Depth First Search

Notations:
G.V (vertices of graph)
G.E (Edges of graph)
(u,v) (a particular edge between two vertices "u" and "v")
G.Adj[v] (list of adjacent vertices of any particular vertex 'v')
Attributes of each vertex:
Color: white (unvisited), gray (partially visited i.e., vertex itself has been visited but all adjacent vertices are not visited yet), black (vertex itself and all adjacent vertices has been visited)
PI (predecessor) it will help to determine the path of traversal from “src” vertex to “dest” vertex.
d (In DFS, d will be used to determine the discovery time or pre-time)
f (finish time or post time)
time (a global variable whose value will be incremented upon vertex discovery or finish)

```
DFS(G){
   //Initialization (white color means all vertices are unvisited)
   for each vertex 'v' belongs to G.V {
      v.color = white
      v.PI = NULL
   }
```

Since there is not a fixed starting point of graph so we are not sure whether the complete graph can be traversed in single run by selecting any random vertex as a source or not. So, after every iteration of DFS traversal we will check for any unvisited vertex in the graph and if found then we call DFS traversal again by selecting that unvisited vertex as a source.

```
   for each vertex 'v' belongs to G.V {
      If (v.color == white){   //select unvisited vertex as a source
         DFS-visit(G,v)
      }
   }
}
time = 0
DFS-visit (G, src){
   time++
   src.color = gray   //partially visited
   src.d = time       // discovery time
   for (each vertex 'v' belongs to G.Adj[src]){       // iterate for all adjacent vertices
      If (v.color == white){
         v.PI = src
         DFS-visit(G,v) //recursive call to unvisited adjacent vertex to explore the depth
      }
      time++
      v.f = time
      v.color = black
   }
}
```

complexity: O(V+E) for adjacency list and O(V^2) for adjacency matrix.

Classification of Edges, Cycle Detection, conversion into DAG

Consider an edge (u,v) i.e., (u -> v) between two vertices ‘u’ and ‘v’ of a directed graph.
Tree Edge: An edge to unvisited vertex will be treated as a tree edge. If the color attribute of vertex ‘v’ is white it means it is unvisited vertex and the edge between ‘u’ and ‘v’ will be tree edge.
Backward edge: An edge to any of the predecessor is called backward. If the color attribute of vertex ‘v’ is gray, it means the edge (u,v) is going back to any of its predecessor.
Cross Edge: An edge to completely visited vertex connecting the components of graph.
Forward Edge: An edge to a completely visited vertex from the predecessor.
Backward edge is most important. It will be used for cycle detection in a graph, or it will help us to convert a cyclic graph into an acyclic graph or directed acyclic graph i.e., (DAG).
Same DFS Algorithm will be used with minor changes.

```
DFS(G)
{
   //Initialization
   for each vertex 'v' belongs to G.V {
      v.color = white
      v.PI = NULL
   }
   for each vertex 'v' belongs to G.V {
      If (v.color == white){
         DFS-visit(G,v)
      }
   }
}
```

```
time = 0
DFS-visit (G, src){
   time++
   src.color = gray
   src.d = time
   for (each vertex 'v' belongs to G.Adj[src]){
      If (v.color == white){
         (src, v) will be a tree edge
         v.PI = src
         DFS-visit(G,v)
      }
      else if (v.color == gray)
         (src, v) will be a backward edge
```

1: Presence of backward edge in the graph is the indication of cyclic graph (cycle detection)
2: Removal of backward edge from the graph can convert it into Acyclic graph. For a directed graph this removal can convert it into a directed acyclic graph called (DAG).

```
         remove (src,v)
      }
```

//if color is neither white nor gray then it will be black so both cross and forward edges will be going to vertex whose color is black. We can differentiate them with discovery time.

```
      else if (src.d < v.d)
         (src,v) will be a forward edge
      else
         (src,v) will be a cross edge
      time++
      v.f = time
      v.color = black
   }
}
```

