L23: DFS & BFS

Alex Steiger
CompSci 201: Spring 2024
4/8/2024
Logistics, coming up

• Today, Monday, April 8
  • Project P5: Huffman due
  • Project P6: Route out by tomorrow

• This Wednesday, April 10
  • APT Quiz 2 due
  • Covers linked list and trees
    • Practice quiz from discussion is similar
  • No regular APTs due this week, just the quiz
Today’s agenda

• General depth-first search (DFS)
  • Seen it on grid graphs, how about arbitrary graphs?

• Introduce breadth-first search (BFS)
Depth-First Search in General Graphs
Pathfinding / Graph Search

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

• Maps/directions
• Video games
• Robot motion planning
• Etc.
Recall: Grid Graph, Maze Example

• Example: 10 x 10 grid
• Edge = no wall, no edge = wall.
• Look for a path from start (lower left) to middle.

```java
public class MazeDemo {
    private int mySize;
    private boolean[][] north; // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
    private boolean[][] west;
}```
Depth-First Search for Solving Maze

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

If nothing new (unvisited) vertex to explore:

• **backtrack** to the most recent vertex adjacent to an unvisited vertex, and then continue.

• if no such vertex, maze is unsolvable.
Representations for Arbitrary Graphs (not only Grid Graphs)

Adjacency List

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Adjacent vertices (edges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td>A, C, D</td>
</tr>
<tr>
<td>C</td>
<td>A, B</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
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().

```
• 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

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

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

```java
public class DFS {
    public static Map<Character, Set<Character>> aList;
    public static Set<Character> visited;
    public static Map<Character, Character> previous;
}
```
Recursive DFS on a General Graph: Visiting all nodes

```java
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : alist.get(start)) {
            dfs(neighbor);
        }
    }
}
```
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]

Visited (set)
{A}

```
public static void dfs(char start) {
    if (!visited.contains(start)) {
        visited.add(start);
        System.out.println(start);
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);
        }
    }
}
```
Recurse on B

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

start: A

Visited (set)
{A, B}

```
14   public static void dfs(char start) {
15       if (!visited.contains(start)) {
16           visited.add(start);
17           System.out.println(start);
18           for (char neighbor : aList.get(start)) {
19               dfs(neighbor);
20           }
21       }
22   }
```
Recurse on E

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

start: A

Visited (set)
{A, B, E}

```java
14   public static void dfs(char start) {
15       if (!visited.contains(start)) {
16           visited.add(start);
17           System.out.println(start);
18           for (char neighbor : aList.get(start)) {
19               dfs(neighbor);
20           }
21       }
22   }
```
Recurse on D

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

Visisted (set)
{A, B, E, D}

```
14 public static void dfs(char start) {
15     if (!visited.contains(start)) {
16         visited.add(start);
17         System.out.println(start);
18         for (char neighbor : aList.get(start)) {
19             dfs(neighbor);
20         }
21     }
22 }
```
Backtrack to E, recurse on F

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

Visisted (set)
{A, B, E, D, F}

```java
14 public static void dfs(char start) {
15     if (!visited.contains(start)) {
16         visited.add(start);
17         System.out.println(start);
18         for (char neighbor : aList.get(start)) {
19             dfs(neighbor);
20         }
21     }
22 }
```
Recuse on C

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

Visited (set)
{A, B, E, D, F, C}

start: A

14 public static void dfs(char start) {
15     if (!visited.contains(start)) {
16         visited.add(start);
17         System.out.println(start);
18         for (char neighbor : aList.get(start)) {
19             dfs(neighbor);
20         }
21     }
22 }

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);
        }
    }
}
```

Recursion uses the call stack to keep track of nodes. Could also explicitly use a stack, can do the same for DFS.
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());
    }
}
```

LIFO = Last In First Out
Push: Add element to stack
Pop: Get last element in
Initializing Iterative DFS

• **Stack** stores nodes we have *visited/discovered*, but not explored from yet.

• Explore from one *current* node at a time.

```java
14 public static void dfs(char start) {
15     Stack<Character> toExplore = new Stack<>();
16     char current = start;
17     toExplore.add(current);
18     visited.add(current);
```

• Stack is LIFO (last-in first-out), so we always explore from the *last node we discovered*, depth-first!
While there are nodes we have not explored from...

```java
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);
        }
    }
}
```

Explore from the most recently discovered node...

Look at all neighbors of current node...

If we haven’t seen them before...

