CSCI 2270 – Data Structures

Recitation 10, March 2020

Graph Algorithms

Objectives

1. Depth First Search
2. Breadth First Search
3. Exercise
    a. Challenge 1 (Silver Badge)
    b. Challenge 2 (Gold Badge)

Depth First Search (DFS)

The strategy followed by depth-first search is, as its name implies, to search “deeper” in the graph whenever possible.

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. The idea is to travel as deep as possible from neighbour to neighbour before backtracking. What determines how deep is possible is that you must follow edges, and you don't visit any vertex twice. To do this properly we need to keep track of which vertices have already been visited, plus how we got to where we currently are, so that we can backtrack.
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.

All the nodes on the current path are visited, after which the next path will be selected. We would be using the Adjacency lists to get the neighbours of any vertex.

Here’s a pseudocode of this recursive approach:

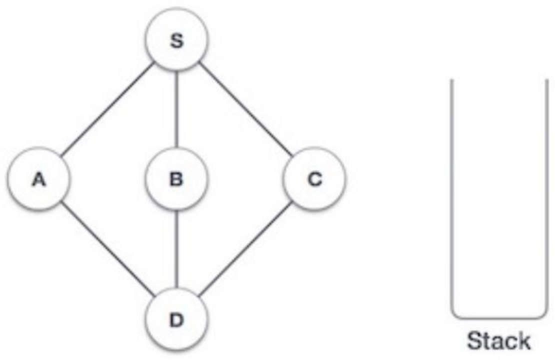
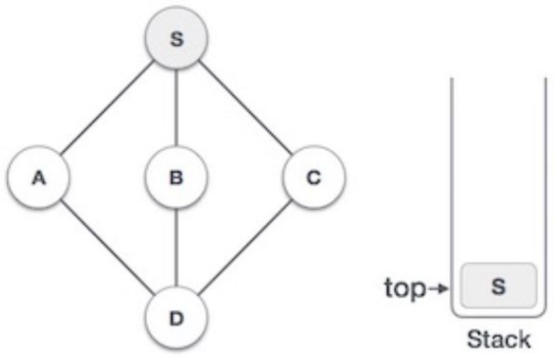
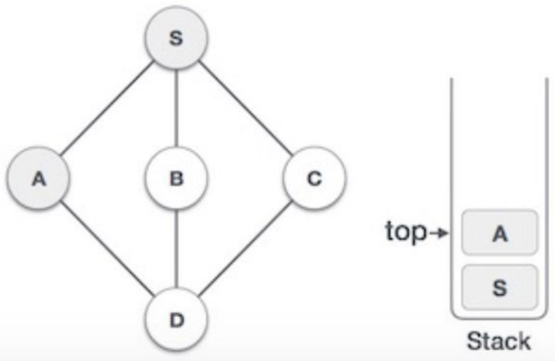
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    for each u ∈ G
        u.visited = false
     for each u ∈ G
```

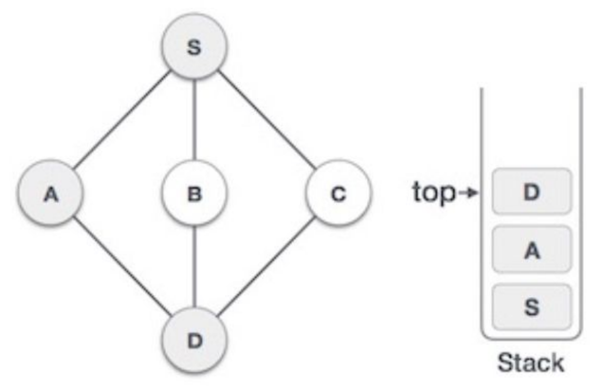
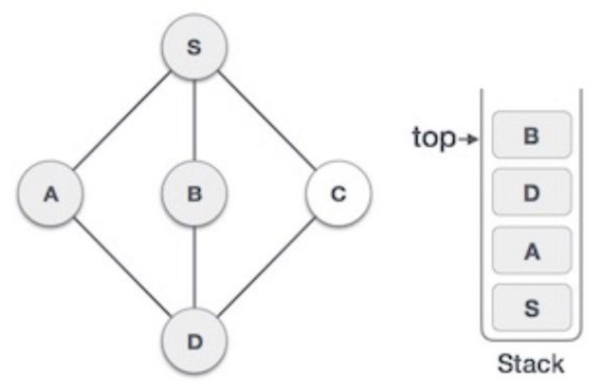
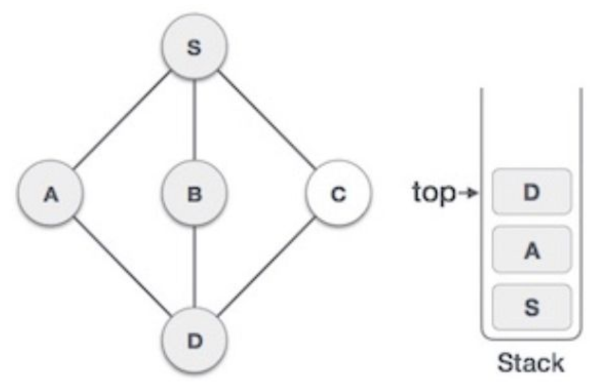
