Graph terms-

1. Back edge – edge to itself or an ancestor -> DFS back edge detection indicate cycle
2. Forward edge – edge to descendant
3. Tree edges – edges that belong to a spanning tree when we visit a new vertex
4. Cross edges – edge to a node that is neither an ancestor or descendant

In DFS of undirected – every edge is either tree edge or back edge

In DFS of directed – contains all type of edges.

In BFS of undirected - every edge is either tree edge or back edge.

In BFS of directed – tree edge , back edge , cross edge and no forward edge.

Review DFS algorithm

*#depth first traversal of a graph  
#make sure to hold an adjacency matrix as well as a memoization trick to avoid nasty recursion  
#time complexity for recursion is at max is Number of vertices and Edges considering each Vertex has unique edges***class** Graph:  
 **def** \_\_init\_\_(self,num):  
 self.num = num  
 self.adjList = {}  
  
 **def** addToGraph(self,V,E):  
 **if**(V **in** self.adjList):  
 self.adjList[V].update({E})  
 **else**:  
 self.adjList[V] ={E}  
  
**def** runDFS(g,key,visited):  
 visited[key]={**True**}  
 print(**'Node '**+str(key))  
 *#print(visited)* **for** key **in** g.adjList:  
 **if**(visited[key] == **False**):  
 runDFS(g,key,visited)  
  
**def** run():  
 graphNodes = 4  
 g= Graph(graphNodes)  
 g.addToGraph(0,1)  
 g.addToGraph(0,2)  
 g.addToGraph(1,2)  
 g.addToGraph(2,0)  
 g.addToGraph(2,3)  
 g.addToGraph(3,3)  
 visited = [**False**]\*graphNodes *#damn this is too cool* runDFS(g,2,visited)  
 print (g.adjList)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 run()

There is also an iterative algorithm – use a stack to maintain the list of adjacent vertices. Pop it and print it. After that visit its neighbors. Continue this for every node. If you don’t do for every node then it will print only nodes reachable from that particular node.

Application of DFS –

1. **Detect cycle**
   1. **Algo to find cycle using DFS on directed graph.**

#include <iostream>

#include <vector>

//detect cycle for undirected graph

using namespace std ;

class graph

{

public:

vector<int> \*adjList;

bool \*visited;

int size ;

graph(int size)

{

adjList = new vector<int>[size];

visited = new bool[size];

this->size = size;

int i = 0;

for(;i<size;i++)

visited[i]=false;

}

void addEdge(int u ,int v)

{

adjList[u].push\_back(v);

adjList[v].push\_back(u);

}

int dfs\_visit(int n,int parent)

{

vector<int> &edges = adjList[n];

int i =0;

bool ret =false;

visited[n] =true;

for(;i<edges.size();i++)

{

if(!visited[edges[i]])

{

ret = dfs\_visit(edges[i],n);

if (ret == true) break;

}

else

{

if(visited[edges[i]])

if(edges[i] != parent)

{

ret = true;

break;

}

}

}

return ret;

}

bool isCycle()

{

int i =0;

bool ret = false;

for(;i<size;i++)

{

vector<int> &edges = adjList[i];

int j=0;

if(!visited[i])

if(dfs\_visit(i,-1) == true)

{

cout<<"cycle exists !"<<endl;

ret = true;

break;

}

}

return ret;

}

};

int main(int argc,char \*\*argv)

{

int size = 5;

graph g(size);

bool ret = false;

g.addEdge(1, 0);

g.addEdge(0, 2);

g.addEdge(2, 0);

g.addEdge(0, 3);

g.addEdge(3, 4);

ret = g.isCycle();

cout<<ret<<endl;

graph g2(3);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

ret = g2.isCycle();

cout<<ret<<endl;

return 0;

}

* 1. Algo to find cycle using union find (Disjoint set datastructure)algorithm on directed graph.
  2. Algo to find cycle using DFS on un-directed graph.

1. **Topo sort**
   1. **Using stack method , of print by exit or recursion unwind**

//TOPO SORT using stacks

#include<iostream>

#include<stack>

#include<vector>

using namespace std;

class graph

{

public:

int size;

vector<int> \*adjList;

stack<int> \*topoStack;

bool \*visited;

graph(int size)

{

this->size = size;

adjList = new vector<int>[size];

topoStack = new stack<int>;

visited = new bool[size];

int i;

for(i=0;i<size;i++)

visited[i]=false;

}

void topoRecurse(int v)

{

visited[v] = true;

for(vector<int>::iterator it = adjList[v].begin(); it < adjList[v].end() ; it++)

{

if(!visited[\*it])

{

topoRecurse(\*it);

}

}

topoStack->push(v);

}

void addEdge(int u , int v)

{

adjList[u].push\_back(v);

}

void topoSort()

{

int v = 0;

for (;v < size ; v++)

{

if(!visited[v])

topoRecurse(v);

}

int i = topoStack->size();

while(i)

{

cout<<topoStack->top()<<", ";

topoStack->pop();

--i;

}

cout<<endl;

}

};

int main(int argc,char \*\*argv)

{

int size = 6;

graph g(6);

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

g.topoSort();

return 0;

}