Algorithms Analysis and Design

Chapter 3

Brute Force and Exhaustive Search
Part 2

Assignment problem

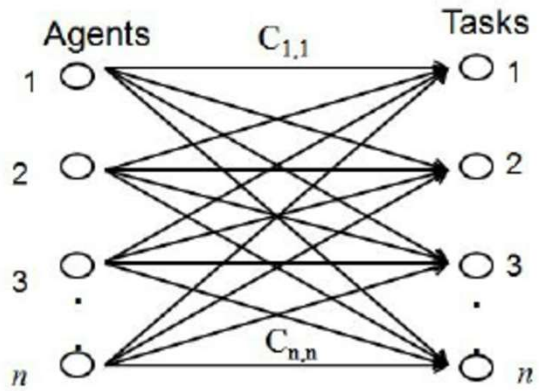
- **Assignment problem definition**: Suppose there are n jobs to be performed and n persons are available for doing these jobs, one person per job [That is, each person is assigned to exactly one job and each job is assigned to exactly one person]. Assume that each person can do each job at a term, though with varying degree of efficiency. The cost of assigning person i to job j is $C[i,j]$.

  The problem is to find an assignment (which job should be assigned to which person) So that **the total cost of performing all jobs is minimum.**

C[1,1]

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Table entries represent the assignment cost



Assignment problem

- **Exhaustive search to solve the assignment problem:**
  - **Generate** all legitimate assignments
  - **Evaluate their costs**
  - select the cheapest one.

How many assignments are there?

$n!$

The exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, . . . , n

$n!$

Assignment problem

- **Example on small instance of the assignment problem**

	Job 1	Job 2	Job 3
Person 1	9	2	7
Person 2	6	4	3
Person 3	5	8	1

n = 3

Cost matrix

$$C = \begin{bmatrix} 9 & 2 & 7 \\ 6 & 4 & 3 \\ 5 & 8 & 1 \end{bmatrix}$$

- **The exhaustive search approach**

1. **Generate** all the permutations of integers 1, 2, *3*
2. **Compute** the total cost of each assignment by summing up the corresponding elements of the cost matrix.
3. Select the one with the smallest sum

Assignment (col.#s)	Total Cost
1, 2, 3	9+4+1=14
1, 3, 2	9+3+8=20
2, 1, 3	2+6+1=**9**
2, 3, 1	2+3+5=10
3, 1, 2	7+6+8=21
3, 2, 1	7+4+5=16

Note: <2, 3, 1> indicates the assignment of
Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 1

Exhaustive Search for Assignment Problem

Since the number of permutations to be considered for the general case of the assignment problem is $n!$,
exhaustive search is **impractical** for all but very small instances of the problem (It is practical for only small instances)

Are there other more efficient algorithms for solving the assignment problem???

there is a much more efficient algorithm for this problem called the ***Hungarian method***

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time **only on very small instances**

- In some cases, there are much better alternatives!
  - shortest paths
  - minimum spanning tree
  - assignment problem

- In many cases, exhaustive search or its variation is the only known way to get exact solution.

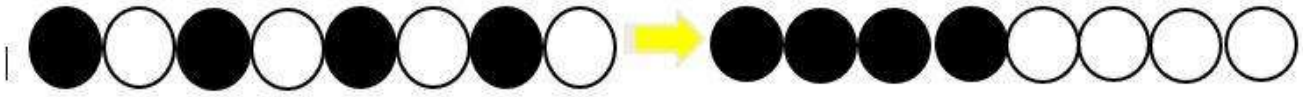
Exercises

Exercise 1

- **The Alternating Disk Problem**

You have a row of $2n$ disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on (as shown in the figure). You want to get all the dark disks to the left-hand end, and all the light disks to the right-hand end. The only moves you are allowed to make are those that interchange the positions of two neighboring disks.

Design an algorithm for solving this puzzle and **determine the number of moves it takes**.



Input: a positive integer n and a list of 2n disks of alternating colors dark-light, starting with dark

Output: a list of 2n disks, the first n disks are dark, the next n disks are light, and an integer m representing the number of swaps to move the dark ones before the light ones.

Exercise 2

You are given an array of 0s and 1s in random order. Design an efficient linear-time algorithm to seperate 0s on the left side and 1s on the right side of the array. You should traverse the array only once.

