Course Information and Introduction

Arash Rafiey

January 12, 2016

Course Information

1. **Instructor : Arash Rafiey**
   - Email : arash.rafiey@indstate.edu
   - Office : Root Hall A-127
   - Office Hours : Thursdays 2:30 pm to 3:30 pm in my office (A-127)
2. **Course Home Page :**
   http://cs.indstate.edu/~arash/algo658.html

Course Information

Objective : Introducing concepts and problem-solving techniques that are used in the design and analysis of efficient algorithms.
How ? : By studying various algorithms and data structures.

1. Graph search algorithms (Greedy) BFS, DFS, Dijkstra's, Kruskal's, and Prim's (Review)
2. Divide and conquer algorithms (solving recurrences)
3. Dynamic programming algorithms
4. Graph search algorithms
5. Network Flow algorithms and matching
6. NP-completeness
7. Approximation algorithms
8. Fixed parametrized algorithms
9. Randomized algorithms
10. Heuristic searches algorithms
11. Some Number theory algorithms

Course Information

Textbooks :

Algorithm Design, J. Kleinberg, É. Tardos, Addison Wesley, 2006.

Introduction to Algorithms (3rd Edition), T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, MIT Press, 2009.

Grading

1. 2 Homework assignments and 2 Presentation (each 10 %)
2. Midterm (25 %)
3. Final (35 %)

Graph (Basic Definition)

Graph : Represents a way of encoding pairwise relationships among a set of objects.

Graph G consists of a collection V of *nodes* and a collection E of *edges*, each of which joins two of the nodes.

$E(G) \subseteq \{\{u, v\} | u, v \in V(G)\}$

If the relation is not symmetric (directed graph) we have $E(G) \subseteq \{(u, v) | u, v \in V(G)\}$.

1. Transportation networks (airline carrier, airports as node and direct flights as edges (direct edge).
2. Communication networks (a collection of computers as nodes and the physical link between them as edges).
3. Information networks (World Wide Web can be viewed as directed graph, the Web pages are nodes and the hyperlink between the pages are directed edges).
4. Social Network (People are nodes and friendship is an edge).

Let G be a graph. For simplicity instead of edge $\{u, v\}$ we write edge uv.

Two vertices u and v are called *adjacent* if uv is an edge of G.

We say v is a *neighbor* of u if uv is an edge of G.

Let $G = (V, E)$ be an undirected graph.

A path is a sequence P of nodes $v_0, v_1, \ldots, v_{k-2}, v_{k-1}$ with the property that $v_i v_{i+1}$ is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of P is $k - 1$.

Let $G = (V, E)$ be an undirected graph.

A path is a sequence P of nodes $v_0, v_1, \ldots, v_{k-2}, v_{k-1}$ with the property that $v_i v_{i+1}$ is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of P is $k-1$.

A cycle is a sequence C of nodes $v_1, v_2, \ldots, v_{k-1}, v_k, v_1$ with the property that $v_i v_{i+1}$ (sum module k) is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of C is k.

In the directed path and directed cycle, each pair of consecutive nodes (v_i, v_{i+1}) is a directed edge, i.e. $v_i v_{i+1}$ is an *arc* .

A walk is an alternating sequence of nodes and edges, beginning and ending with a node.

A walk is *closed* if its first and last nodes are the same.

A trail is a walk in which all the edges are distinct.

A path is a *simple* walk (no two nodes repeated).

