CompSci 201, L22: Graphs, DFS

Person in CS: Katherine Johnson

Logistics, Coming up

- P5 Huffman due today, Monday 11/14

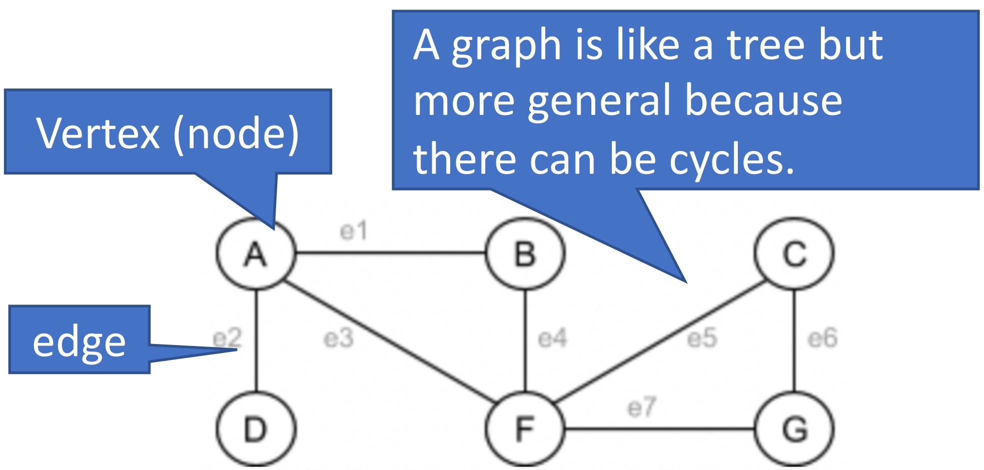
- APT10 (greedy) due Wed., 11/16

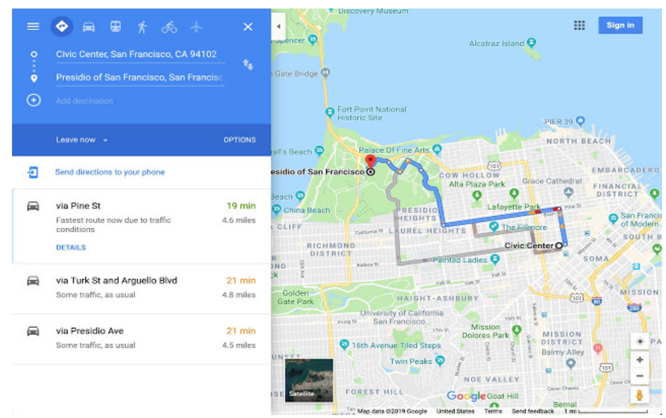
- APT Quiz 2:
  - Release: This Thursday 11/17
  - Complete by: Monday 11/21
  - Quiz, not a hw, no late period

What is a graph?

A ***graph*** is a data structure for representing connections among items, and consists of vertices connected by edges.

- A ***vertex*** (or node) represents an item in a graph.
- An ***edge*** represents a connection between two vertices in a graph.





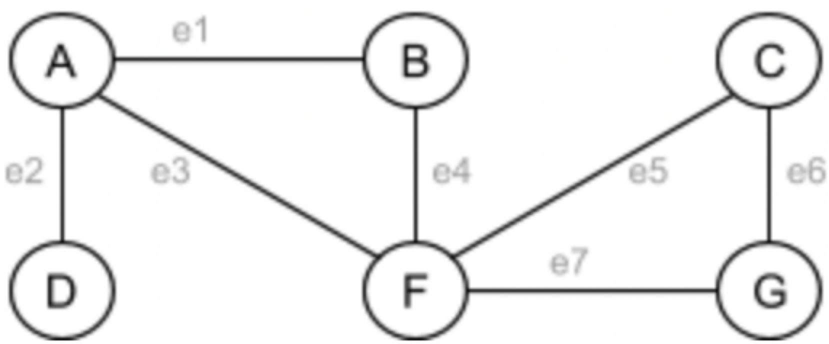
Maps/directions software:
- Vertices ~= intersections
- Edges ~= roads

