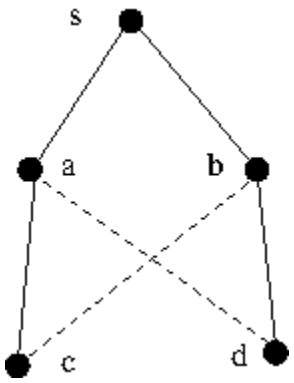
Tutorial 9 Solutions

1. (Jalay) Please study the undirected graph below and the tree edges (in black), non-tree edges in green (and some forbidden edges in red). We have seen that this gives a tree, in fact, a shortest-path tree for the vertex s. In other words, it is a tree T for which the path from s to another vertex u is the shortest path in the original graph. Thus bfs produces a shortest path tree. Note that there may be many shortest path trees and the particular tree produced by bfs is one of them, which depends on the vertex ordering, i.e., how the vertices are ordered among themselves. Now, given a shortest path tree T, is there always an ordering of the vertices which will produce this tree T as outputs?



Consider the graph above with the tree and non-tree edges. Assume that we start at s. Without loss of generality, let us assume a<b. When that happens, we should have b,c,d in the queue, with c and d in some order. When we next come to b, both c and d will be grey and will not be visited. Thus the edge (b,d) will not be taken.

Thus not every shortest path tree is a bfs tree for some order.

2. (Shivam) For the same graph below, execute dfs and produce a trace of the algorithm. List the vertices in order of push and pop events and mark the arrival and departure numbers. Modify the DFS to label tree edges twice with the counter, once while going forward, i.e., push (i.e., the recursive call) and the second time when popping, i.e., when exiting.

Ans:

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

const int N = 2; // Number of vertices 0-indexed
int e; // Number of Edges
bool vis[N]; // visited array
int in[N],out[N]; // in-out times
vector<int> G[N]; // adjacency list of graph
```

```
int counter = 0;

void dfs(int src , int par = -1)
{
    in[src] = counter++;
    vis[src] = true;
    cout<<"Pushed vertice No. "<<src<<"\n";

    for(auto child : G[src])
    {
        if(!vis[child])
        {
            cout<<"Edge push "<<src<<"---"<<child<<"\n";
            dfs(child,src);
        }
    }

    cout<<"Popped vertice No. "<<src<<"\n";

    if(par != -1)
        cout<<"Edge pop "<<src<<"---"<<par<<"\n";

    out[src] = counter++;
}

int main()
{
    memset(in,-1,sizeof(in));
    memset(out,-1,sizeof(out));
    int u,v;
    cin>>e;
    for(int i = 0; i < e; i++)
    {
        cout<<"hello";
        cin>>u>>v;
        G[u].push_back(v);
        G[v].push_back(u);
    }