Connectivity

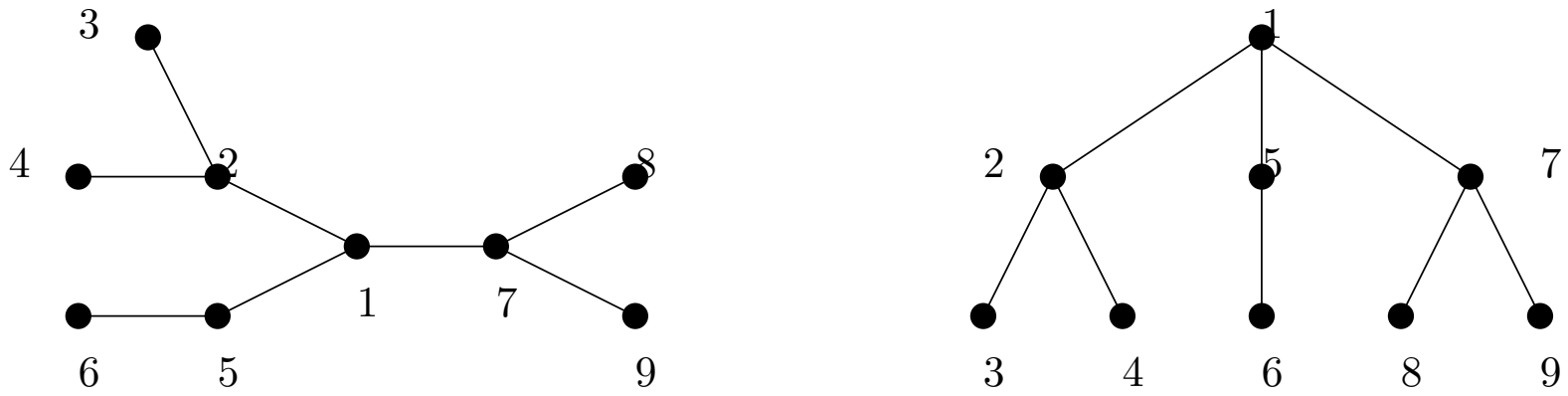
We say (undirected) graph G is connected if, for every pair of nodes u and v, there is a path from u to v.

We say digraph D is strongly connected if for every pair of nodes u, v there is a directed path from u to v and there is a directed path from v to u.

A directed cycle is a strong digraph.

Distance between two nodes u, v, $d(u, v)$ is the length of the shortest path between u and v.

A tree is a connected graph that has no cycle. A tree with n nodes has exactly $n - 1$ edges.
We usually consider a node as a root and the rest of the nodes hag downward from the root. The nodes that are at the end (have only one neighbor) are called leaves.



Two drawings of the same tree. On the right, the tree is rooted at node 1

s-t connectivity and Graph Traversal

Given a graph G and two nodes s, t. The $s - t$ connectivity problem asks whether there is a path from s to t.

Breadth-First Search (BFS). Start with node s and set $L_0 = s$.

At step i let L_i be the set of nodes that are not in any of

$L_0, L_1, \ldots, L_{i-1}$ and have a neighbor in L_{i-1}.

Lemma

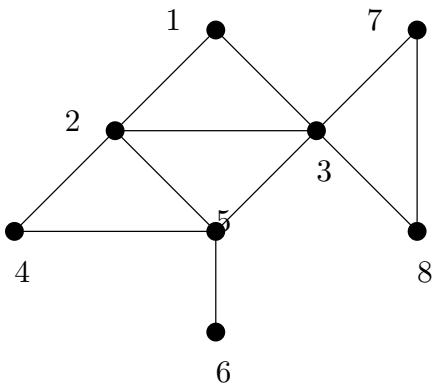
L_j is the set of nodes that are at distance exactly j from s.

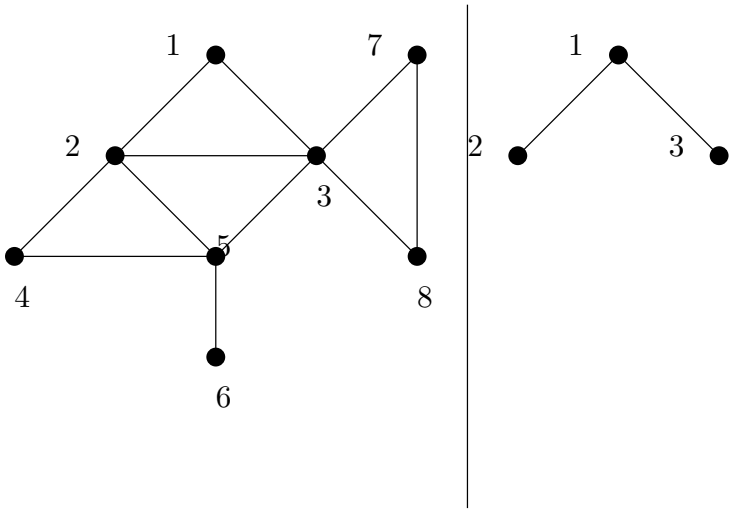
The BFS algorithm creates a tree with root s.

