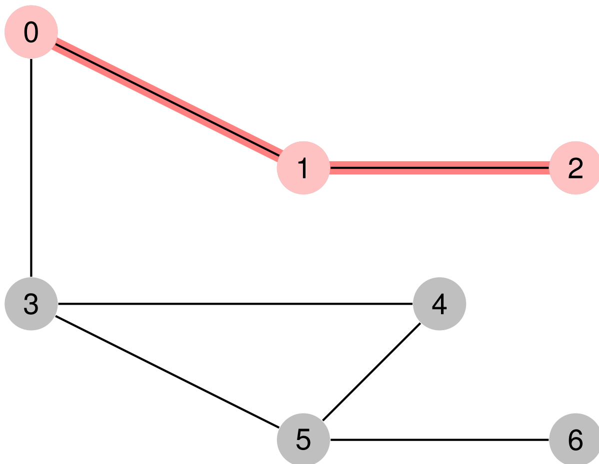
Ejemplo



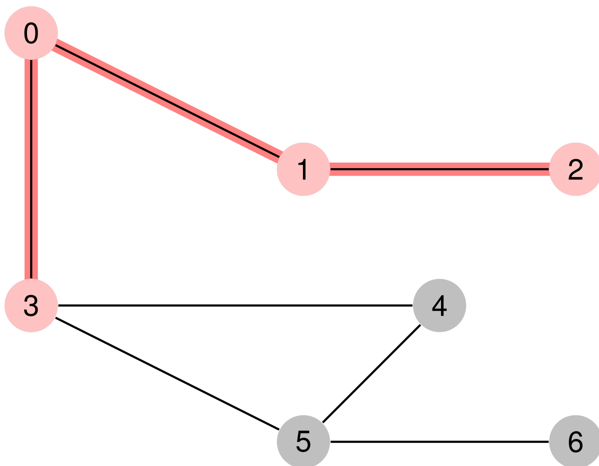
Ejemplo



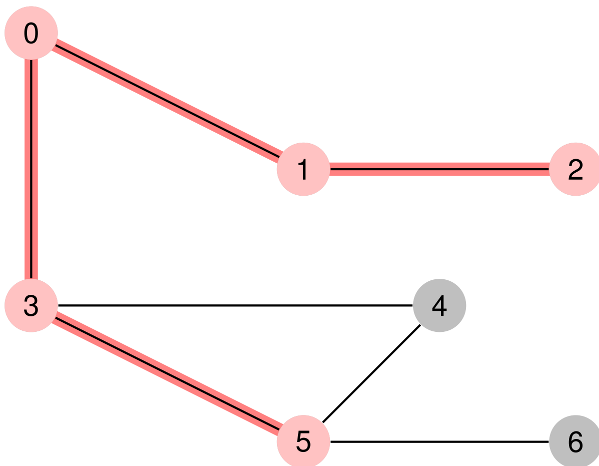
Ejemplo



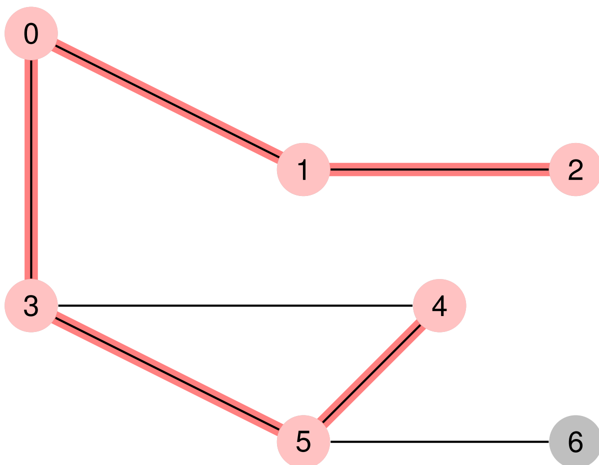
Ejemplo



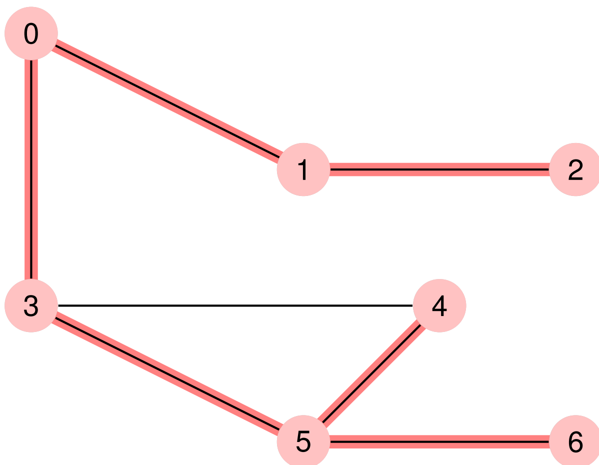
Ejemplo



Ejemplo



Ejemplo



Ejemplo

En el ejemplo anterior vimos cómo recorre el grafo un DFS. Ahora tratemos de implementar un DFS resolviendo un problema (por supuesto, DFS se puede aplicar a muchos problemas más).

