GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: RECORRIDO EN PROFUNDIDAD (DFS)

1. Introducción

La busqueda en profundidad (*Depth First Search*) es uno de los principales algoritmos en grafos.

La primera búsqueda en profundidad encuentra la primera ruta lexicográfica en el gráfico desde un vértice de origen u a cada vértice. La primera búsqueda en profundidad también encontrará las rutas más cortas en un árbol (porque solo existe una ruta simple), pero en grafos generales este no es el caso.

2. Conocimientos previos

2.1. Recursividad

La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre. Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente

2.2. Pila

Una pila es una estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Para el manejo de los datos se cuenta con dos operaciones básicas: apilar, que coloca un objeto en la pila, y su operación inversa, des-apilar (retirar), que retira el último elemento apilado.

2.2.1. C++

En el caso de C++ para utilizar la cola incluimos la biblioteca *queue* que nos permite utilizar la cola propia del lenguaje la cual cuenta con las siguientes funcionalidades

- **stack::top():** Devuelve el elemento que esta en el tope de la pila.
- **stack::empty():** Está función retorna verdad si la pila está vacía y retorna falso si es que por lo menos tiene un elemento como tope.
- **stack::size():** Está función retorna cuantos elementos tiene la pila, pero sin embargo no se puede acceder a ellas por lo que no es muy usual el uso de está función.
- **stack::push(g):** Agrega el elemento 'gál tope de la pila.
- **stack::pop():** Elimina el elemento que esta en el tope de la pila.

```
#include <iostream>
#include <stack>
using namespace std ;
int main (){
   stack<int> stc;
   stc.push(100) ;
```

```
    stc.push (200) ;
    stc.push(300) ;
    cout<<stc.top()<<"\n";//resultado 300
    stc.pop();
    cout<<stc.top()<<"\n";//resultado 200
}
```

2.2.2. Java

En java para se uso de la pila debemos hacer uso de la clase *Stack* la cual extiende de la clase *Vector*. Para incluirla en nuestra solución debemos primero importarla del paquete *java* en el subpaquete *util*

```
import java.util.Stack ;
```

Luego solo debemos crear una instancia de esta clase y tenemos una pila lista para ser usada en nuestra solución.

```
Stack< <Tipo de dato> > pila =new Stack< <Tipo dedato> > ();
```

Por supuesto la expresión *<Tipo de dato>* se sustituye en la sentencia anterior por el tipo de dato que va almacenar la pila, por ejemplo a continuación una pila para almacenar cadena de caracteres.

```
Stack<String> pila =new Stack<String> ();
```

Los principales fucionalidades que posee la clase *Stack* son:

- **push:** Adiciona al tope de la pila el elemento pasado por parámetro

  ```
  pila.push("Matanzas");
  ```

- **pop:** Elimina y devuelve el elemento que esta en el tope de la pila siempre que esta tenga elemento, en caso de estar vacia se lanza una excepción.

  ```
  String tope=pila.pop();
  ```

- **peek:** Devuelve el elemento sin eleiminarlo que esta en el tope de la pila siempre que esta tenga elemento, en caso de estar vacia se lanza una excepción.

  ```
  String tope=pila.peek();
  ```

- **empty:** Comprueba si la pila esta vacia. Devuelve verdadero si la pila esta vacia y falso en caso contrario.

```
boolean isEmpty=pila.empty();
```

- **search:** Busca un elemento de la pila y devuelve la posición con respecto al tope de la pila donde se encuentra la primera ocurrencia del elemento. En caso que el elemento fuera el tope de la pila el valor devuelto sería 1 y así sucesivamente se iría incrementando a medida que se alejara del tope. En caso que elemento no este el valor devuelto será -1.

```
int position=pila.search("Matanzas");
```

3. Desarrollo

Una búsqueda en profundidad (en inglés DFS o *Depth First Search*) es un algoritmo que permite recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (*Backtracking*), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

El siguiente ejemplo ilustra el funcionamiento del algoritmo DFS sobre un grafo de ejemplo. El algoritmo comienza por el nodo 0.