Zybook chapter 23

Undirected versus directed graphs

Undirected Graph

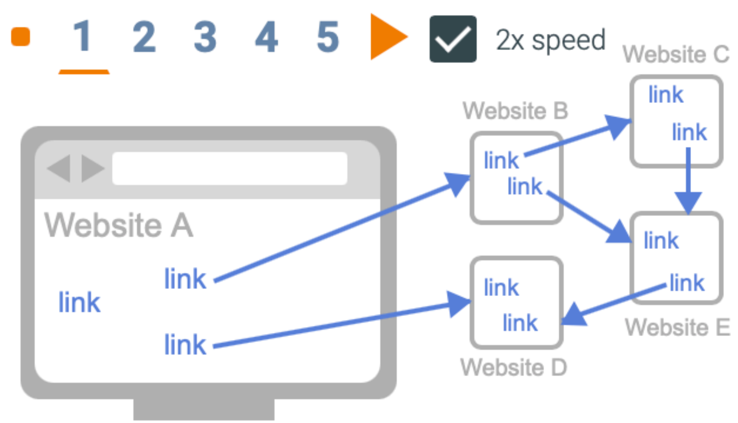
Edges go both ways



Facebook network, most road networks, are undirected

Directed Graph

Edges go one way only

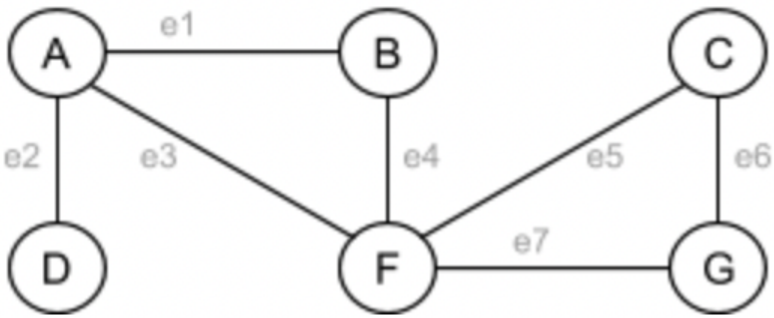


Worldwide web is a directed graph of webpages (nodes) and links (directed edges)

Zybook chapter 23

Simple Graphs and Graph Sizes

- In a **simple** graph, there is at most one (undirected) edge between nodes (or 2 directed).

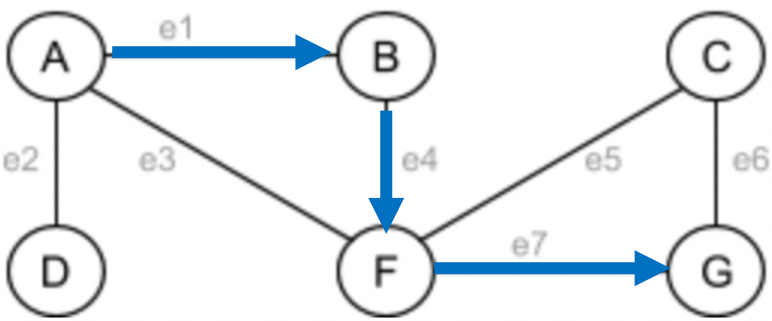


- Usually parameterize the size of the graph as:
  - N (or |V|) = number of vertices/nodes
  - M (or |E|) = number of edges
    - M <= N^2 for a **simple** graph

Zybook chapter 23

Paths in Graphs

- A **path** is a sequence of unique vertices where subsequent nodes are connected by edges
  - (Also commonly defined as a sequence of edges with unique vertices)

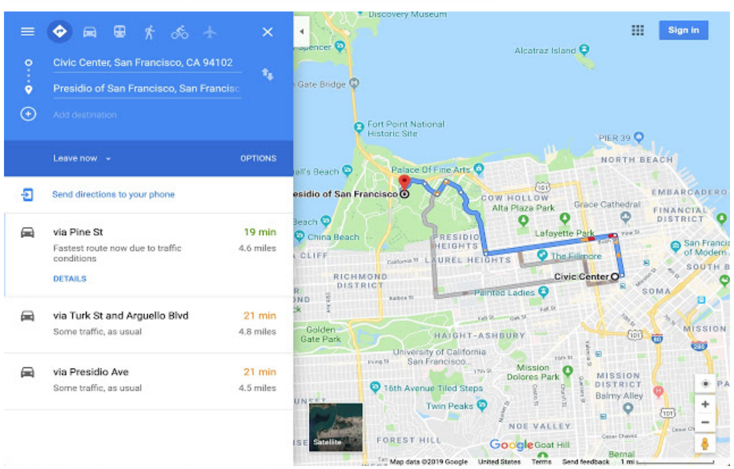


- Example in bold blue: [A, B, F, G].
  - (or [e1, e4, e7])

Pathfinding or Graph Search

Is there a way to get from point A to point B?