Topological Sort

Pre-condition: Input graph must be DAG.
Since we are not sure whether the input graph is cyclic or acyclic, firstly we will apply DFS and remove backward edge to convert it into DAG.
We can use topological sort for scheduling related task or to represent a graph in a Linearized way.
Assume that we are solving the problem for process scheduling. In process scheduling, a process may be dependent upon multiple processes, so we associate an attribute of **depCount** with each vertex.

```
Topological Sort(G){
  for each vertex 'v' belongs to G.V
  {
    v.depCount = 0
  }
  for each vertex 'v' belongs to G.V
  {
    for each vertex 'u' belongs to G.Adj[v]
    {
      u.depCount = u.depCount + 1
    }
  }
  Now create a ready list "R" and a solution list "S"
//Insert vertices with depCount equals to 0 in the ready list because these vertices are not dependent upon any other process
  for(each vertex 'v' belongs to G.V)
  {
    if(v.depCount == 0)
    {
      R.addtoTail(v)
    }
  }
//Now iterate until the ready list is non-empty
  while (R != {})
  {
    v = R.removeHead()
    S.addtoTail(v)
    for each vertex 'u' belongs to G.Adj[v]
    {
      u.depCount = u.depCount - 1
      if(u.depCount == 0) //if depCount of any vertex became 0 then add it to ready list because it is
                          now independent
      {
        R.addtoTail(u)
      }
    }
  }
//To display the schedule or linearized output you can print the solution set "S"
}
```

Strongly Connected Components (SCC)

Vertices (a_1, a_2, a_3, …, a_m) in a graph are called strongly connected if we can traverse all of them by starting from any vertex of the graph. If there are “N” vertices in the graph, and we can travers all of them by starting from any vertex, then there is only one strongly component in the graph.

Main Idea to determine strongly connected components in graph:

1: Apply Depth-First Search (DFS) on the original graph “G” and keep a stack “s”. Add vertices to the stack once they are completely explored, meaning when they are marked as black.

2: Now Take transpose the graph so that we can reach the sink component.

3: Apply DFS traversal on the transposed graph by using the stack.

```
SCC(G)
{
//Apply Depth-First Search (DFS) on the original graph “G” and keep a stack “s”. Add vertices to the stack once they are completely explored, meaning when they are marked as black.
   1: DFS_1(G)
//Draw a component DAG to understand the phenomenon of extracting strongly connected components. Each DAG must contain at least one source component (i.e., no inward edge) and one sink component (i.e., no outward edge). We can extract SCC by starting the traversal from sink components. Why?
Ans: Because if we start the traversal from sink component it will not enter any other component and just traverse the vertices inside the sink component.
The question is how to reach the sink component?
Ans: Take the transpose of the graph. Transpose means reverse the edges. Now draw the component DAG of transposed graph. You will observe that the source component of the original graph is now the sink component of transposed graph. We have also maintained a stack and the top element in the stack is the part of sink component of transposed graph. So, we can extract SCC by applying DFS using stack.
   2: $G^T$ = Transpose(G)
   3: DFS_2($G^T$)
}
```

```
DFS_1()
{
   for each v belongs to G.V
      v. key = ∞
      v. PI = NULL
   for each v belongs to G.V
   {
      if (v. color == white)
      {
         DFS-visit_1(G, v)
      }
   }
}
create a stack "s" and insert the vertices in stack when they are completely visited.
time = 0
DFS-visit_1(G,src){
   time++
   src. d = time
   src. color = gray
   for each v belongs to G. Adj[src]
   {
      if (v. color == white)
      {
         v. PI = src
         DFS-visit (G, v)
      }
   }
   time++
   v. color = black
   s. push(v)
}
```

```
DFS_2(G, src)
{
   for each v belongs to G.V
      v.key =
      v.PI = NULL
//count will be used to determine total number
of strongly connected components in graph.
   count = 0
   while (s != {})
   {
      v = s.pop()
      if(v.color == white)
      {
         count++
         DFS-visit_2(G, v, count)
      }
   }
}
```

```
DFS-visit_2(G,src,n)
{
   src. color = gray
   src. compNo = n
// Vertices sharing the same component number
belong to the same component.
   for each v belongs to G.Adj[src]
   {
      if(v.color == white)
         v.PI = src
         DFS-visit_2(G,v,n)
   }
   src.color = black
}
```