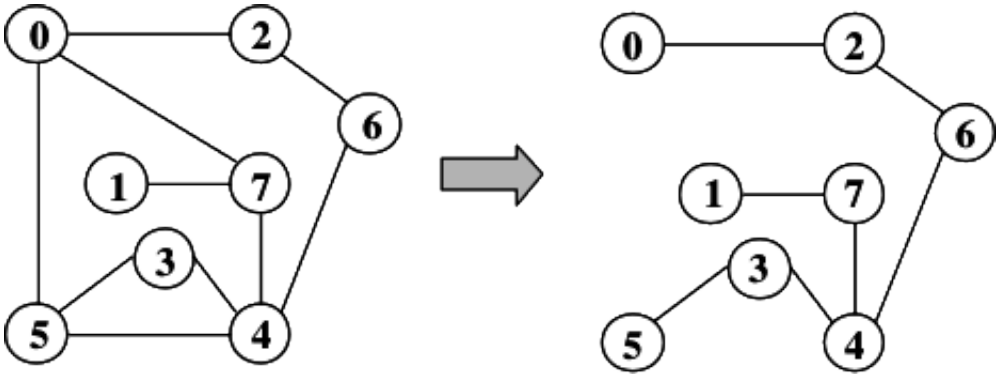

Figura 1: Salida del DFS sobre el grafo.

El pseudocódigo seria el siguiente:

```
DFS(grafo G)
  PARA CADA vertice u que pertenece V[G] HACER
     estado[u] = NO_VISITADO
     padre[u] = NULO
     tiempo = 0
     PARA CADA vertice u que pertenece V[G] HACER
        SI estado[u] = NO_VISITADO ENTONCES
           DFS_Visitar(u,tiempo)
```

```
DFS_Visitar(nodo u, int tiempo)
   estado[u] = VISITADO
   tiempo = tiempo + 1
   d[u] = tiempo
   PARA CADA v que pertenece Vecinos[u] HACER
      SI estado[v] = NO_VISITADO ENTONCES
         padre[v] = u
             DFS_Visitar(v,tiempo)
   estado[u] = TERMINADO
   tiempo = tiempo + 1
   f[u] = tiempo
```

Como se puede observar esta variante es recursiva y es un ejemplo clásico de un *Backtracking* lo cual no es muy recomendable usar para grafos con una cantidad de nodos mayor de 9 nodos por la cantidad de operaciones que representa (9!), es por eso que se decide sustituir el elemento recursivo por una estructura de datos que puede simular la "recursividad" inicial del algoritmo y disminuye la complejidad del algoritmo. Dicha estructura es una pila

La idea detrás de DFS es profundizar lo más posible en el grafo y retroceder una vez que esté en un vértice sin vértices adyacentes no visitados.

Es muy fácil describir/implementar el algoritmo recursivamente: comenzamos la búsqueda en un vértice. Después de visitar un vértice, realizamos un DFS para cada vértice adyacente que no hayamos visitado antes. De esta manera visitamos todos los vértices que son alcanzables desde el vértice inicial.

Podemos clasificar las aristas usando el tiempo de entrada y salida de los nodos finales u y v de las aristas (u,v). Estas clasificaciones se usan a menudo para problemas como encontrar puentes y encontrar puntos de articulación.

Realizamos un DFS y clasificamos las aristas encontradas usando las siguientes reglas:

- Si no se visita v:
  - Arista del árbol: si se visita v después de u, entonces el borde (u, v) se denomina arista del árbol. En otras palabras, si se visita v por primera vez y se está visitando u actualmente, entonces (u,v) se denomina arista del árbol. Estas aristas forman un árbol DFS y, por lo tanto, el nombre de arista del árbol.
- Si se visita v antes que u:
  - Aristas posteriores: si v es un antepasado de u, entonces el borde (u, v) es una arista posterior. v es un ancestro exactamente si ya ingresamos a v, pero aún no salimos de él. Las aristas posteriores completan un ciclo, ya que hay un camino desde el antepasado v hasta el descendiente u (en la recurrencia de DFS) y una arista desde el descendiente u hasta el antepasado v (arista posterior), por lo que se forma un ciclo. Los ciclos se pueden detectar usando aristas posteriores.

- Aristas delanteros: si v es un descendiente de u, entonces el borde (u, v) es una arista delantera. En otras palabras, si ya visitamos y salimos de v y entrada[u]<entrada[v] entonces la arista (u,v) forma una arista delantero.
- Aristas cruzadas: si v no es ni un ancestro ni un descendiente de u, entonces el borde (u,v) es uan arista cruzada. En otras palabras, si ya visitamos y salimos de v y entrada[u]>entrada[v] entonces (u,v) es una arista cruzada.

Nota: Las aristas delanteras y las aristas cruzadas solo existen en grafos dirigidos.