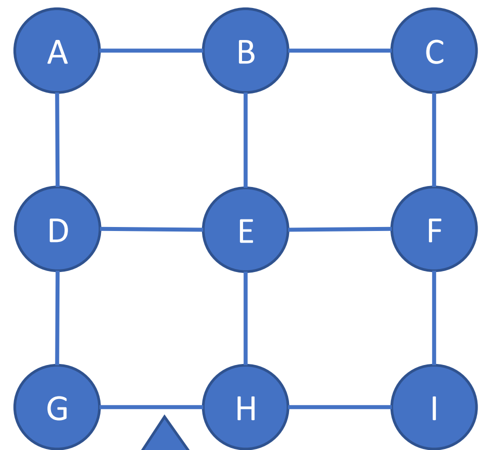
- Maps/directions
- Video games
- Robot motion planning
- Etc.



Recursive Depth-first search (DFS) in Grid Graphs

Two-dimensional grid is simple graph with implicit structure

	0	1	2
0	A	B	C
1	D	E	F
2	G	H	I



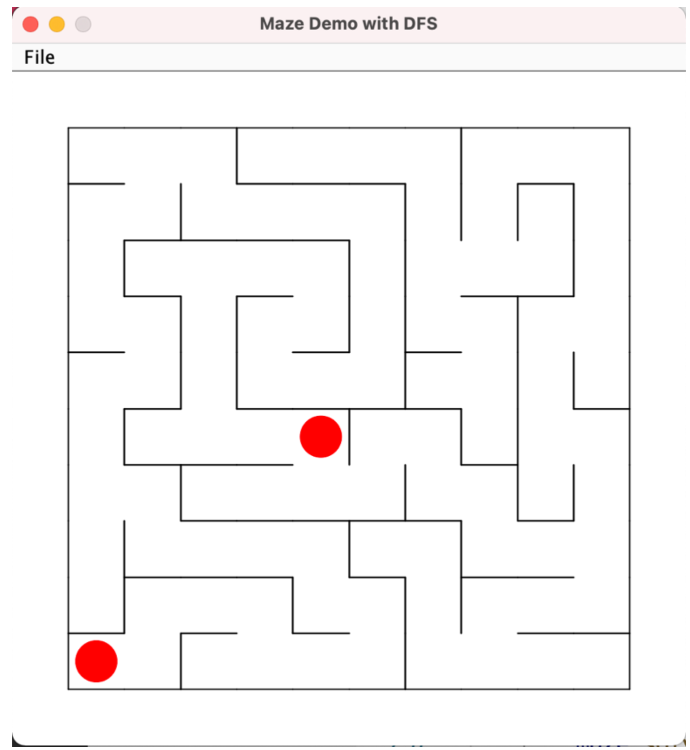
Represent as 2d array, e.g., char[][]
Two nodes adjacent if indices +/- 1

If not all of these edges are present, can represent a maze.

A maze is a grid graph

```java
public class MazeDemo {
    private int mySize;                 // dimension of maze
    private boolean[][] north;          // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
    private boolean[][] west;
```

- Example: ten by ten grid
- Edge = no wall, no edge = wall.
- Look for a path from start (lower left) to middle.