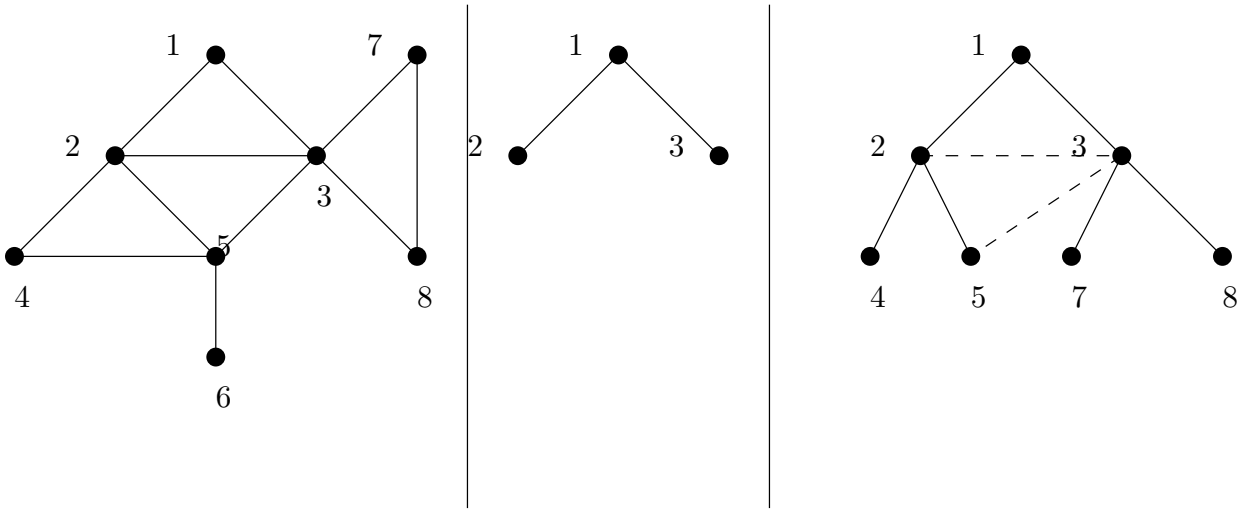
Once a node v is discovered by BFS algorithm we put an edge from v to all the nodes u that have not been considered. This way v is set to be the father of node u.