4. Implementación

4.1. C++

Variante no recursiva

```
#include <stack>
#include <vector>
#define MAX_N 5001
using namespace std;

struct Node{
   vector<int> adj;
};
Node graf[MAX_N];
bool mark[MAX_N];

inline void DFS(int start){
   stack<int> dfs_stek;
   dfs_stek.push(start);
   while(!dfs_stek.empty()){
      int xt = dfs_stek.top();
          dfs_stek.pop();
      mark[xt] = true;
      for (int i=0;i<graf[xt].adj.size();i++){
         if (!mark[graf[xt].adj[i]]){
            dfs_stek.push(graf[xt].adj[i]);
            mark[graf[xt].adj[i]] = true;
         }
      }
    }
}
```

Variante recursiva

```
vector< vector<int> > adj; // el grafo representado como una lista de
   adyacencia
int n; // numeros de vertices
```

```
vector<bool> visited;

void dfs(int v){
  visited[v] = true;
  for (int u : adj[v]){
     if (!visited[u])
       dfs(u);
  }
}
```

Esta es la implementación más simple de DFS. Como se describe en las aplicaciones, también podría ser útil calcular los tiempos de entrada y salida y el color del vértice. Colorearemos todos los vértices con el color 0, si no los hemos visitado, con el color 1 si los visitamos, y con el color 2, si ya salimos del vértice.

Aquí hay una implementación genérica que además los calcula:

```
vector< vector<int> > adj; // el grafo representado como una lista de
    adyacencia
int n; // numeros de vertices

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v){
   time_in[v] = dfs_timer++;
   color[v] = 1;
   for (int u : adj[v])
      if (color[u] == 0)
         dfs(u);
   color[v] = 2;
   time_out[v] = dfs_timer++;
}
```

4.2. Java

```
private class Graph{
   public List<ArrayList<Integer> > lAdyancencia;
   public int nodes;

   public Graph(int _nodes){
      this.nodes =_nodes;
      lAdyancencia =new ArrayList<ArrayList<Integer> >(this.nodes+1);
      for(int i=0;i<this.nodes+1;i++) {
         lAdyancencia.add(new ArrayList<Integer>());
```

```
        }
        boolean [] visited = new boolean [this.nodes+1];
        Arrays.fill(visited,false);
    }

    private void dfsRecursive(int node){
        visited[node]=true;
        int nvecinos = lAdyancencia.get(node).size(),vecino;
        for(int i=0;i<nvecinos;i++){
            vecino = lAdyancencia.get(node).get(i);
            if(visited[vecino]==false){
                dfsRecursive(vecino, access, visited);
            }
        }
    }

    private void dfsStack(int node){
        Stack<Integer> dfsStack =new Stack<Integer>();
        boolean [] visited =new boolean [this.nodes+1];

        Arrays.fill(visited, false);
        int nvecinos,vecino;

        dfsStack.add(node);

        while(dfsStack.empty()==false){
          node = dfsStack.pop();
          visited[node] =true;
          nvecinos = lAdyancencia.get(node).size();
          for(int i=0;i<nvecinos;i++) {
             vecino = lAdyancencia.get(node).get(i);
             if(visited[vecino] == false) {
                dfsStack.add(vecino);
                visited[vecino] = true;
             }
          }
         }
    }

    public void addEgde(int a,int b) {
        lAdyancencia.get(a).add(b);
    }
}
```

5. Complejidad

Este algoritmo tiene una complejidad de O(V+E) donde V es la cantidad de vértices del grafo y E las aristas.

6. Aplicaciones

El algoritmo recibe como parámetro el nodo inicial por el cual se inicia el DFS. Una bondad de este algoritmo es que los nodos solo se vistan una vez. Esto implica que si se salvan en alguna estructura las aristas que se van recorriendo se obtiene un conjunto de aristas de cubrimiento mínimo del grafo, lo cual se utiliza frecuentemente se utiliza para reducir la complejidad del grafo cuando la perdida de información de algunas aristas no es importante. Este resultado se conoce como árbol DFS (DFS Tree)

El DFS puede modificarse fácilmente y utilizarse para resolver problemas sencillos como los de conectividad simple, detección de ciclos y camino simple. Por ejemplo, el número de veces que se invoca a la acción DFS_R desde la acción DFS en el algoritmo anterior es exactamente el número de componentes conexas del grafo, lo cual representa la solución al problema de conectividad simple.

Otras de las aplicaciones de este algoritmo son:

- Para un grafo de pocos nodos se puede implementar un dfs recursivo lo cual podría generar todos los posibles caminos desde un nodos hasta los otros.
- Compruebar si un vértice en un árbol es un antepasado de algún otro vértice.
- Encuentra el ancestro común más bajo (LCA) de dos vértices.
- Compruebar si un grafo dado es acíclico y encontrar ciclos en un grafos.
- Encuentra las componentes fuertemente conectados en un grafo dirigido.
- Encuentra los puentes en un grafo dirigido

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando este algoritmo:

- DMOJ - Transformación: de A hacia B.
- DMOJ - Herederos
- DMOJ - Conteo de Pozos
- DMOJ - Fábrica de leche
- DMOJ - Picnic Vacuno