Example:

- Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
- Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Exercise 3

There are n stacks of n identical-looking coins. All of the coins in one of these stacks are counterfeit, while all the coins in the other stacks are genuine. Every genuine coin weighs 10 grams; every fake weighs 11 grams. You have an analytical scale that can determine the exact weight of any number of coins.

a. Devise a brute-force algorithm to identify the stack with the fake coins and determine its worst-case efficiency class.

b. What is the minimum number of weighings needed to identify the stack with the fake coins?

Exercise 4

Based on the bubble sort algorithm discussed in the lecture:

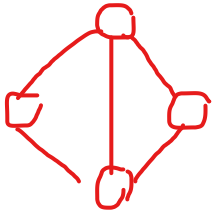
a. Prove that if bubble sort makes no exchanges on its pass through a list, the list is sorted and the algorithm can be stopped.

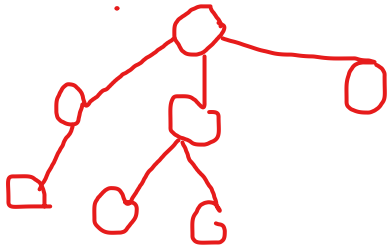
b. Write pseudocode of the method that incorporates this improvement.

c. Prove that the best-case efficiency of the improved version is linear.

d. Prove that the worst-case efficiency of the improved version is quadratic.

Graph Traversal Algorithms



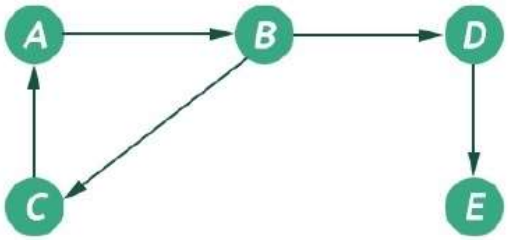


Overview of graph terminology

- Graph is a non-linear data structure.
- Graph consists of some **nodes** (vertices) and their connected **edges**.
- The edges may be director or undirected.
- This graph can be represented as **G(V, E)**
  - V: a finite set of nodes.
  - E: a set of edges.
- Example:
- The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})

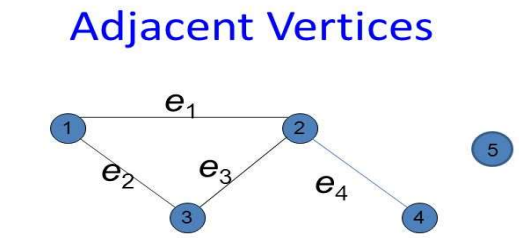
See Chapter 1

vertices Edges



Overview of graph terminology

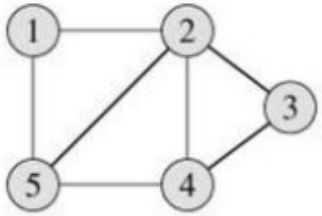
- The vertices in a graph are **adjacent** if there is an edge connecting the vertices.



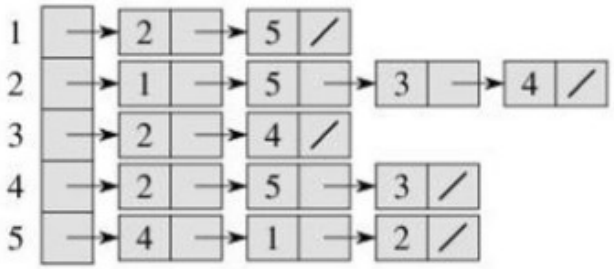
- Two vertices are on a **path** if there is a sequences of vertices beginning with the first one and ending with the second one.
- **Weighted** graphs have values associated with edges.