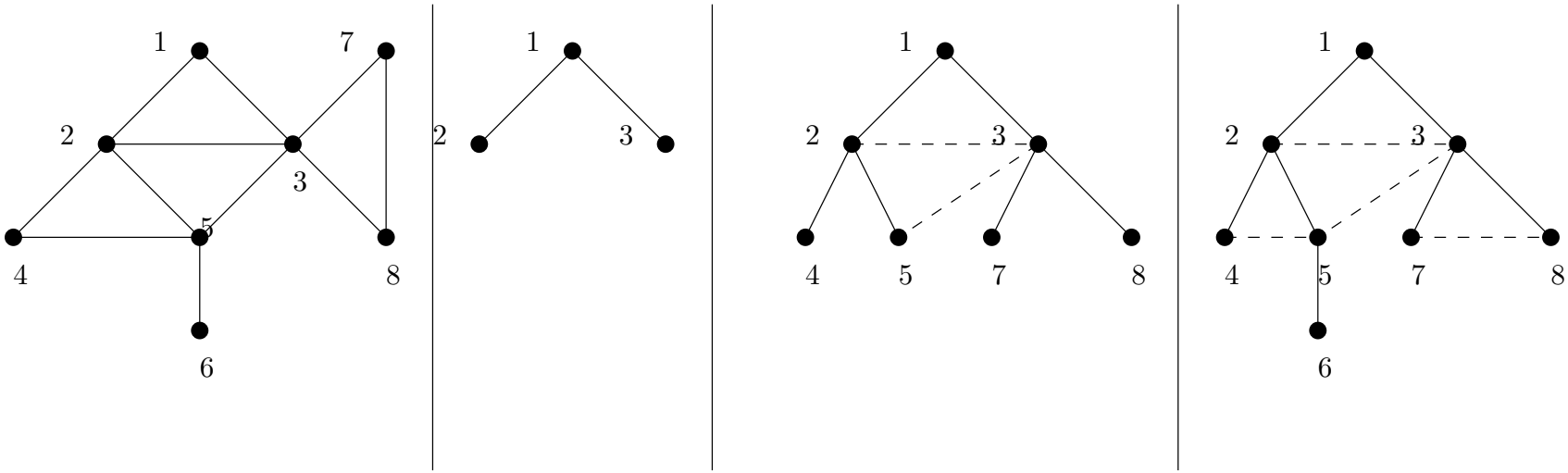
BFS (s)

1. Set Discover[s]=true and Discover[v]=false for all other v
2. Set $L[0] = \{s\}$
3. Set layer counter i=0
4. Set $T = \emptyset$ // T is the tree to build
4. While $L[i]$ is not empty
5. Initialize empty set $L[i+1]$
6. For each node $u \in L[i]$
7. Consider each edge uv
8. If $Discover[v] =$ false then
9. Set $Discover[v] =$ true
10. Add edge uv to T
11. Add v to the list $L[i+1]$, increase i by one


1
7
2
3
5
4
8
6


1
7
2
3
5
4
8
6
2
1
3


1
7
2
3
5
4
8
6
1
2
3
1
2
3
4
5
7
8


1
7
2
3
5
4
8
6
1
2
3
1
2
3
4
5
7
8
1
2
3
4
5
7
8
6

BFS running time

1) If we represent the graph G by adjacency matrix then the running time of BFS algorithm is $O(n^2)$, where n is the number of nodes.

2) If we represent the graph G by link lists then the running time of BFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.

BFS Algorithm

BFS (s)

1. Set Discover[s]=true and Discover[v]=false for all other v
2. Set $L[0] = \{s\}$
3. Set layer counter i=0
4. Set $T = \emptyset$ // T is the tree to build
4. While $L[i]$ is not empty
5. Initialize empty set $L[i+1]$
6. For each node $u \in L[i]$
7. Consider each edge uv
8. If $Discover[v] =$ false then
9. Set $Discover[v] =$ true
10. Add edge uv to T
11. Add v to the list $L[i+1]$, $i = i+1$

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)
Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.
If at the end there is an edge whose end points receive the same color G is not bipartite.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)
Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.
If at the end there is an edge whose end points receive the same color G is not bipartite.

This is essentially is the BFS algorithm. We color the nodes in L_0 by red and the nodes in L_1 by blue and the nodes in L_2 by red and so on.

Next we read each edge uv of G. If both u, v have the same color then G is not bipartite. Otherwise G is bipartite.

Lemma

Let G be a connected graph, and let $L_0, L_1, L_2, \ldots, L_k$ be the layers produced by BFS algorithm starting at node s.

(i) *There is no edge of G joining two nodes of the same layer. In this case G is bipartite and $L_0, L_2, \ldots, L_{2i}$ can be colored red and the nodes in odd layers can be colored blue.*

(ii) *There is an edge of G joining two nodes of the same layer. In this case G contains an odd cycle and G is not bipartite.*

Proof :

Suppose (i) happens. In this case the red nodes and blue nodes give a bipartition, and all the edges of G are between the red and blue nodes.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

3) Portion of P, say P' from z to x has length $j - i$ and portion of Q, say Q' from z to y has length $j - i$.