    dfs(0,-1); // Treating 0 as source node
```

```
    for(int i = 0; i < N; i++)
    {
        cout<<"In time of vertice "<<i<<" = "<<in[i]<<"\n";
        cout<<"Out time of vertice "<<i<<" = "<<out[i]<<"\n";
    }
}
```

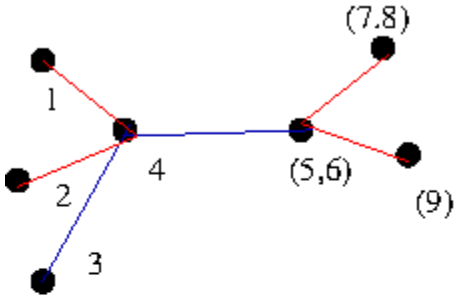
3. (Sachin) Suppose that there is an undirected graph G(V,E) where the edges are colored either red or blue. Given two vertices u and v. It is desired to (i) find the shortest path irrespective of colour, (ii) find the shortest path, and of these paths, the one with the fewest red edges, (iii) a path with the fewest red edges. Draw an example where the above three paths are distinct. Clearly, to solve (i), BFS is the answer. How will you design algorithms for (ii) and (iii)?

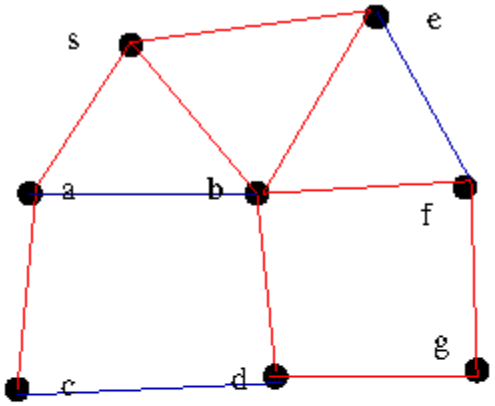
   Solution :
   i) For computing the shortest path we can do Breadth First Search. It will give us the shortest path.
   ii) Here is the solution:

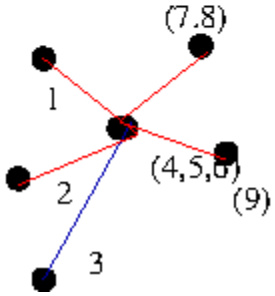
```
BFS(G,s)
01 for each vertex u ∈ V[G]-{s}
02       color[u] ← white
03       d[u] ← ∞
04       π[u] ← NIL
05 color[s] ← gray
   red[s]=0;
06 d[s] ← 0
07 π[u] ← NIL
08 Q ← {s}
09 while Q ≠ ∅ do
10       u ← head[Q]
11       for each v ∈ Adj[u] do
12       adder=0; if (u,v)==red adder=1;
         if color[v] = white then
13       color[v] ← gray
14       d[v] ← d[u] + 1
         red[v]=red[u]+adder;
15       π[v] ← u
16       Enqueue(Q,v)
         if (color[v]=gray or black) && (d[v]=d[u]+1) && (red[v]>red[u]+adder)
            red[v]=red[u]+adder;
```

π[v] ← u
17 Dequeue(Q)
18 color[u] ← black

(iii) There is no easy way unless you use Dijkstra. There is another way which is “edge collapsing”. If there is a blue edge (u,v), we collapse this edge, i.e., create a graph with one fewer edge. This will mean changing the adjacency list and also keeping track of older vertex names.







4. Suppose that we have a graph and we have run dfs starting at a vertex s and obtained a dfs tree. Next, suppose that we add an edge (u,v). Under what conditions will the dfs tree change? Give examples.

   The tree will not change iff and only if the new edge added is an up-down edge.

5. (Ankit - BTech) We have a new search algorithm which uses a set S for which we have two functions (i) add(x,S) which adds x to S, and (ii) y=select(S) which returns an element of S following a certain rule (and removes that element from the set S).

   Function mysearch
   Global visited;
   For all u visited[u]=false;
   for all edges e, found[e]=false;
   S=empty;