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]

start: A

<table>
<thead>
<tr>
<th>toExplore (stack)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>{A}</td>
</tr>
</tbody>
</table>
Pop A 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]

start: A

**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]

start: A

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]

start: A

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

start: A

<table>
<thead>
<tr>
<th>toExplore (stack)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B &lt;- A</td>
<td>{A, B, D}</td>
</tr>
<tr>
<td>D</td>
<td>D &lt;- A</td>
<td></td>
</tr>
</tbody>
</table>
Find E from D

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

start: A

<table>
<thead>
<tr>
<th>toExplore (stack)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>B &lt;- A</td>
<td>{A, B, D, E}</td>
</tr>
<tr>
<td>B</td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E &lt;- D</td>
<td></td>
</tr>
</tbody>
</table>
Pop E 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]
Find F from E

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

start: A

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 | {A, B, D, E, F}
D <- A
E <- D
F <- E
Find C from F

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

start: A

toExplore (stack)

previous (map)

Visited (set)

C
B

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

{A, B, D, E, F, C}
Pop C 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
C <- F | {A, B, D, E, F, C}
Pop B 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]

start: A

<table>
<thead>
<tr>
<th>toExplore (stack)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;- A</td>
<td>B &lt;- A</td>
<td>{A, B, D, E, F, C}</td>
</tr>
<tr>
<td>D &lt;- A</td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td>E &lt;- D</td>
<td>E &lt;- D</td>
<td></td>
</tr>
<tr>
<td>F &lt;- E</td>
<td>F &lt;- E</td>
<td></td>
</tr>
<tr>
<td>C &lt;- F</td>
<td>C &lt;- F</td>
<td></td>
</tr>
</tbody>
</table>
DFS Search Tree

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

start: A

\[\text{toExplore} \ (\text{stack})\]
\[\text{previous} \ (\text{map})\]
\[\text{Visited} \ (\text{set})\]

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

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

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

Can find paths from A to X by following previous backwards from X
DFS Complexity?

```
20 while (!toExplore.isEmpty()) {
21     current = toExplore.pop();
22     for (char neighbor : aList.get(current)) {
23         if (!visited.contains(neighbor)) {
24             previous.put(neighbor, current);
25             visited.add(neighbor);
26             toExplore.push(neighbor);
27         }
28     }
29 }
```

While loop over all nodes (N), potentially?
Loop over edges (M)

Seems like O(NM), but...
DFS Complexity?

```java
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);
        }
    }
}
```

- Pop each of N nodes at most once.
- Loop over neighbors of each node exactly once, considers each edge twice.
- \(N + 2M\) is \(O(N+M)\).
Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

After running DFS, which of these data structures would you use to get the actual path from a start vertex to a destination? *
public class DFS {
    public static Map<Character, Set<Character>> aList;
    public static Set<Character> visited;
    public static Map<Character, Character> previous;

    while (!toExplore.isEmpty()) {
        current = toExplore.pop();
        for (char neighbor : aList.get(current)) {
            if (!visited.contains(neighbor)) {
                Check all nodes reachable by one edge from the node we are exploring
            }
        }
    }
}
Check all of the unvisited nodes

4

Same code. The while loop on line 20 might have fewer than N iterations (when there are N nodes in the graph) when...

```
while (!toExplore.isEmpty()) {
    current = toExplore.pop();
    for (char neighbor : aList.get(current)) {
        if (!visited.contains(neighbor)) {
```

Some nodes are connected to many other nodes in the graph

Some nodes are not reachable from others

Never, the while loop should always have N iterations

5

What best describes the runtime complexity of DFS using a stack and hash-based data structures? Let N be the number of vertices and M be the number of edges.
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);
        }
    }
}
Iterative Breadth-First Search (BFS)
Queue: A FIFO List

• Both add and remove are $O(1)$
  • Add at end of LinkedList
  • Remove from front of LinkedList

```java
public static void qdemo() {
    String[] strs = {"compsci", "is", "wonderful"};
    Queue<String> q = new LinkedList<>();
    for (String s : strs) {
        q.add(s);
    }
    while (!q.isEmpty()) {
        System.out.println(q.remove());
    }
}
```
Level Order Tree Traversal using a Queue

```java
public static void levelOrder(TreeNode tree) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(tree);
    while (!queue.isEmpty()) {
        TreeNode current = queue.remove();
        if (current != null) {
            System.out.println(current.info);
            queue.add(current.left);
            queue.add(current.right);
        }
    }
}
```

Idea: Use a queue to keep track of nodes. First-in first-out, nodes visited in level order
**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.
**Breadth-First Search** for Solving Maze

Explore *all* your neighbors (adjacent vertices) before you visit any of your neighbors’ neighbors.

Looking for the shortest path/solution.
Queue = BFS, Stack = DFS

BFS: FIFO Exploration
search all locations one-away from start, then two-away, ...

DFS: LIFO Exploration
Search path as far as possible, backtrack if need to another branch...
Initializing Iterative BFS

• **Queue** stores nodes we have *visited/discovered*, but not explored from yet.

