Depth-First Search (DFS)

This document contains summary of some of the items discussed as part of the COMS 3110 Lectures. This document **does not replace the lecture materials**. This document may contain some topics that are not covered as part of the lecture; you will not be tested on those parts, they are made available to you for gaining further knowledge on topics/concepts that are related to class-lecture.

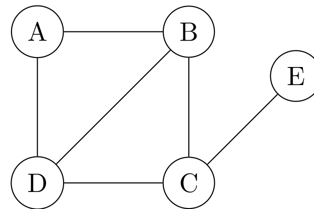
1 Topics Covered

- DFS
- Applications of DFS: topological ordering

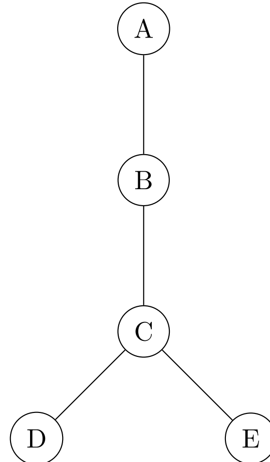
2 Depth-First Search (DFS)

- The DFS algorithm will give us a graph that does not contain cycles.
- DFS differs from BFS in that we keep exploring down a path until we cannot go any further. Instead of exploring in a layered manner, we explore nodes in a depth-first fashion, hence the name.

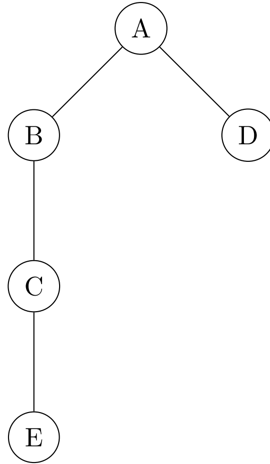
For a graph like this:



The DFS exploration graph starting from vertex A looks like this:



On the other hand, the BFS exploration graph starting from vertex A looks like this:



They are different because the order in which the nodes are explored are different.

Recursive DFS:

```
DFS(v):
    if v.explored == false then:
        v.explored = true
        for all neighbors w of v:
            if w.explored == false:
                DFS(w)
```

Runtime:

- For each vertex, we make a DFS call at most once. We do not call DFS on a vertex that has already been explored. Total number of neighbors we are looking at is $O(|E|)$. For an undirected graph, it will be $2|E|$, for a directed graph it will be $|E|$. Number of DFS calls is the number of vertices, $|V|$. Thus, the total runtime is $O(|V| + |E|)$.

3 DFS with Start and End Time

```
INPUT: G = (V, E)

Initially, all the vertices are marked as not explored (v.explored = false)
i = 0
for all vertex v in G:
    if v.explored == false:
        DFS(v)

DFS(v):
    v.start = i
    i++
    v.explored = true
    for all neighbors w of v:
        if w.explored == false:
            w.parent = v
```

```
            DFS(w)
    v.end = i
    i++
```

- **Start time**: time when a recursive call was made.
- **End time**: time when a recursive call was ended.
- **Parent**: the neighbor relation used to explore the graph. If w is explored from u, then $w.parent = u$.

Types of Edges in a DFS Exploration Tree

- Tree Edge: the edges in the DFS exploration tree through the parent relationship.
- Back Edge: For $u \to v$ to be a back edge, $u.start > v.start$ and $u.end < v.end$.
- Forward Edge: For $u \to v$ to be a forward edge, $u.start < v.start$ and $u.end > v.end$.
- Cross Edge: All the other edges that are not tree edges, back edges, or forward edges, are cross edges. For $u \to v$ to be a cross edge, $u.end > v.end$. u is currently being explored, whereas v has already been explored.
- (**Note**: the above types of edges are possible edges that can be encountered during DFS exploration of a directed graph. In the case of undirected graph, we will have Tree Edges, and only Back Edges.)

4 Detecting Back Edges

A directed graph contains a cycle if and only if it contains a back edge. Thus, to check whether a given directed graph contains a cycle, we simply have to check whether it has a back edge.

We can modify DFS to detect back edges in a graph represented by an adjacency list `adj`.

```
bool find_cycle(adj)
  for (i = 1 to n)
    visited[i] = false;
    stack[i] = false;

  // Loop through all the vertices to make sure that
  // if there are multiple DFS trees, then we check all of them.
  for (i = 1 to n)
    if (visited[i] == false and dfs(adj, i, visited, stack) == true)
      return true;

  return false;
```

Here, we slightly modify the standard recursive version of DFS to keep track of the current recursion stack, which is essentially the path being traced. If we reach a node with an outgoing edge that points back to a node already in the recursion stack, then there is a cycle in the graph.

```
bool dfs(adj, i, visited, stack)
  if (visited[i] == true)
    return false;

  visited[i] = true;
  stack[i] = true;
  for (j in adj[i])
```

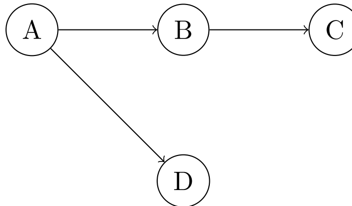
```
    if (stack[j] == true)
      return true;
    if (visited[j] == false and DFS(adj, j, visited, stack) == true)
      return true;

  // Unmark the node from the recursion stack since it is no longer
  // part of the path being traced.
  stack[i] = false;
  return false;
```

The time complexity of `find_cycle` is $O(|V| + |E|)$.

5 Topological Ordering

Given a directed graph with edges that represent a dependency relation between the nodes, we want to find an ordering where the dependent nodes always appear after the nodes that they are dependent on. If there is an edge $u \to v$, then v is dependent on u.

For example,



One topological ordering for the above graph would be A, B, D, C. (**Note**: there are other possible topological orderings as well). The ordering should ensure that a dependent task cannot start until all the tasks it depends on are completed. For example, A, C, B, D is not a valid topological ordering since we cannot do C without first finish doing B.

For a topological ordering to exist, the given dependency graph must not contain a cycle.

Observation 1: The node with higher end time does not depend on the node with lower end time. $u.end > v.end \Rightarrow u$ does not depend on v.

Observation 2: The vertex with the largest end time does not have any incoming edges (i.e., it does not depend on any other vertex).

Algorithm for Topological Sort Using DFS

```
Initialize a stack S

DFS_explore(G=(V, E)):
    for all v in V:
        if v.explored == false:
            DFS(v)

DFS(v, S):
    v.explored = true
    for all neighbors w of v
```

```
        if w.explored == false:
            DFS(w)
    add v to stack S

Output vertices starting from the top of the stack S.
```

The vertex with the largest end time will be at the top of the stack (we do not need to keep track of the end times for any vertex; the stack order will determine it). The stack will contain vertices in decreasing order of their end time.

Runtime:

- The runtime of this algorithm will just be the same as the DFS algorithm, which is $O(|V| + |E|)$.