Design and Analysis of Algorithms

Assignment(September 5th 2017)

Solutions (Hints)

REMARK: The codes given below are just a guide, this is not the only way to write programs for these algorithms. What is important is that your program should implement the algorithm in the correct manner.

1. Mother vertex using BFS

A mother vertex in a graph G = (V,E) is a vertex v such that all other vertices in G can be reached by a path from v.

```
#include<iostream>
#include<bits/stdc++.h>
#include <list>

using namespace std;
int f=0;
int par[20];

// This class represents a directed graph using adjacency list representation
struct Graph
{
   int V;    // No. of vertices
   list<int> *adj;
};   // Pointer to an array containing adjacency lists

struct Graph* creategraph(int V)
{
            Graph* graph=new Graph;
   graph->V = V;
  graph->adj = new list<int>[V];
}

void addEdge(struct Graph* graph,int v, int w)
{
   graph->adj[v].push_back(w); // Add w to v's list, if the graph is undirected,add v to w's list as well
}

bool BFS(struct Graph* graph,int s)
{
   int d,flag=0;
   d=s;
```

```
   bool *visited = new bool[graph->V];
   for(int i = 0; i < graph->V; i++)
      visited[i] = false;

   // Create a queue for BFS
   list<int> queue;

   // Mark the current node as visited and enqueue it
   visited[s] = true;
   queue.push_back(s);

   // 'i' will be used to get all adjacent vertices of a vertex
   list<int>::iterator i;

   while(!queue.empty())
   {
      // Dequeue a vertex from queue and print it
      s = queue.front();
      cout << s << " ";
      f++;
      queue.pop_front();

      // Get all adjacent vertices of the dequeued vertex s
      // If a adjacent has not been visited, then mark it visited
      // and enqueue it
      for(i = graph->adj[s].begin(); i !=graph-> adj[s].end(); ++i)
      {

         if(!visited[*i])
         {
            visited[*i] = true;
            queue.push_back(*i);
         }
      }
   }
   cout<<"\n";
   for(int j=0;j<=6;j++)
   {
           if(visited[j]!=true)
           flag=1;
           }
           if(flag==0)
           cout<<"The vertex "<< d <<" is a mother vertex";
   cout<<"\n";
}
```

```
int main()
{

   struct Graph* graph=creategraph(7);
   addEdge(graph,0,1);
   addEdge(graph,0,2);
   addEdge(graph,1,3);
   addEdge(graph,4,1);
  addEdge(graph,6,4);
  addEdge(graph,5,6);
  addEdge(graph,5,2);
  addEdge(graph,6,0);
   for(int i=0;i<=6;i++)
   {
           cout<<"the BFS traversal from vertex "<< i << " is\n";
           BFS(graph,i);
           cout<<"\n";
           }

   return 0;
}
```

2. Transitive Closure using Depth First Traversal

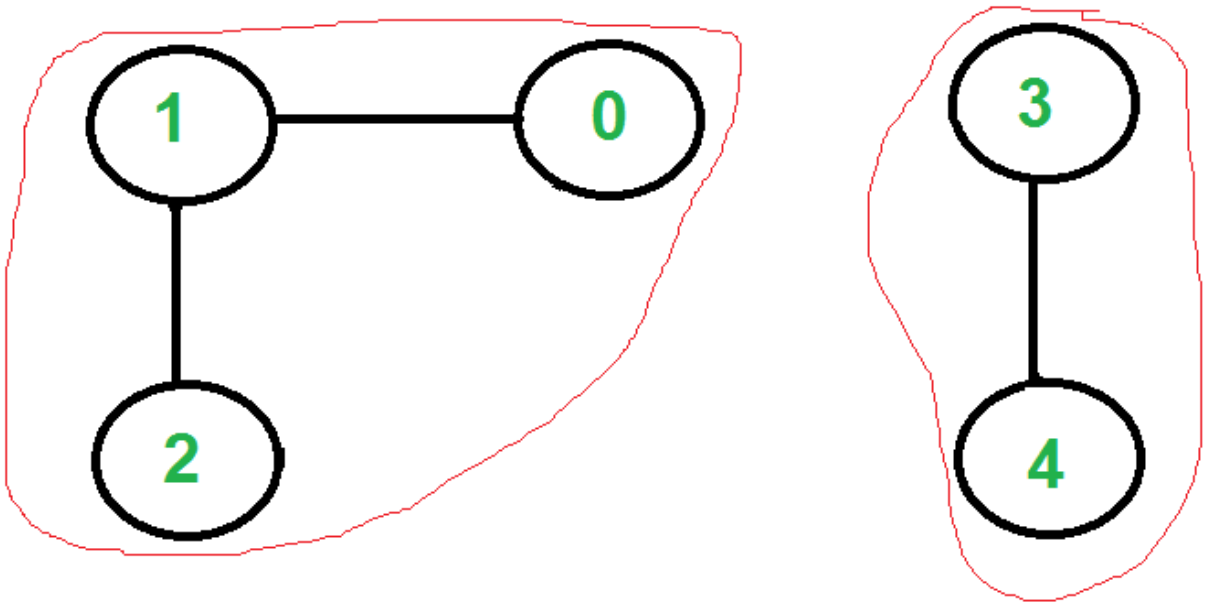
Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable mean that there is a path from vertex u to v. The reach-ability matrix is called transitive closure of a graph.

```
#include<bits/stdc++.h>
#include<iostream>
#include<list>
using namespace std;
int tc[10][10];

struct Graph
{
   int V;
   list<int> *adj;
};

struct Graph* creategraph(int V)
{
           Graph* graph=new Graph;
   graph->V = V;
   graph->adj = new list<int>[graph->V];
}
void addEdge(struct Graph* graph,int v, int w)
{
```

```
                    graph->adj[v].push_back(w);
}

// A recursive DFS traversal function that finds
// all reachable vertices for s.
void DFSUtil(struct Graph* graph,int s, int v)
{
   // Mark reachability from s to t as true.

   tc[s][v] = 1;

   // Find all the vertices reachable through v
   list<int>::iterator i;
   for (i =graph->adj[v].begin(); i !=graph->adj[v].end(); ++i)
      if (tc[s][*i] == 0)
         DFSUtil(graph,s, *i);
}

// The function to find transitive closure. It uses
// recursive DFSUtil()
void transitiveClosure(struct Graph* graph)
{
   // Call the recursive helper function to print DFS
   // traversal starting from all vertices one by one
    for (int i=0; i<4; i++)
   {
           for(int j=0;j<4;j++)
      tc[i][j]=0;
   }
   for (int i = 0; i < 4; i++)
      DFSUtil(graph,i,i); // Every vertex is reachable from self.

   for (int i=0; i<4; i++)
   {
      for (int j=0; j<4; j++)
         cout << tc[i][j] << " ";
      cout << endl;
   }
}
 int main()
{

   struct Graph* graph= creategraph(4);
   addEdge(graph,0, 1);
   addEdge(graph,0, 2);
   addEdge(graph,1, 2);
   addEdge(graph,2, 0);
   addEdge(graph,2, 3);
```

```
    addEdge(graph,3, 3);
    cout << "Transitive closure matrix is \n";
    transitiveClosure(graph);

    return 0;
}
```