Graph representation - undirected

See Chapter 1



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

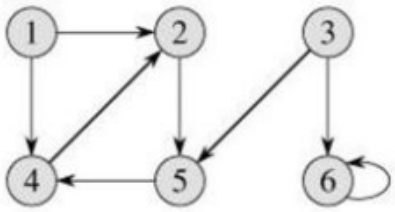
(c)

graph

Adjacency list

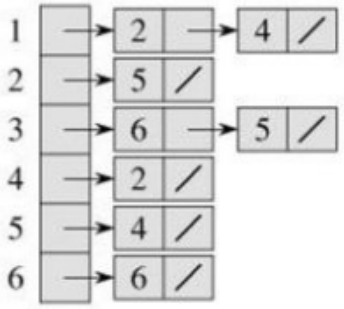
Adjacency matrix

Graph representation - directed



(a)

graph



(b)

Adjacency list

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Adjacency matrix

Adjacency list representation is usually preferred since it is more efficient in representing **sparse** graphs (graphs for which |E| is much less than $|V|^2$

Graph Traversing Algorithms

- Graph Traversal (also known as **graph search**) refers to the process of visiting (checking and/or updating) each vertex in a graph.
- Graph traversal process typically begins with a start vertex and attempt to visit the remaining vertices from there.
- Such traversals are **classified by the order in which the vertices are visited**.
- Many graph applications need to visit the vertices of a graph in some specific order.
- Traversing algorithms works on both **directed and undirected graphs**

Graph Traversing Algorithms

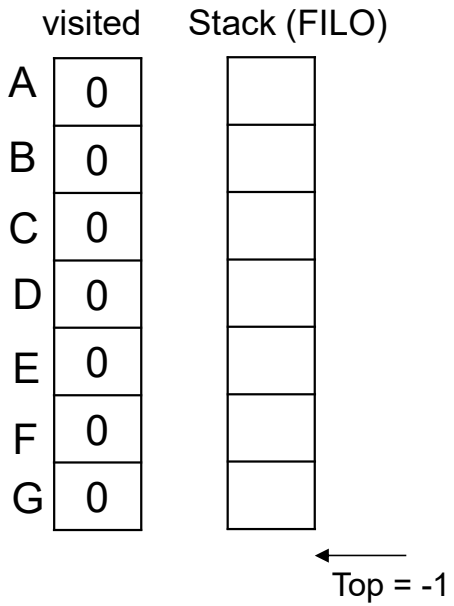
- Graph traversal algorithms:
  - Depth-first search (**DFS**)
  - Breadth-first search (**BFS**)
- These algorithms have proved to be very useful for many applications
  - Investigate the connectivity of the graph.
  - Investigate cycle presence.
  - Finding path between two given vertices.
  - Shortest path and minimum spanning tree.
  - Other applications ……

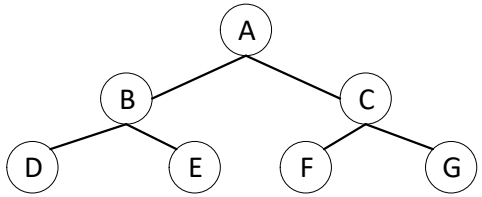
Depth-First Search

- Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.
- It searches "**deeper**" the graph when possible
- Uses a **stack** to trace the operation of depth-first search
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex
- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

Depth-First Search

- Example: Start at vertex A





In our examples, we always break ties by the alphabetical order of the vertices.