Problema

Dado un grafo conexo (es decir, que existe al menos un camino entre todo par de nodos) queremos ver si dicho grafo es un **árbol**.

¿Pero qué significa que un grafo sea un **árbol**?

Árboles

Un árbol es un tipo especial de grafo no dirigido. Es un grafo donde entre cualquier par de nodos existe un único camino que los conecta (notar que en particular, todos los nodos tienen que estar conectados entre sí).

Los árboles tienen mil propiedades interesantes, como $m = n - 1$, que no tiene ciclos y más cosas sobre las que hoy no hablaremos.

Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?

Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?
- Iremos recorriendo el grafo con DFS, siempre moviéndome a un vecino no visitado aún. Para no visitar un nodo dos veces (eso sería innecesario y empeoraría la performance) marcamos cada nodo cuando lo revisamos. ¿Qué significa si llego a un nodo y un vecino de él (que no es por el que llegué) ya está visitado?

Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?
- Iremos recorriendo el grafo con DFS, siempre moviéndome a un vecino no visitado aún. Para no visitar un nodo dos veces (eso sería innecesario y empeoraría la performance) marcamos cada nodo cuando lo revisamos. ¿Qué significa si llego a un nodo y un vecino de él (que no es por el que llegué) ya está visitado?
- Significa que desde mi nodo inicial pude llegar a un nodo de dos formas distintas: es decir que no existe un único camino, lo que contradice la definición de árbol. Ergo, no es árbol.

Pseudocódigo del DFS

```
1  // inicialmente se llena el vector visitado en false pues ningun nodo fue
       visitado
2  // ademas el nodo inicial no tiene padre y por eso se llama con padre = −1
3  bool esArbol(grafo, int nodoActual, vector<bool> &visitado, int padre):
4      visitado[nodoActual] = true
5
6      para cada vecino v de nodoActual en grafo:
7          si v ya fue visitado y v != padre:
8              devolver false
9              // encontre dos caminos distintos y luego no es arbol
10         si v no fue visitado aun:
11             si no esArbol(grafo, v, visitado, nodoActual):
12             // llamo recursivamente al dfs con el subarbol con raiz en v,
                   ahora el padre es nodoActual
13                 devolver false
14
15     // si para ningun vecino encontre algun ciclo, es un arbol porque se que
           es conexo
16     devolver true
```

Implementación en C++ del DFS

```
1  bool esArbol(vector<vector<int> > &lista, int t, vector<bool> &visitado, int
       padre)
2  {
3      visitado[t] = true;
4      for(int i = 0; i < lista[t].size(); i++)
5      {
6          if(visitado[lista[t][i]] == true && lista[t][i] != padre)
7              return false;
8          if(visitado[lista[t][i]] == false)
9              if(esArbol(lista, lista[t][i], visitado, t) == false)
10                 return false;
11     }
12     return true;
13 }
```

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cual fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cual 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.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cual 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.
- Si el nodo no lo visitamos, pero desde uno de sus vecinos podemos llegar a un ciclo, entonces es porque hay un ciclo en el grafo y por lo tanto no es un árbol.

Otro algoritmo de búsqueda: BFS

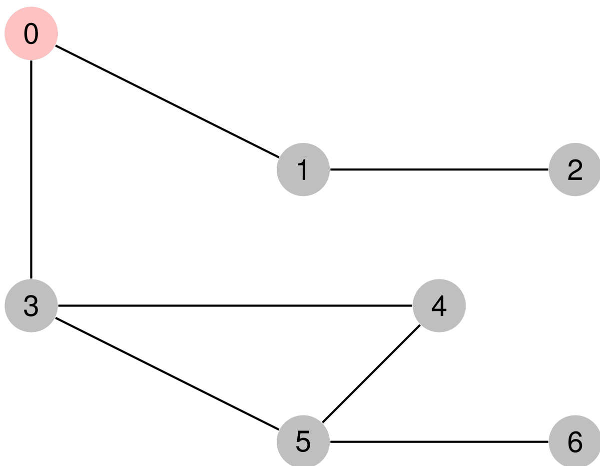
Muchas veces no basta con recorrer el grafo, sino que queremos hacerlo de una forma en particular. BFS suele ser muy útil en muchos problemas, veamos de qué se trata!