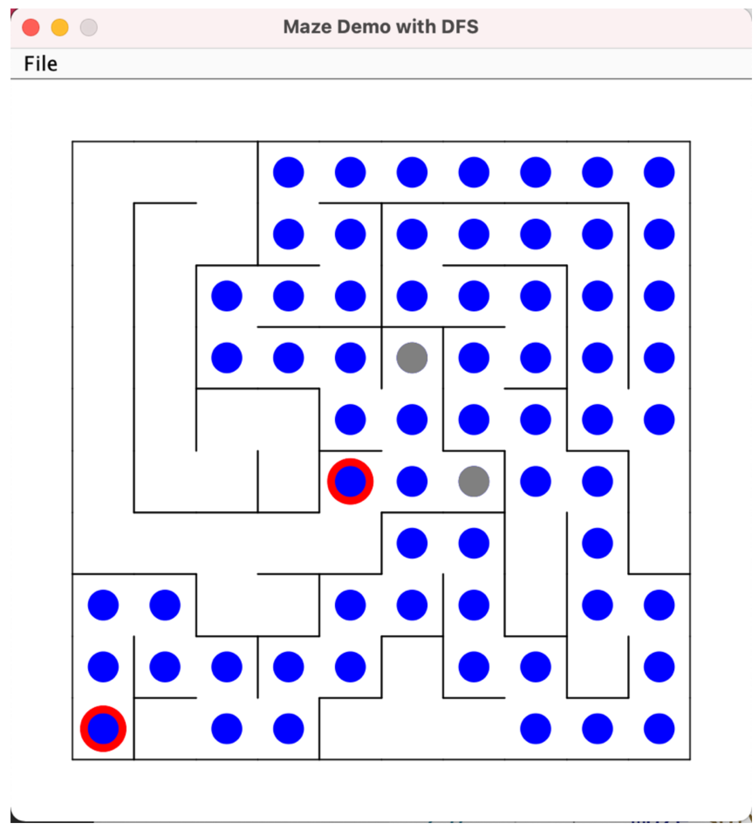


Depth First Search for Solving Maze

Always explore (recurse on) a new (unvisited) adjacent vertex if possible.

If impossible, **backtrack** to the most recent vertex adjacent to an unvisited vertex and continue.

coursework.cs.duke.edu/cs-201-fall-22/maze-demo



How is DFS Graph Traversal like Recursive Tree Traversal?

Tree traversals assumed only two adjacent nodes (children) and no cycles

```java
public void inOrder(TreeNode root) {
    if (root != null) {
        inOrder(root.left);
        System.out.println(root.info);
        inOrder(root.right);
    }
}
```

- Just try recursing on every adjacent vertex?
- Unlike in a tree, there are cycles: How do we avoid infinite recursion?

Base Cases and Visited Set

```
23      private boolean[][] visited;
```

Need to keep track to avoid infinite recursion

```
160  private int solveDFS(int x, int y, int depth) {
161      if (x == 0 || y == 0 || x == mySize +1 || y == mySize +1) return 0;
162      if (visited[x][y]) return 0;
163
164      visited[x][y] = true;
```

- Line 161: Base case: Searching off the grid
- Line 162: Base case: Already explored here

```
171      // reached middle which is goal of maze
172      if (x == mySize /2 && y == mySize /2) {
173          return depth;
174      }
```

- Line 172: Base case: Found the middle!

Recursive case

!north[x][y] → no wall above, can go that way.

y+1 → recurse on node above

Tracking length of path

```
if (!north[x][y]) {
    int d = solveDFS(x, y + 1,depth+1);
    if (d > 0) return d;
}
```

If you found the center, return the path length

3 more symmetric cases for other 3 directions

Runtime complexity for Recursive DFS maze/grid

- Suppose the grid has N = width x height nodes.

- Each node will be recursed on <= 4 times:
  - Has 4 neighbors that could recurse on it,
  - Keep track of visited, we don't recurse from the same neighbor twice.

- Each recursive call is O(1).
- Overall runtime complexity is O(N).

WOTO

Go to duke.is/8vshq

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

Iterative Depth-first search (DFS) in General Graphs

Stack Abstract Data Structure: LIFO List

```java
public static void sdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Stack<String> st = new Stack<>();
    for(String s : strs) {
        st.push(s);
    }
    while (! st.isEmpty()) {
        System.out.println(st.pop());
    }
}
```

```
wonderful
is
compsci
```