```
add(s,S); nos=1; record[nos]=s; visited[s]=true;
While nonempty(S)
   y=select(S)
   nos=nos+1; record[nos]=y;
   For all v adjacent to y
     If visited[v]==false
        visited[v]=true;
        found[(u,v)]=true;
        add(v,S);
     Endif;
   Endfor;
Endwhile
Endfunction;
```

(i) Compare the bfs and dfs algorithms with the above code. Take special care in understanding ‘visited’.

Solution*.* It is easy to see that the function ‘mysearch’ as defined here is a generalization of the well known searching algorithms BFS and DFS.

If the set S in ‘mysearch’ is implemented with a First-In-First-Out (FIFO) rule, it becomes BFS, since S then operates like a queue, with newly found vertices being added at the back of the queue for further exploration. If S is implemented with a Last-In-First-Out (LIFO) rule, it becomes DFS, with S operating like a stack; newly found vertices are explored earlier.

S may also be implemented with any arbitrary rule, resulting in various kinds of search trees. The ‘found’ array maintains all the edges that are part of this search tree. The ‘visited’ boolean value for any vertex v in the graph is set to true exactly when v is added to the set S. In a connected graph, every vertex will get added to S at some point of time, so by the end of the algorithm, ‘visited’ is true for every vertex.

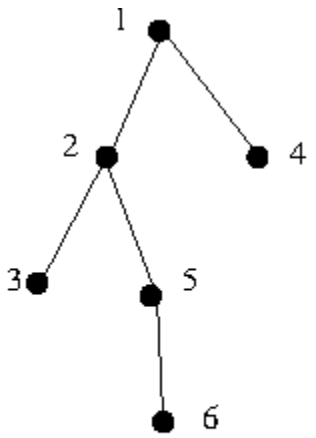
Moreover, we can say that, the ‘visited’ boolean corresponding to vertex v is set to true exactly when a path has been established (and indicated by the ‘found’ array) from the source vertex s to the vertex v. This is because visited[v] becomes true when v is added to S, i.e., when an edge is found (and added to the ‘found’ array) to v from a vertex that was previously in S. That vertex in turn must have had an edge from some other vertex, and so on, all the way back to the source s. (Can be proved formally by induction.)

(ii) Let us look at the sequence record[1],record[2],...,record[n]. Show that there is a path from record[i] to record[i+1] using only edges which have been found at that point.

This has been argued above,

(iii) Compare BFS and DFS in terms of the above path lengths.

Let Pi=(v1,v2,...,vk) be the path from record[i] to record [i+1], with vk=record[i+1] and v1=record[i]. Look at v(k-1). This is a vertex which has already been visited, So v(k-1) is record[k] for a k<i+1. Thus this edge must be visited when record[i+1] is found. Now, consider record[j] for j>i+1. Also note that,we must return back to s, the starting point. Thus, either there is a record[j] with j>i+1 which is not below record[i+1] in the tree or when we return, the edge (v(k-1),record[i+1]) is traversed again. In dfs, unless there is a pop, the path record[i] to record[i+1] is of length one. Thus the dfs is the order which will give the shortest total path.



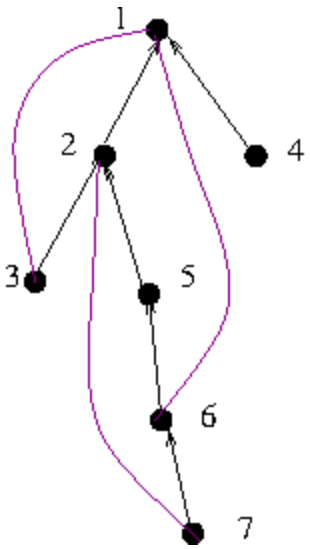
(1,2)(2,3)(3,2,1,4)(4,1,2,5)(5,6)(6,5,2,1)

---

6. (Ankit - MTech1) This is the theoretical basis of edge-2-connectedness. Let G(V,E) be a graph. We define a relation on edges as follows: two edges e and f are related (denoted by e~f iff there is a cycle containing both. Show that this is an equivalence relation. The equivalence class [e] of an edge e is called its 2-connected component.

7. Given a dfs tree for an undirected graph and a vertex v, we define C(v) as the edge (x,y) where x is a descendent of y while x is a parent of v, with x!=v and y!=v. Modify dfs to list C(v) and to compute |C(v)| . In the example below C(6)=1, C(5)=0 and C(1) =3. How much time does your algorithm take? Modify the code counterdfs.c

```
Increment(u,v) // which is a back-edge
While Pi(u) !=v
   C(Pi(u))=C(Pi(u)+1;
   u=Pi(u)
 Endwhile
```



Incremer

(3,1)= 2
(7,2)=5,6
(6,1)=5,2

We may introduce the code during the usual bfs. This will increase the time from O(e) to O(E)*O(V). Can this be done in O(E) time?