• Explore from one *current* node at a time.

```java
32     public static void bfs(char start) {
33         Queue<Character> toExplore = new LinkedList<>();
34         char current = start;
35         visited.add(current);
36         toExplore.add(current);
```

• Queue is FIFO (first-in first-out), so we always explore from the *first/closest (unvisited) node we discovered, breadth-first*!
Iterative BFS Loop

While there are nodes we have not explored from...

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

Explore from the **closest** discovered node...

Look at all neighbors of current node...

If we haven’t seen them before...

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]

**Adjacency List:**

<table>
<thead>
<tr>
<th>Node</th>
<th>Adjacent Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, D</td>
</tr>
<tr>
<td>B</td>
<td>A, E, F</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
</tr>
<tr>
<td>D</td>
<td>A, E</td>
</tr>
<tr>
<td>E</td>
<td>B, D, F</td>
</tr>
<tr>
<td>F</td>
<td>B, C, E</td>
</tr>
</tbody>
</table>

**toExplore (queue)**

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

**previous (map)**

<table>
<thead>
<tr>
<th>Node</th>
<th>Previous Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

**Visited (set)**

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

start: A
Remove A from the queue

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

toExplore (queue) 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]

start: A

toExplore (queue)  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]

start: A

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B &lt;- A</td>
<td>{A, B, D}</td>
</tr>
<tr>
<td>D</td>
<td>D &lt;- A</td>
<td></td>
</tr>
</tbody>
</table>

Note the difference, add to end of queue!
Remove B from queue

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

start: A
B was first in, B is first out

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>B &lt;- A</td>
<td>{A, B, D}</td>
</tr>
<tr>
<td>D &lt;- A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Find E from B

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

start: A

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>B &lt;- A</td>
<td>{A, B, D, E}</td>
</tr>
<tr>
<td>E</td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E &lt;- B</td>
<td></td>
</tr>
</tbody>
</table>
Find F from B

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

toExplore (queue) | previous (map) | Visited (set)
---|---|---
D | B <- A | {A, B, D, E, F}
E | D <- A |
F | E <- B |
    | F <- B |
Remove D from queue

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

start: A

toExplore (queue) | previous (map) | Visited (set)
---|---|---
E | B <- A | \{A, B, D, E, F\}
F | D <- A
   | E <- B
   | F <- B

4/8/24
Remove E from queue

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

toExplore (queue)    previous (map)    Visited (set)
F                  B <- A
D <- A
E <- B
F <- B
{A, B, D, E, F}
Remove F from queue

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

start: A

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;- A</td>
<td></td>
<td>{A, B, D, E, F}</td>
</tr>
<tr>
<td>D &lt;- A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E &lt;- B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &lt;- B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Find C from F

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

start: A

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B &lt;- A</td>
<td>{A, B, D, E, F, C}</td>
</tr>
<tr>
<td></td>
<td>D &lt;- A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E &lt;- B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F &lt;- B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C &lt;- F</td>
<td></td>
</tr>
</tbody>
</table>
Remove C from queue

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

start: A

<table>
<thead>
<tr>
<th>toExplore (queue)</th>
<th>previous (map)</th>
<th>Visited (set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt;- A</td>
<td></td>
<td>{A, B, D, E, F, C}</td>
</tr>
<tr>
<td>D &lt;- A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E &lt;- B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &lt;- B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C &lt;- F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BFS Search Tree

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

start: A

**toExplore** (queue)
B <-> A
D <-> A
E <-> B
F <-> B
C <-> F

**previous** (map)

**Visited** (set)
{A, B, D, E, F, C}
Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *  

solutions

2

True or false: These global data structures will not work for / need to be changed for BFS vs DFS. *  
Which line of code best explains what is different about BFS vs. DFS algorithmically?

- Line 33
- Line 38
- Line 40
What best explains why the while loop on line 38 only considers each node in the graph once / is $O(N)$? *  

○ Because Queues do not store duplicates  
○ Because we only consider each node as a "neighbor" once  
○ Because of the visited Set
If there are N nodes and M edges in the graph and the graph is connected, how many total times might line 41 be executed? *

- O(N)
- O(M)
- O(NM)

6

True or false: BFS can find shortest paths from the start node to all other reachable nodes. *

- True
- False
True or false: BFS explores all possible paths from the start node to all other reachable nodes. *

- [ ] True
- [x] False
Comparing DFS and BFS Search Trees

start: A

A → B
D → E

previous (map)
B <- A
D <- A
E <- D
F <- E
C <- F

Length 4 path from A to C

start: A

A → B
D → E

previous (map)
B <- A
D <- A
E <- B
F <- B
C <- F

Length 3 path from A to C, shorter!
Pathfinding Properties

• DFS and BFS **both** find valid paths to *all* nodes reachable from the start.
  • Can return early if you only want to find a path to a specific target node

• BFS finds the **shortest path** to every reachable node, DFS does *not* guarantee this.