```
            If u.visited==false
          DFS(G, u)
}
```

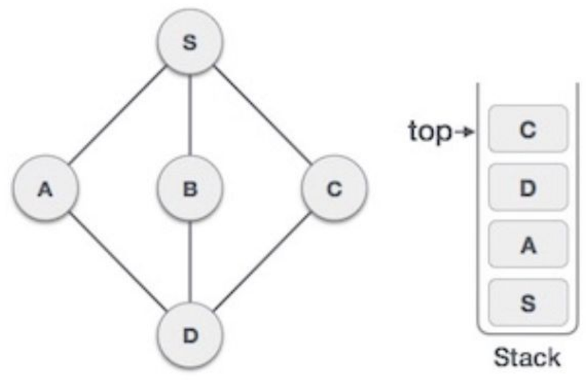
Make sure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

This recursive behaviour (which uses a system stack) can be simulated by an iterative algorithm explicitly using a stack. So, DFS uses a stack to push nodes we mean to visit onto that.

We gave a pseudocode for DFS traversal using recursion and we will be giving a visualization of DFS traversal using stacks to help you understand better. The following will give you the basic idea of Stack implementation.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Applications of DFS

- Finding connected components in an undirected graph
  (a variant of DFS is used for doing the same in directed graphs - Kosaraju algorithm)
- Solving puzzles, such as mazes (DFS helps to reach the goal faster)

Breadth First Search (BFS)

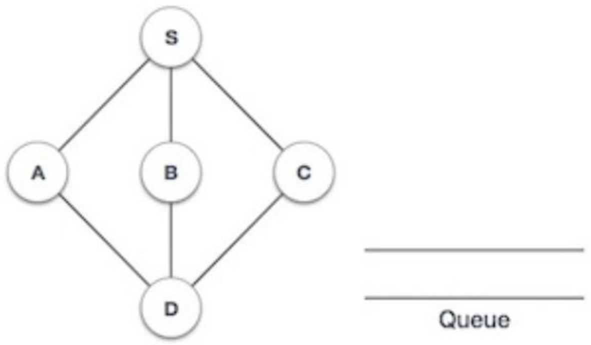
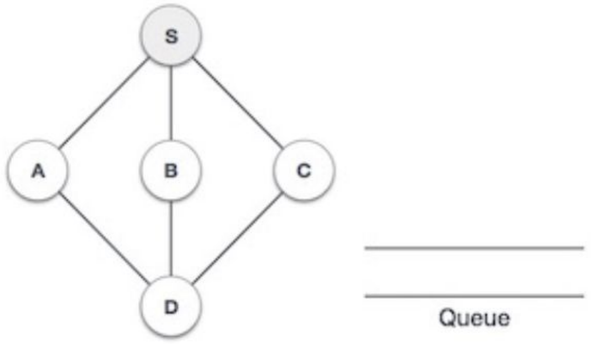
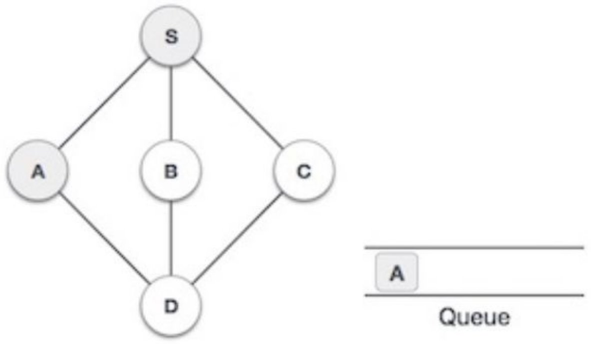
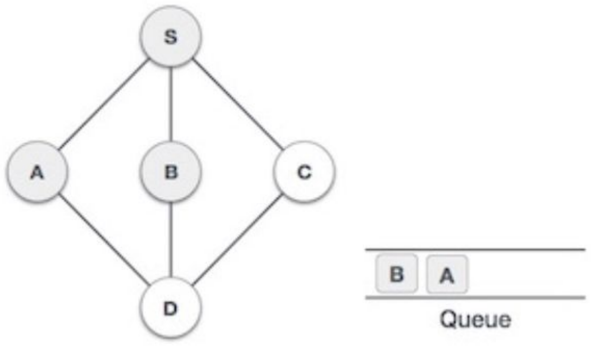
Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

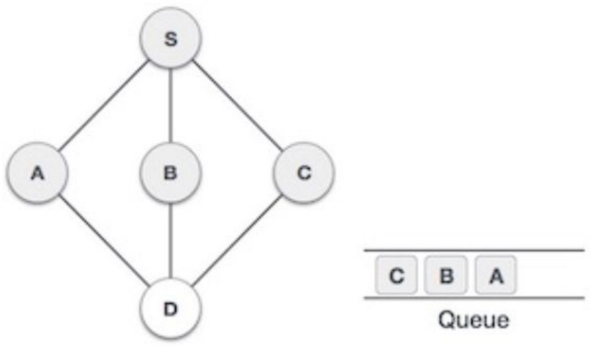
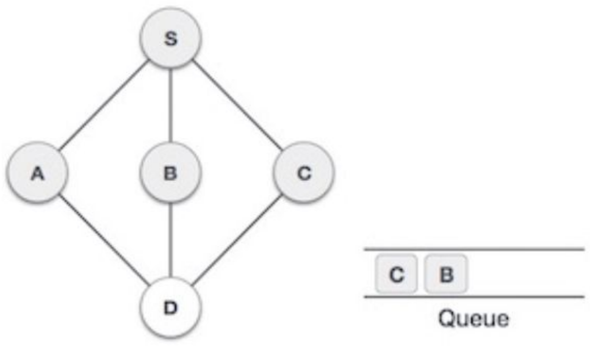
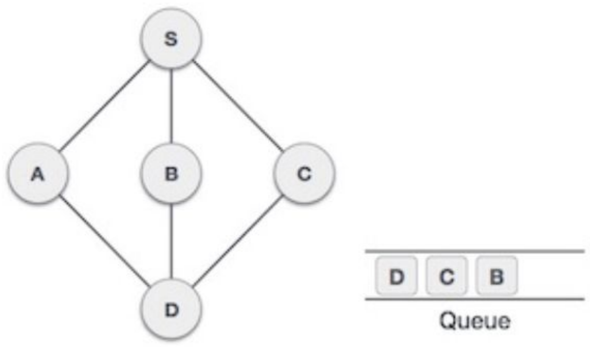
BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1. In the second stage, it visits all vertices at second level. These new vertices are the one which are adjacent to level 1 vertices.