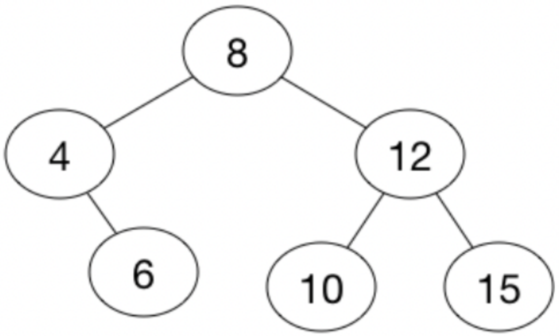
LIFO = Last In First Out

Push: Add element to stack

Pop: Get last element in

Did we really need recursion? preOrder Tree Traversal with Stack

```java
public static void preOrder(TreeNode tree) {
    Stack<TreeNode> myStack = new Stack<>();
    myStack.add(tree);
    while (!myStack.isEmpty()) {
        TreeNode current = myStack.pop();
        if (current != null) {
            System.out.println(current.info)
            myStack.add(current.right);
            myStack.add(current.left);
        }
    }
}
```

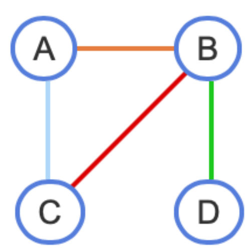


8
4
6
12
10
15

Recursion uses the call stack to keep track of nodes
Could also explicitly use a stack, can do the same for DFS

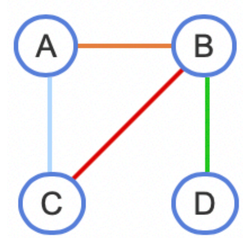
General data structures for graphs: Not necessarily a grid

Adjacency List



Vertices	Adjacent vertices (edges)
A	B C
B	A C D
C	A B
D	B

Adjacency Matrix

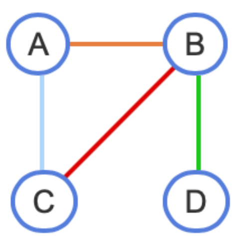


	A	B	C	D
A		1	1	
B	1		1	1
C	1	1		
D		1		

Zybook chapter 23

Efficient Adjacency “List” Using Double Hashing

- HashMap<Vertex, HashSet<Vertex>> aList
  - Vertex type can be Integer, char, String, custom object, …, needs to have good hashCode() and equals().



Vertices	Adjacent vertices (edges)
A	B C
B	A C D
C	A B
D	B

- aList.put(‘A’, new HashSet())
- aList.get(‘A’).add(‘B’)
- aList.get(‘A’).add(‘C’)
- …

O(1) time to check if nodes are connected or get the neighbors of a node (assuming good hashCode)

Graph Search Data Structures

- Have an adjacency list for the graph
- Keep track of visited nodes in a set
- Keep track of the *previous* node: During search, how did I get to this node?

```
public class DFS {
    public static Map<Character, Set<Character>> aList;
    public static Set<Character> visited;
    public static Map<Character, Character> previous;
```

- Example has Character nodes, could be any label for the nodes.
- Storing as instance variables, accessible in methods.

Initializing Iterative DFS

- **Stack** stores nodes we have *visited/discovered*, but not explored from yet.
- Explore from one *current* node at a time.

```
    public static void dfs(char start) {
        Stack<Character> toExplore = new Stack<>();
        char current = start;
        toExplore.add(current);
        visited.add(current);
```

- Stack is LIFO (last-in first-out), so we always explore from the *last node we discovered,* **depth-first**!

Iterative DFS Loop

While there are nodes we have not explored from…

Explore from the most recently discovered node…

Look at all neighbors of current node…

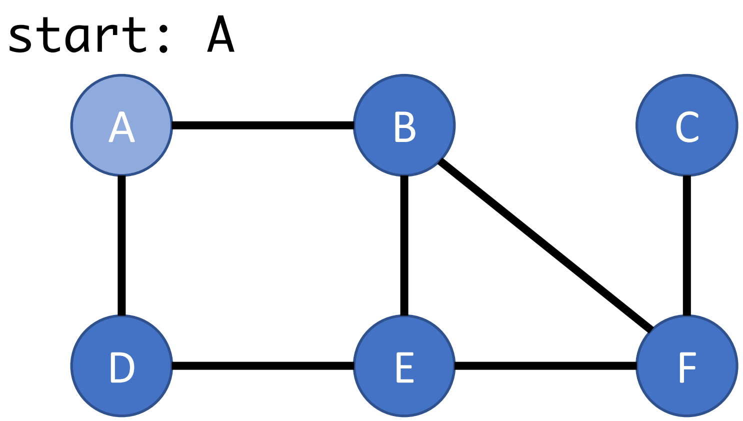
If we haven't seen them before…

```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
            previous.put(neighbor, current);
            visited.add(neighbor);
            toExplore.push(neighbor);
        }
    }
}
```