Breadth First Search o *búsqueda en anchura*

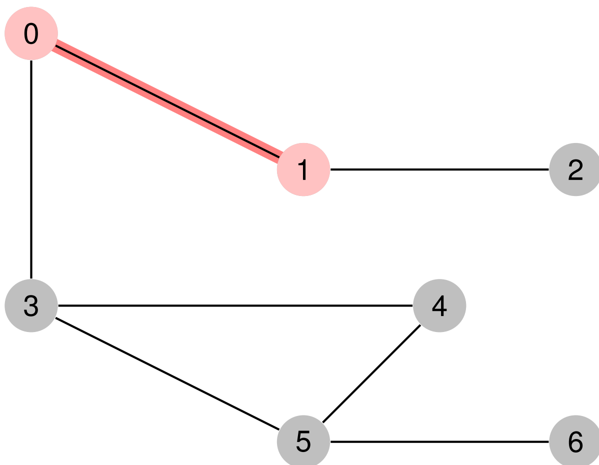
El Breadth First Search o *búsqueda en anchura* es un algoritmo que recorre un grafo comenzando por un nodo en particular, y se exploran todos los vecinos del nodo. Luego, se exploran todos los vecinos de los vecinos del nodo inicial, y así siguiendo hasta recorrer todos los nodos.

Veámoslo gráficamente.

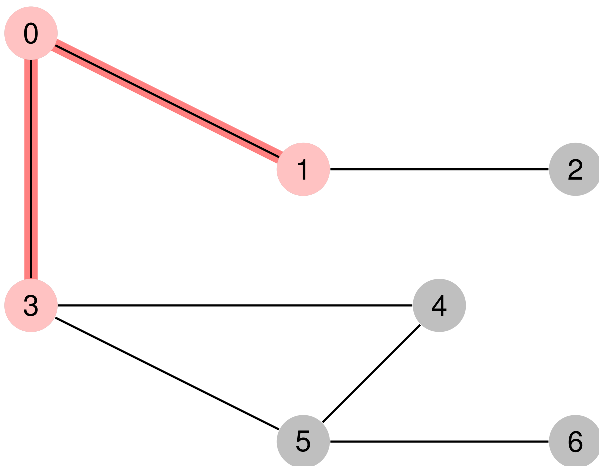
Ejemplo de recorrido de un grafo usando BFS



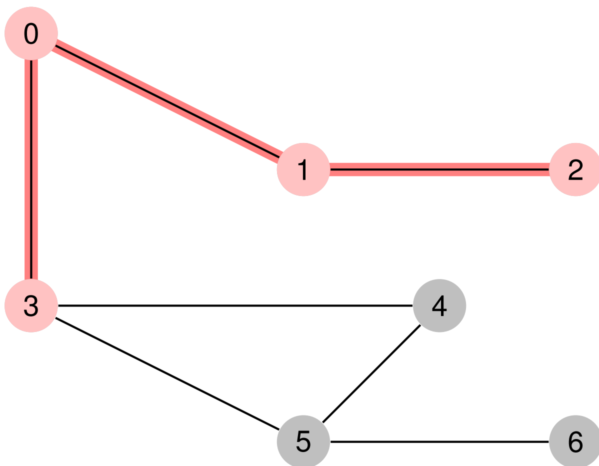
Ejemplo de recorrido de un grafo usando BFS



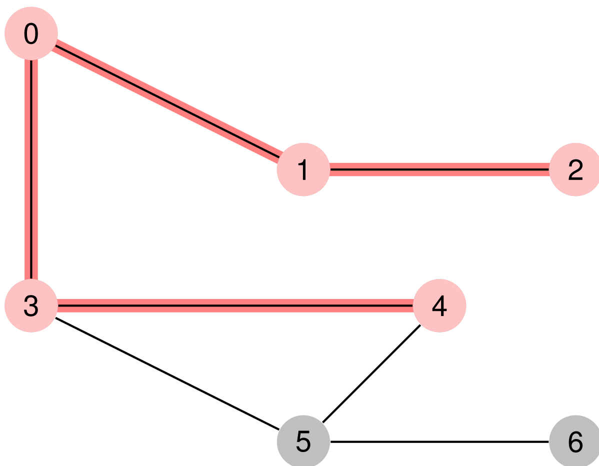
Ejemplo de recorrido de un grafo usando BFS