BFS continues this process until all the levels of the graph are completed. Generally queue data structure is used for storing the vertices of a level. As similar to DFS, assume that initially all vertices are marked as unvisited. Vertices that have been processed and removed from the queue are marked visited. We use a queue to represent the visited set as it will keep the vertices in order of when they were first visited.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.

Similarly we have an example for BFS as well using queues.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

Applications of BFS

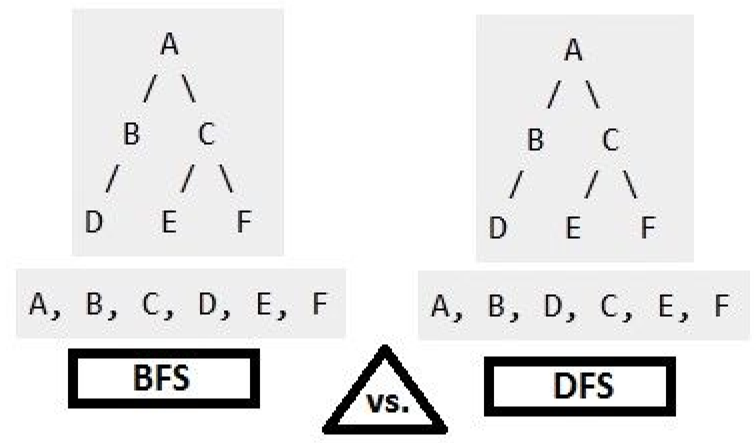
- Finding all connected components in an undirected graph.
- Finding the shortest path between two nodes.

CSCI 2270 – Data Structures

Recitation 10, March 2020

Graph Algorithms

Comparison of BFS vs DFS on a simple graph:



Exercise

A. Silver Badge Problem (Mandatory)

1. Download the Lab10 **zipped** file from Moodle.
2. Follow the TODOs and in Graph.cpp, complete the **findShortestPath()** function.
3. This function finds the length of the shortest path between a source and destination vertex in an undirected and unweighted graph.

B. Gold Badge Problem

1. In Graph.cpp, complete the **printPath()** function after modifying the **findShortestPath()** function so as to print the shortest path after finding it's length.