Then:

1. note how we got here
2. Note we have seen
3. Mark to explore later

Initialize search at A

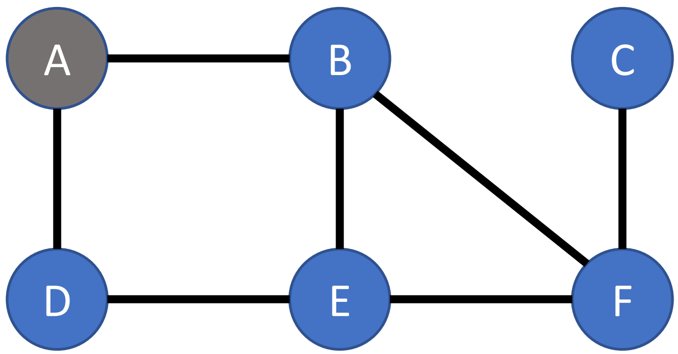


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
A		{A}

Pop A off the stack

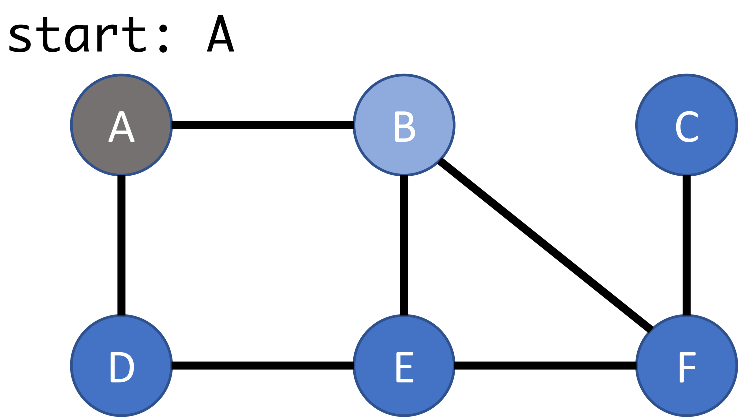
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
		{A}

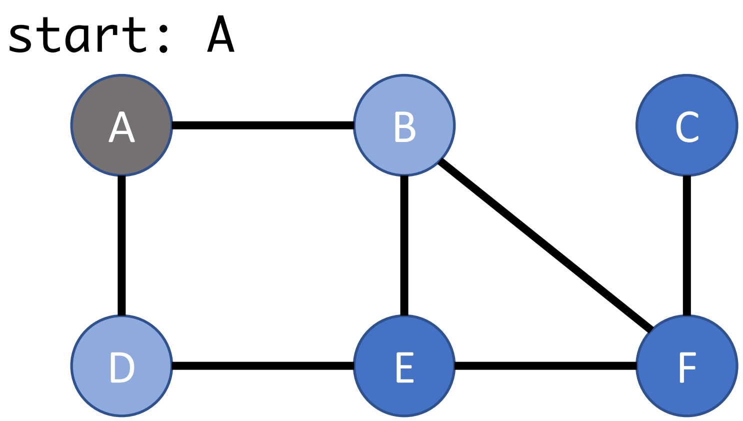
Find B from A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
B	B <- A	{A, B}

Find D from A



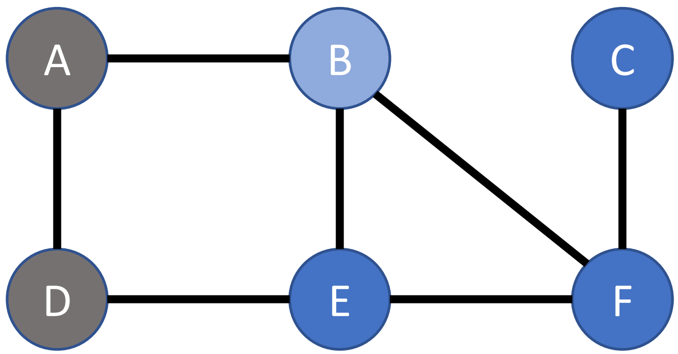
Adjacency List:

```
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]
```

toExplore (stack)	**previous** (map)	**Visited** (set)
D	B <- A	{A, B, D}
B	D <- A	