3. Connected components of undirected graph (Here DFS is used)



There are two connected components in above undirected graph
0 1 2
3 4

```
#include<bits/stdc++.h>
#include<iostream>
#include<list>
using namespace std;


struct Graph
{
    int V;
    list<int> *adj;
};

struct Graph* creategraph(int V)
```

```
{
  Graph* graph=new Graph;
   graph->V = V;
   graph->adj = new list<int>[graph->V];
}
void addEdge(struct Graph* graph,int v, int w)
{
                        graph->adj[v].push_back(w);
                        graph->adj[w].push_back(v);
}
void DFSUtil(struct Graph* graph,int v, bool visited[])
{
   // Mark the current node as visited and print it
   visited[v] = true;
   cout << v << " ";

   // Recur for all the vertices adjacent to this vertex
   list<int>::iterator i;
   for (i = graph->adj[v].begin(); i != graph->adj[v].end(); ++i)
   {
                          if (!visited[*i])
         DFSUtil(graph,*i, visited);
           }
}
void connectedcomponents(struct Graph*  graph)
{
            bool *visited = new bool[graph->V];
   for(int i = 0; i < graph->V; i++)
      visited[i] = false;

   for (int j=0; j<graph->V; j++)
   {
      if (visited[j] == false)
      {
         // print all reachable vertices
         // from v
         DFSUtil(graph,j, visited);

         cout << "\n";
      }
   }
}
int main()
{
   // Create a graph given in the above diagram
   struct Graph* graph= creategraph(5); // 5 vertices numbered from 0 to 4
   addEdge(graph,1, 0);
   addEdge(graph,1, 2);
```

```
   addEdge(graph,3, 4);

   cout << "Following are connected components \n";
   connectedcomponents(graph);

   return 0;
}
```

4. Check whether directed graph is cyclic or not. Together with this ,if we check connectivity as well, we know whether graph is tree or not.
To check connectivity,check whether the dfs of a vertex reaches all other vertices, if not it is not connected.
A graph has to connected and acyclic to be a tree.

```
#include<bits/stdc++.h>
#include<iostream>
#include<list>
using namespace std;

struct Graph
{
   int V;
   list<int> *adj;
};

struct Graph* creategraph(int V)
{
          Graph* graph=new Graph;
   graph->V = V;
   graph->adj = new list<int>[graph->V];
}
void addEdge(struct Graph* graph,int v, int w)
{
                    graph->adj[v].push_back(w);
}
bool iscyclicutil(struct Graph* graph,int v,bool visited[],bool recstack[])
{
          if(visited[v]==false)
          {
                    // Mark the current node as visited and part of recursion stack
                    visited[v]=true;
                    recstack[v]=true;

                    list<int>::iterator i;
      for(i =graph->adj[v].begin(); i!=graph->adj[v].end(); ++i)
      {
         if ( !visited[*i] && iscyclicutil(graph,*i, visited,recstack) )
            return true;
```

```
        else if (recstack[*i])
            return true;
      }

   }
   recstack[v] = false;  // remove the vertex from recursion stack
   return false;
}
bool iscyclic(struct Graph* graph)
{
  bool *visited = new bool[graph->V];
   bool *recstack = new bool[graph->V];
   for(int i = 0; i < graph->V; i++)
   {
      visited[i] = false;
      recstack[i] = false;
   }

   // Call the recursive helper function to detect cycle in different
   // DFS trees
   for(int i = 0; i < graph->V; i++)
      if (iscyclicutil(graph,i, visited, recstack))
         return true;

   return false;
}
int main()
{
            struct Graph* graph= creategraph(4);
   addEdge(graph,1, 2);
   addEdge(graph,2, 3);
   addEdge(graph,3, 1);
   if(iscyclic(graph))
   cout<<"The graph contains a cycle";
   else
   cout<<"The graph does not contain a cycle";
}
```