Graphs

Problems

Give an algorithm to check if a simple path exists from a source s to destination d in a graph G. Assume G is represented using the Adjacency Matrix :

```
void HasSimplePath(struct graph *G, int s, int d) {
    int t;
    visited[s] = 1;
    if(s == d) {
        return 1;
    }
    for(int i = 0; i < G->V; i++) {
        if(G->adjMatrix[s][t] && !visited[t]) {
            if(DFS(G, t, d)) {
                return 1;
            }
        }
    }
    return 0;
}
```

Count simple paths for a given graph G which has simple path from source s to destination d. Assume G is represented using the Adjacency Matrix :

```
void CountSimplePaths(struct Graph *G, int s, int d) {
    int t;
    visited[s] = 1;
    if(s == d) {
        count ++;
        visited[s] = 0;
        return;
    }
    for(t = 0; t < G->V; t++) {
        if(G->adjMatrix[s][t] && !visited[t]) {
            CountSimplePaths(G, t, d);   // recursively explore the
path
            visited[t] = 0;  // for backtracking
        }
    }
}
```

Finding cut vertices of a graph using DFS :

```
int adjMatrix[256][256];
int dfsnum[256], num = 0, low[256];
void CutVertices(int u) {
```

```
    low[u] = dfsnum = num++;
    for(int v = 0; v < 256; v++) {
        if(adjMatrix[u][v] && dfsnum[v] == -1) {
            CutVertices(v);
            if(low[v] > dfnum[u]) {
                cout << "Cut Vertex : " << u << " ";
            }
            low[u] = min(low[u], low[v]);
        }
        else {  // (u, v) is a backedge
            low[u] = min(low[u], dfsnum[v]);
        }
    }
}
```

Finding cut edges/ bridges using DFS :

```
// an edge is a bridge if it is not a part of cycle

int dfsnum[256], num = 0, low[256];
void Bridges(struct graph *G, int u) {
    low[u] = dfsnum = num++;
    for(int v = 0; v < G->V; v++) {
        if(G->adjMatrix[u][v] && dfsnum[v] == -1) {
            Bridges(v);
            if(low[v] > dfnum[u]) {
                cout << "Bridge : " << u << ", " << v << " ";
            }
            low[u] = min(low[u], low[v]);
        }
        else {  // (u, v) is a backedge
            low[u] = min(low[u], dfsnum[v]);
        }
    }
}
```

Find strongly connected components using DFS :

```
int adjMatrix[256][256], table[256];
vector<int> st;
int counter = 0;
int dfsnum[256], num = 0, low[256];
void StronglyConnectedComponents(int u) {
    low[u] = dfsnum[u] = num++;
    push(st, u);
    for(int v= 0; v < 256; v++) {
        if(graph[u][v] && table[v] == -1) {
```

```
            if(dfsnum[v] == -1) {
                StronglyConnectedComponents(v);
            }
            low[u] = min(low[u], low[v]);
        }
    }
    if(low[u] == dfsnum[u]) {
        while(table[u] != counter) {
            table[st.back()] = counter;
            push(st);
        }
        counter++;
    }
}
```

Count the no of connected components of a graph which is represented by adjacency matrix : using DFS

```
int visited[G->V];
void DFS(struct graph *G, int u) {
    visited[u] = 1;
    for(int v = 0; v < G->V; v++) {
        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}

void DFSTraversal(struct graph *G) {
    int count = 0;
    for(int i = 0; i < G->V; i++) {
        visited[i] = 0;
    }
    for(int i = 0; i < G->V; i++) {
        (!visited[i]) {
            DFS(G, i);
            count++;
        }
    }
    return count;
}
```

Count the no of connected components of a graph which is represented by adjacency matrix : using BFS

```
int visited[G->V];
void BFS(struct graph *G, int u) {
    int v;
    Queue q = createQueue();
    enQueue(Q, u);
    while(!isEmpty(Q)) {
        u = deQueue(Q);
        cout << u;
        visited[s] = 1;
    }
    for each unvisited adjacent node v of u {
        enQueue(Q, v);
    }
}

void BFSTraversal(struct graph *G) {
    for(int i = 0; i < G->V; i++) {
        visited[i] = 0;
    }
    for(int i = 0; i < G->V; i++) {
        (!visited[i]) {
            BFS(G, i);
        }
    }
}
```

For an undirected graph G(V, E), given an algorithm for finding a spanning tree which takes O(|E|) TC (not necessarily a MST) :

```
S = {};
for each edge e belongs to E {
     if(adding e to S does not form a cycle) {
          add e to S;
          mark e;
     }
}
```

Detecting a cycle in a DAG :

```
int DetectCycle(struct graph *G) {
    for(int i = 0; i < G->V; i++) {
        visited[s] = 0;
        predecessor[i] = 0;
    }
    for(int i = 0; i < G->V; i++) {
```

```
        if(!visited[i] && HasCycle(G, i)) {
            return 1;
        }
    }
    return 0;
}

int HasCycle(struct graph *G, int u) {
    visited[u] = 1;
    for(int i = 0; i < G->V; i++) {
        if(G->Adj[s][i]) {
            if(predecessor[i] != u && visited[i]) {
                return 1;
            } else {
                predecessor[i] = u;
                return HasCycle(G, i);
            }
        }
    }
    return 0;
}
```

Find depth of a DAG :

```
int DepthInDAG(struct graph *G) {
    struct Queue *Q;
    int counter;
    int v, w;
    Q = createQueue();
    counter = 0;
    for(v = 0; v < G->V; v++) {
        if(indegree[v] == 0) {
            enQueue(Q, v);
        }
    }
    enQueue(Q, '$');
    while(!isEmpty(Q)) {
        v= deQueue(Q);
        if(v == '$') {
            counter++;
            if(!isEmpty(Q)) {
                enQueue(Q, '$');
            }
        }
        for each w adjacent to v {
            if(--indegree[w] == 0) {
```

```
                enQueue(Q, w);
            }
        }
    }
    deleteQueue(Q);
    return counter;
}
```

Hamiltonian Path in DAG :

```
bool seenTable[32];
void HamiltonianPath(struct graph *G, int u) {
    if(u == t) // check that we have seen all vertices
    else {
        for(int v = 0; v < n; v++)
        if(!seenTable[v] && G->Adj[u][v]) {
            seenTabe[v] = true;
            HamiltonianPath(v);
            seenTabe[v] = false;
        }
    }
}
```

Reversing Graph :

```
Graph reverse(struct Graph *G) {
    Graph reversedGraph = createGraph();
    for each vertex of given graph G {
        for each vertex w adjacent to v {
            add the w to v edge in reversedGraph;
        }
    }
    return reversedGraph;
}
```