Pop D off the stack

start: A

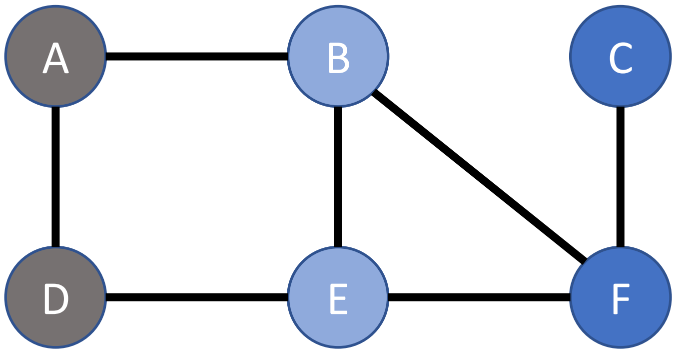


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
B	B <- A D <- A	{A, B, D}

Find E from D

start: A

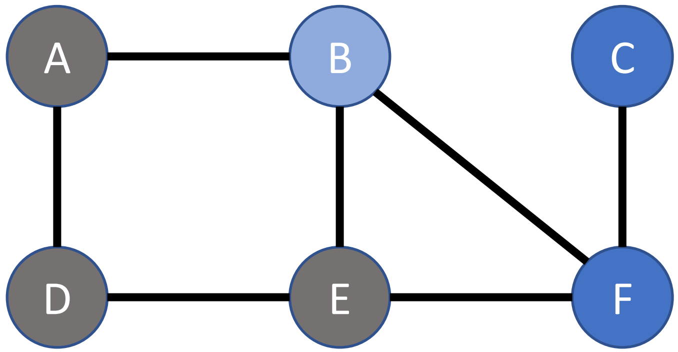


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
E	B <- A	{A, B, D, E}
B	D <- A	
	E <- D	

Pop E off the stack

start: A

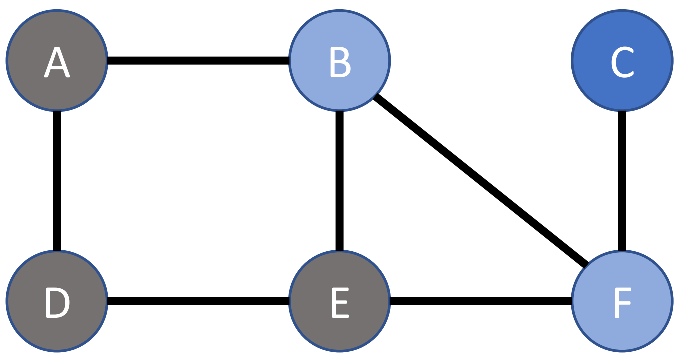


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore **(stack)**	**previous** **(map)**	**Visited** **(set)**
B	B <- A D <- A E <- D	{A, B, D, E}

Find F from E

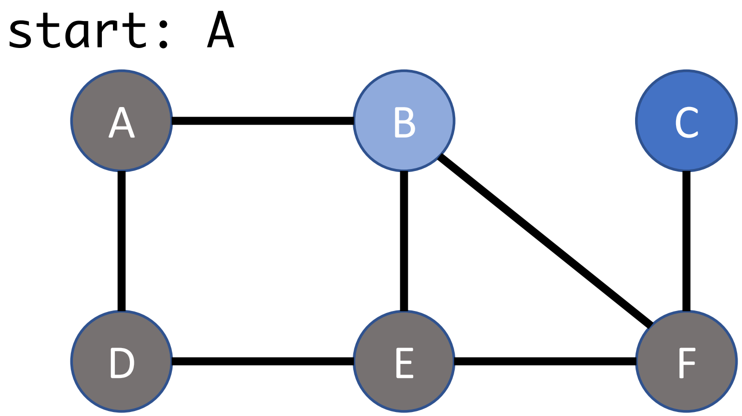
start: A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
F	B <- A	{A, B, D, E, F}
B	D <- A	
	E <- D	
	F <- E	