4) By adding xy edges into P', Q' we get a cycle of length $(j - i) + 1 + (j - i)$ which is of odd length.

Depth-First Search (backtracking approach)

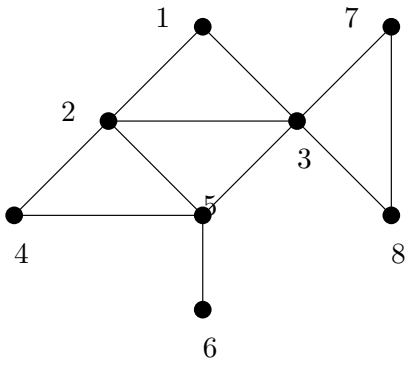
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

Depth-First Search (backtracking approach)

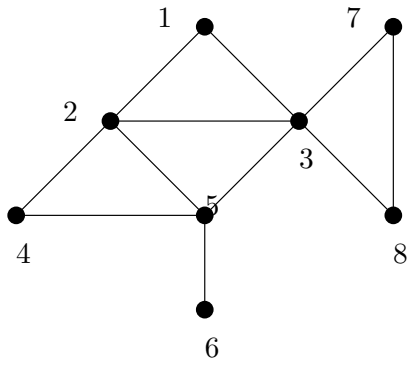
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

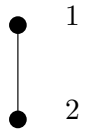
DFS (u)

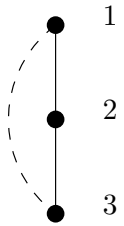
1. Mark u as Explored and add u to R
2. For every edge uv
3. If v is not Explored then call DFS (v)