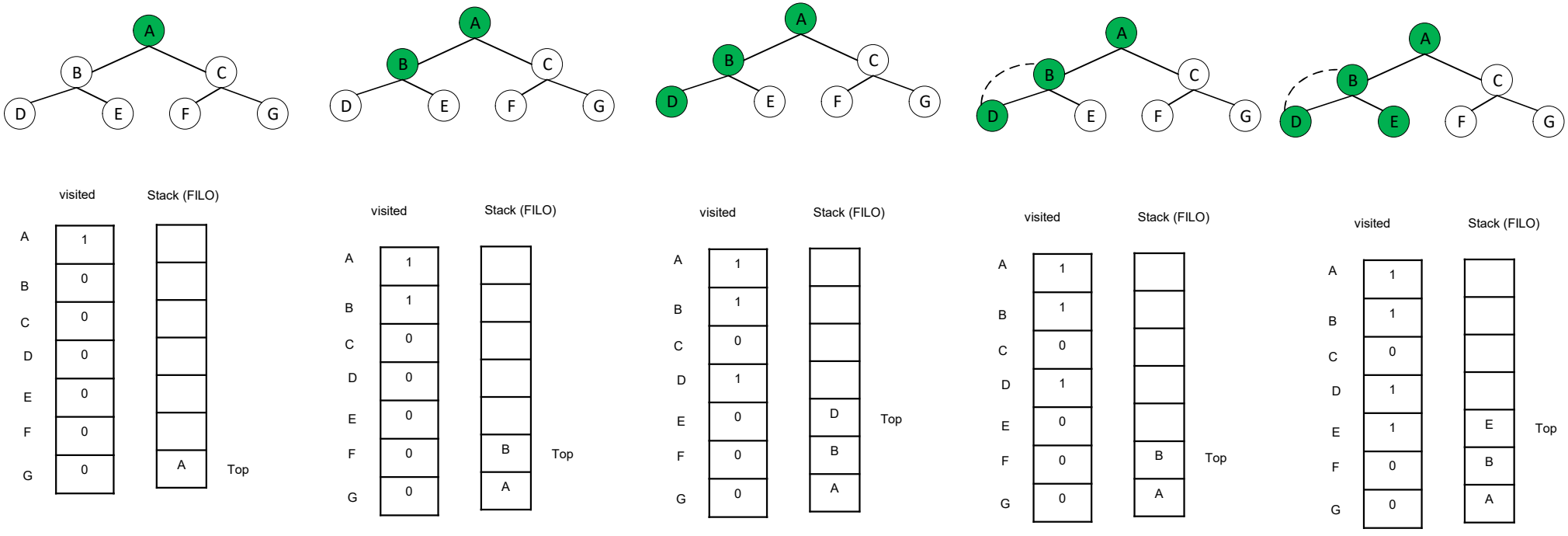
a **dead end**—a vertex with no adjacent unvisited vertices

At a dead end, the algorithm backs up one edge to the vertexit came from and tries to continue visiting unvisited vertices from there

Output:

Depth-First Search

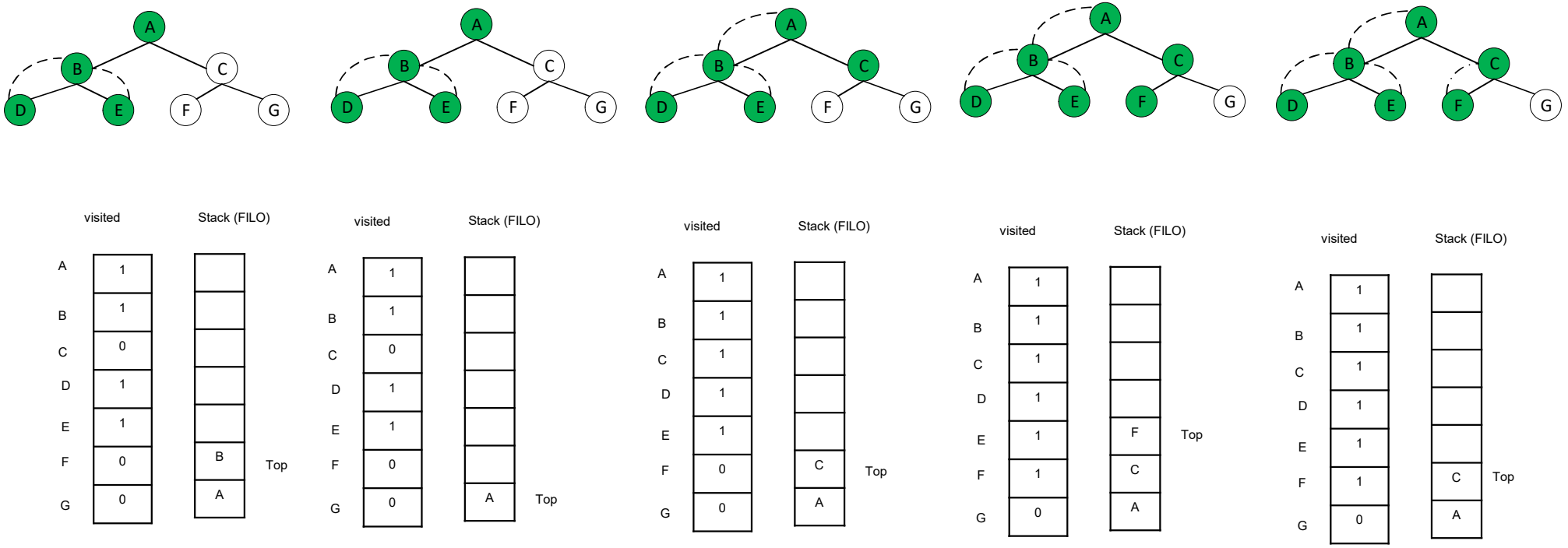
- Example: Start at vertex A



Output: A , B , D , E

Depth-First Search

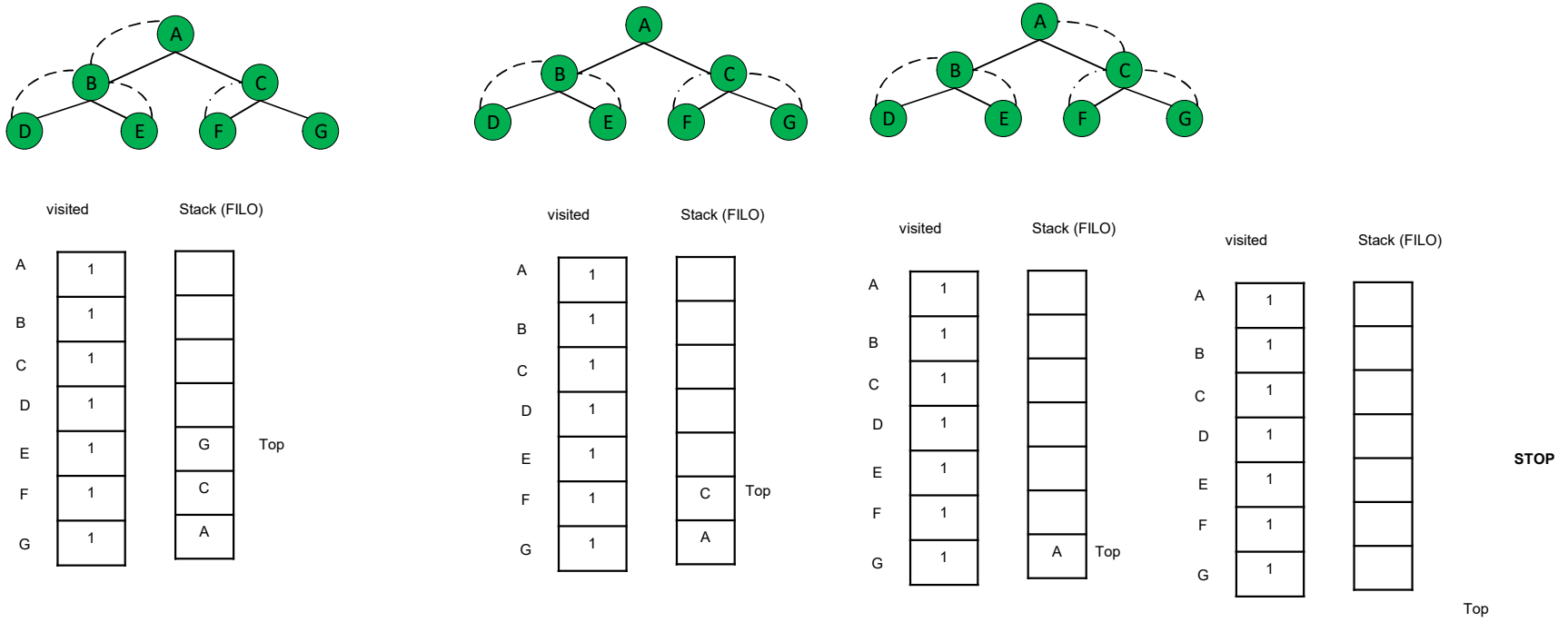
- Example: Start at vertex A



Output: A , B , D , E , C , F

Depth-First Search

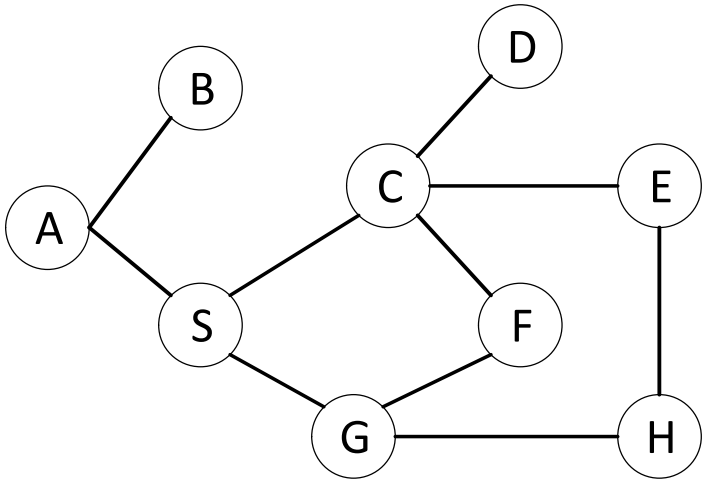
- Example: Start at vertex A



Output: A , B , D , E , C , F, G

Example

- Traverse the following graph by ***depth-first search*** ***starting from vertex A***(Show your work)



Output: A B S C D E H G F

Notes on DFS

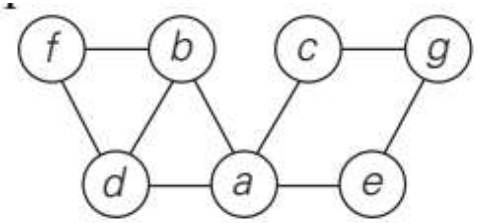
- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$

- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)

- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity
  - finding articulation points and biconnected components