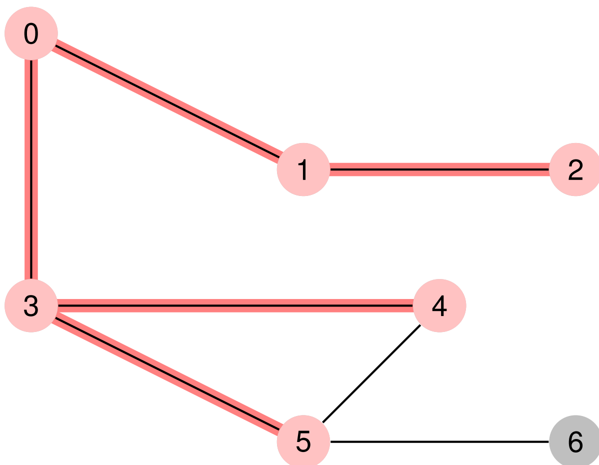
Ejemplo de recorrido de un grafo usando BFS



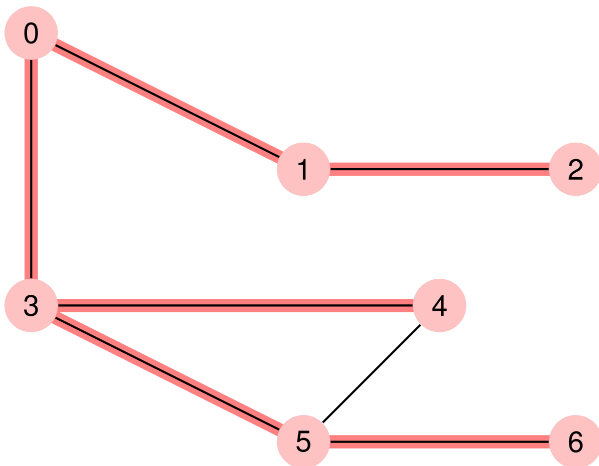
Ejemplo de recorrido de un grafo usando BFS



Ejemplo de recorrido de un grafo usando BFS



Ejemplo de recorrido de un grafo usando BFS



Implementemos un BFS resolviendo un problema. Vale notar que este problema no podía ser resuelto con DFS.

Calcular las distancias de un nodo a todos los demás

Dado un nodo inicial, queremos hallar la distancia de cada nodo a todos los demás. Recordemos que puede haber más de una forma de ir de un nodo a otro, pero para la distancia siempre tomaremos la mínima.

¿Cómo podemos resolverlo usando BFS? Por ahora no nos centremos en cómo se implementa el algoritmo, sino en cómo usarlo para resolver el problema. Luego intentaremos implementarlo.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).
- 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.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).
- 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.

Pseudocódigo del BFS

```
1  vector<int> BFS (Grafo listaAdyacencias, int nodoInicial):
2      cantidadDeNodos = longitud(listaAdyacencias)
3      queue<int> cola // aqui encolare los nodos que todavia no analizamos
4
5      // distancias[i] = distancia de i a nodoInicial, inicialmente es INFINITO
6      vector<int> distancias(cantidadDeNodos, INFINITO)
7
8      cola.encolar(nodoInicial)
9      distancias[nodoInicial] = 0
10     mientras (la cola no esta vacia):
11         tope = cola.tope() // tomo el tope de la cola (y lo saco de la cola)
12
13         para todos los vecinos v de "tope" en el grafo:
14             si la distancia entre v y el nodoInicial es infinito:
15                 distancias[v] = distancias[tope] + 1 // esta un nodo mas
                       lejos que su vecino
16                 cola.encolar(v) // encolo v para analizarlo luego
17
18     devolver distancias
```

Implementación del BFS

```
vector<int> BFS(vector<vector<int> > &lista, int nodoInicial){
    int n = lista.size(),t;
    queue<int> cola;
    vector<int> distancias(n,n);
    cola.push(nodoInicial);
    distancias[nodoInicial] = 0;
    while(!cola.empty()){
        t = cola.front();
        cola.pop();
        for(int i=0;i<lista[t].size();i++){
            if(distancias[lista[t][i]]==n){
                distancias[lista[t][i]] = distancias[t]+1;
                cola.push(lista[t][i]);
            }
        }
    }
    return distancias;
}
```

¿Preguntas?