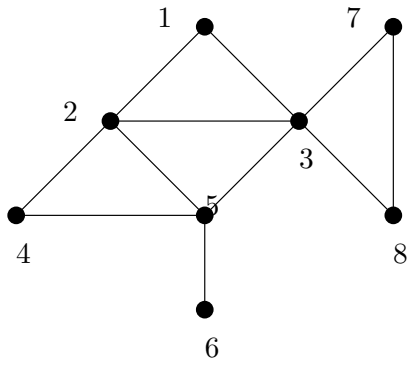

1
7
2
3
5
4
8
6

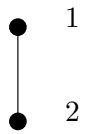

1
2

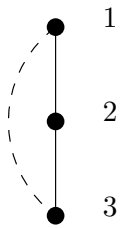

1
7
2
3
5
4
8
6

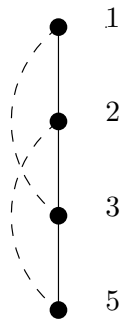

1
2

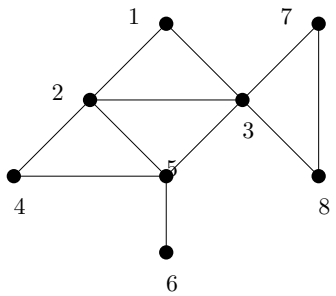

1
2
3

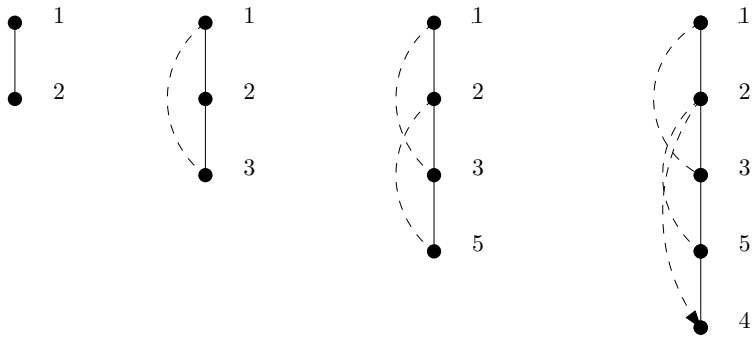

1
7
2
3
5
4
8
6


1
2


1
2
3


1
2
3
5



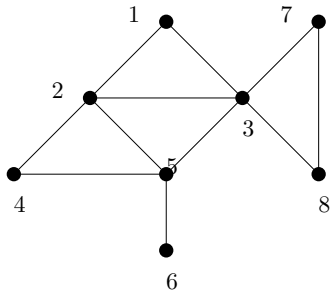


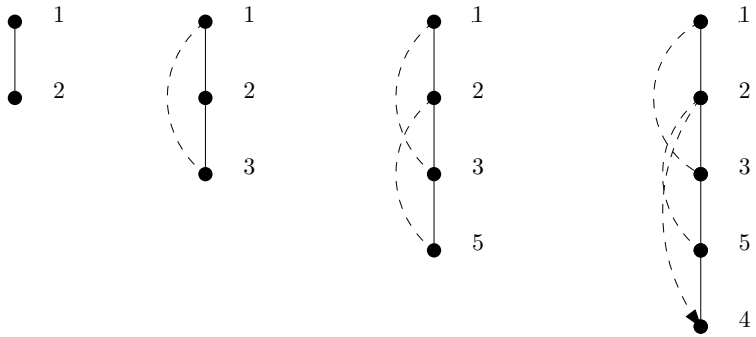
1

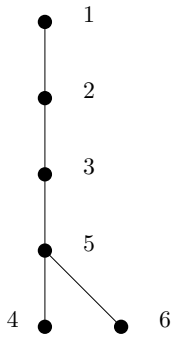
2

3

5





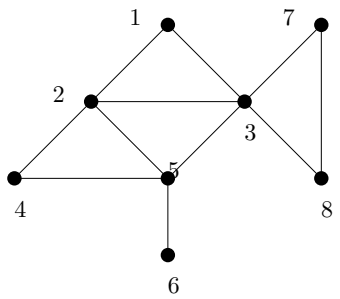


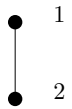
1

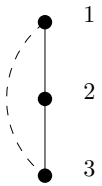
2

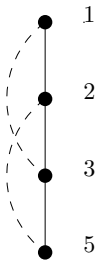
3

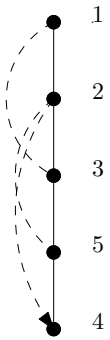
5

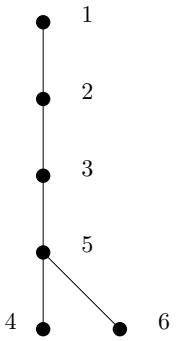

1
7
2
3
5
4
8
6

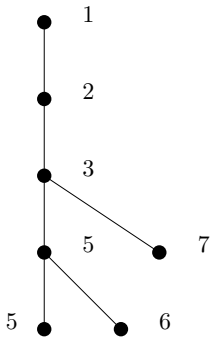

1
2

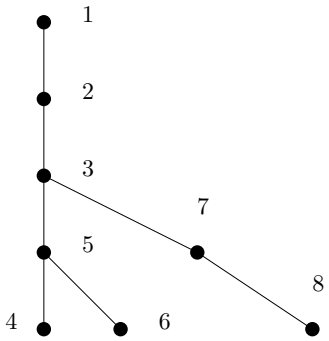

1
2
3


1
2
3
5


1
2
3
5
4


1
2
3
5
4
6


1
2
3
5
7
5
6


1
2
3
7
5
8
4
6

Implementing Depth-First Algorithm using Stack

DFS(s)

1.Initialize S to be a stack with element s only.

2. While S is not empty

3. Take a node u from top of S.

4. If Explored[u] =false then

5. Set Explored[u]=true

6. For every uv edge add v to S.

DFS running time

1) If we represent the graph G by adjacency matrix then the running time of DFS algorithm is $O(n^2)$, where n is the number of nodes.

2) If we represent the graph G by link lists then the running time of DFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.