  - searching state-space of problems for solution (AI)

Exercise

- Consider the following graph:



a) Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

b) Starting at vertex ***a*** and resolving ties by the vertex alphabetical order, traverse the graph by ***depth-first search*** and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack).

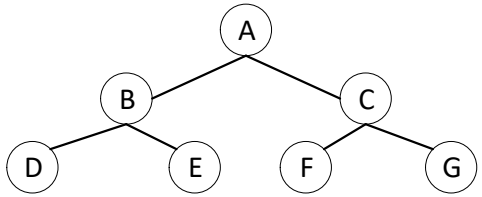
Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a **queue**
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Breadth-first search (BFS)

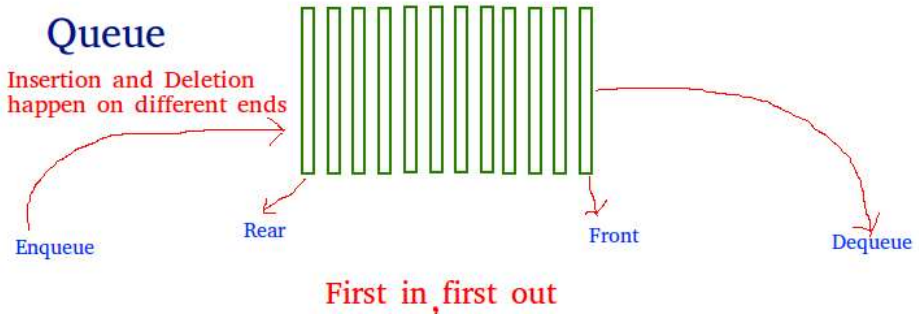
- Example: Start at vertex A

	visited	Queue (FIFO)
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	
G	0	



In our examples, we always break ties by the alphabetical order of the vertices.