Pop F off the stack

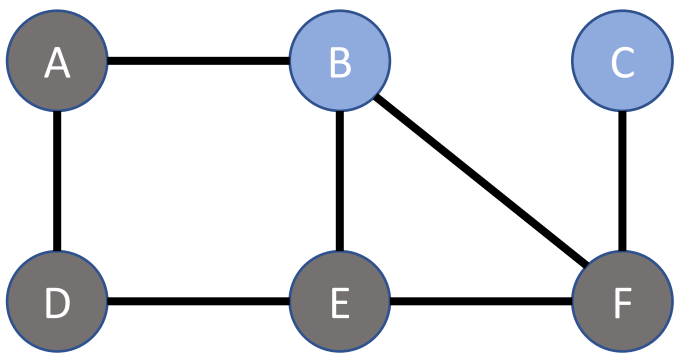


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
B	B <- A D <- A E <- D F <- E	{A, B, D, E, F}

Find C from F

start: A

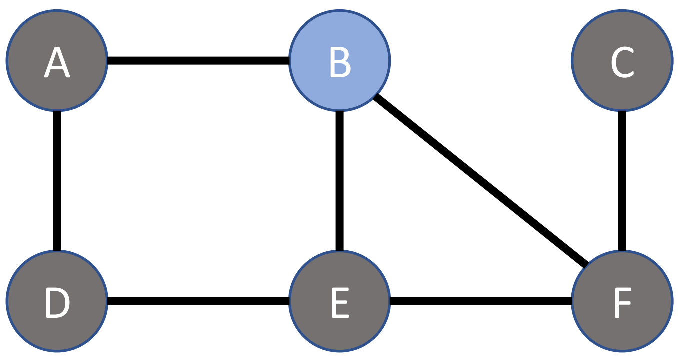


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous** (map)	**Visited** (set)
C	B <- A	{A, B, D, E, F, C}
B	D <- A	
	E <- D	
	F <- E	
	C <- F	

Pop C off the stack

start: A

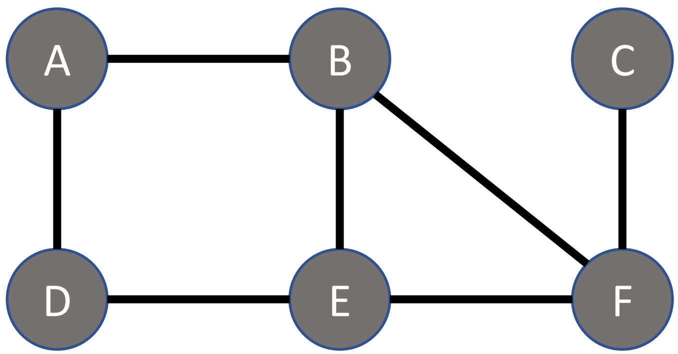


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
B	B <- A D <- A E <- D F <- E C <- F	{A, B, D, E, F, C}

Pop B off the stack

start: A



Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)

previous (map)

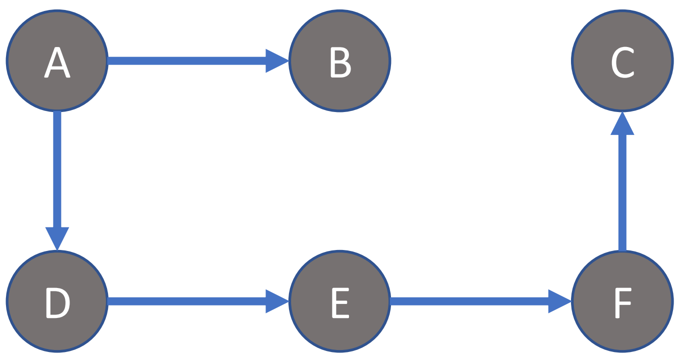
B <- A
D <- A
E <- D
F <- E
C <- F

Visited (set)

{A, B, D, E, F, C}

DFS Search Tree

start: A



Adjacency List:

A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (stack)	**previous (map)**	**Visited (set)**
	B <- A D <- A E <- D F <- E C <- F	{A, B, D, E, F, C}

Can find paths from A to X by following previous backwards from X

Path from A to C:
C <- F <- E <- D <- A