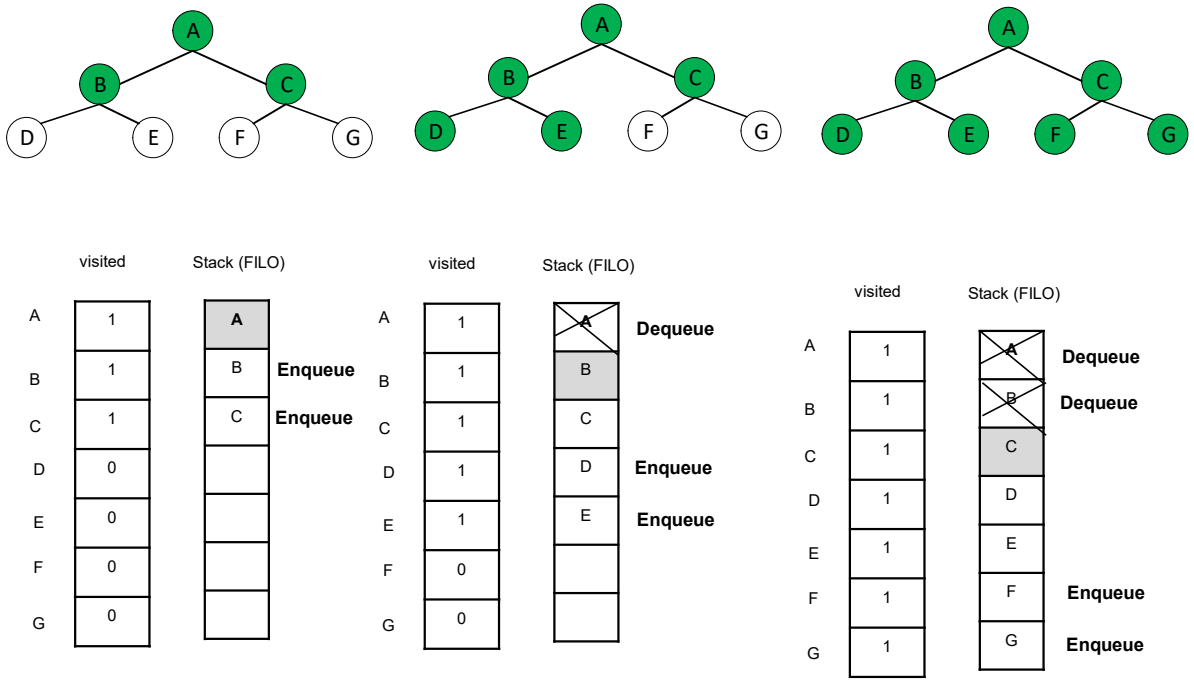
a **dead end**—a vertex with no adjacent unvisited vertices



Output:

Breadth-first Search

- Example: Start at vertex A



Output: A , B , C , D , E , F , G

Notes on BFS

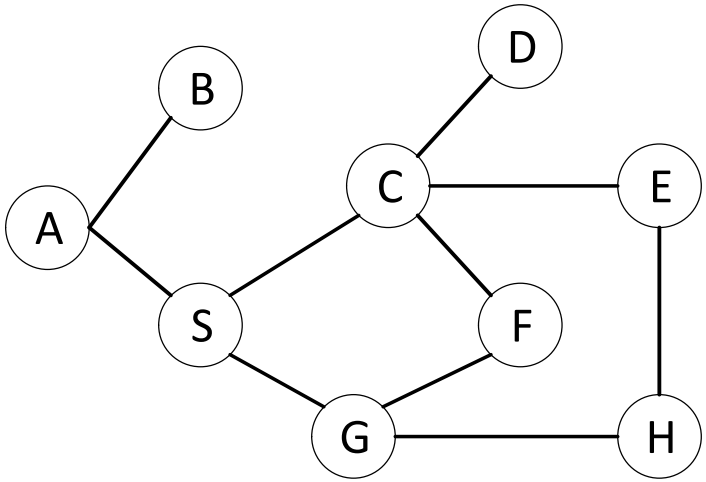
- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(|V|^2)$
  - adjacency lists: $\Theta(|V|+|E|)$

- Yields single ordering of vertices (order added/deleted from queue is the same)

- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges

Exercise

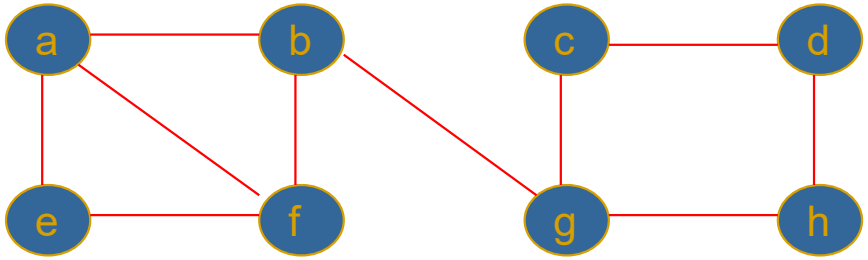
- Traverse the following graph by ***breadth-first search*** ***starting from vertex A***(Show your work)



Output: A B S C G D E F G H

Exercise

- Consider the following graph:



a) Traverse the graph by **breadth-first search**. Start the traversal at vertex ***a*** and resolve ties by the vertex alphabetical order.