L23: DFS & BFS

Alex Steiger
CompSci 201: Spring 2024
4/8/2024

1

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

2

Today's agenda

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

3

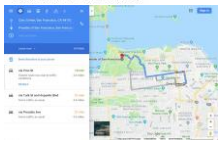
Depth-First Search in General Graphs

4

Pathfinding / Graph Search

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

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

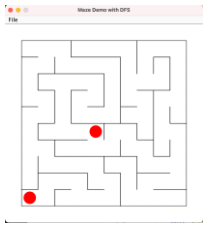


5

Recall: Grid Graph, Maze Example

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

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



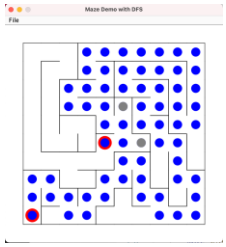
6

Depth-First Search for Solving Maze

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

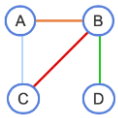
If nothing new (unvisited) vertex to explore:

- ***backtrack*** to the most recent vertex adjacent to an unvisited vertex, and then continue.
- if no such vertex, maze is unsolvable.



7

Representations for Arbitrary Graphs (not only Grid Graphs)



Adjacency List

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

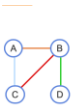
Adjacency Matrix

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

8

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)

9

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?

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

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

10

Recursive DFS on a General Graph: Visiting all nodes

```
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  }
```

Base case: If already visited, backtrack

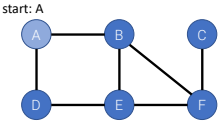
Else, visit this node

And explore its neighbors, adjacent nodes

11

Initialize search at A

start: 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)

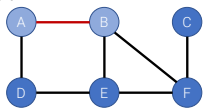
{A}

```
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  }
```


12

Recurse on B

start: 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)

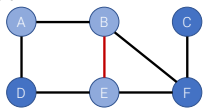
{A, B}

```
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  }
```


13

Recurse on 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]

Visited (set)

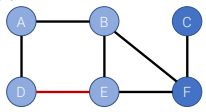
{A, B, E}

```
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  }
```


14

Recurse on 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]

Visited (set)

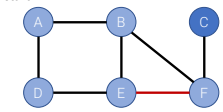
{A, B, E, D}

```
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  }
```


15

Backtrack to E, recurse on 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]

Visited (set)

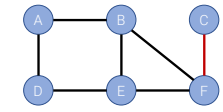
{A, B, E, D, F}

```
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  }
```


16

Recurse on C

start: 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)

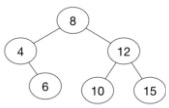
{A, B, E, D, F, C}

```
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  }
```


17

Did we really need recursion? preOrder Tree Traversal with Stack

```
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

18

Stack Abstract Data Structure: LIFO List

```
5   public static void sdemo() {
6       String[] strs = {"compsci", "is", "wonderful"};
7       Stack<String> st = new Stack<>();
8       for(String s : strs) {
9           st.push(s);
10      }
11      while (! st.isEmpty()) {
12          System.out.println(st.pop());
13      }
14  }
```

wonderful
is
compsci

LIFO = Last In First Out

Push: Add element to stack

Pop: Get last element in

19

Initializing Iterative DFS

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

```
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**!

20

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…

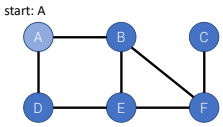
If we haven't seen them before…

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

```
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      }
```


21

Initialize search at A

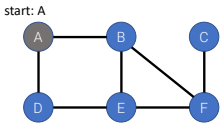


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

22

Pop A off the stack

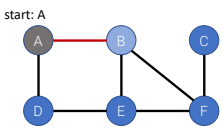


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

23

Find B from A

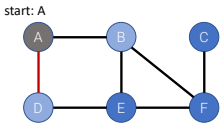


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

24

Find D from A

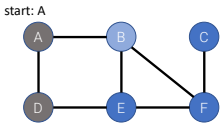


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

25

Pop D off the stack

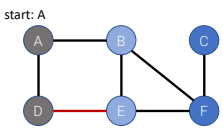


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

26

Find E from D

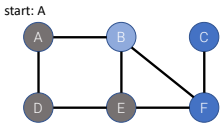


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

27

Pop E off the stack

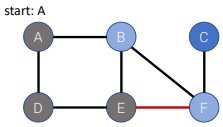


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

28

Find F from E

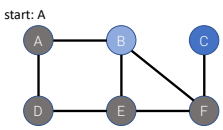


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

29

Pop F off the stack



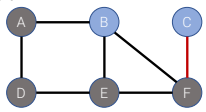
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

30

Find C from F

start: A



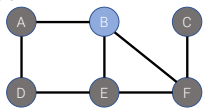
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

31

Pop C off the stack

start: A



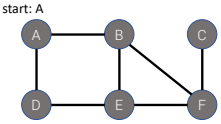
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

32

Pop B off the stack

start: A

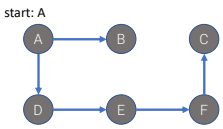


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

33

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]

toExplore (stack) | **previous (map)** | **Visited (set)**

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

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

34

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…

36

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  }
```

Loop over edges **adjacent to current node**

- 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).

37

L22-WOTO2-GeneralDFS-Sp24

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? *

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

- ○ aList
- ○ visited
- ◉ previous
- ○ none of the above

3

The best explanation of the loop on line 22 is... *

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

- ○ Check all nodes reachable by one edge from any visited nodes
- ◉ 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... *

```
20    while (!toExplore.isEmpty()) {
21        current = toExplore.pop();
22        for (char neighbor : aList.get(current)) {
23            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. *

```
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            }
```

- ○ O(N)
- ◉ O(N+M)
- ○ O(NM)

6

True or false: This dfs algorithm will always find the shortest path from the start node to other nodes *

- ○ True
- ◉ False

Iterative Breadth-First Search (BFS)

38

`Queue`: A FIFO List

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

LinkedList implements the Queue interface.

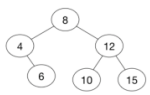
```
5   public static void qdemo() {
6       String[] strs = {"compsci", "is", "wonderful"};
7       Queue<String> q = new LinkedList<>();
8       for(String s : strs) {
9           q.add(s);
10      }
11      while (! q.isEmpty()) {
12          System.out.println(q.remove());
13      }
14  }
```

compsci
is
wonderful

39

Level Order Tree Traversal using a Queue

```
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);
        }
    }
}
```



8
4
12
6
10
15

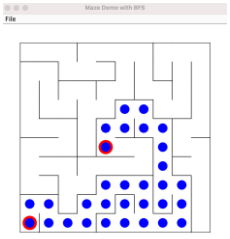
Idea: Use a queue to keep track of nodes.
First-in first-out, nodes visited in ***level order***

40

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.

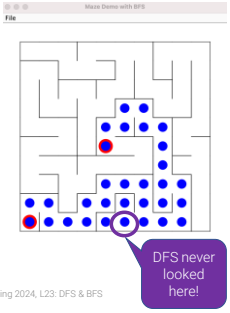


41

Breadth-First Search for Solving Maze

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

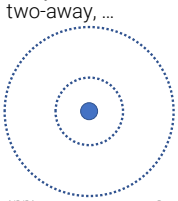
Looking for the shortest path/solution.



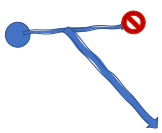
42

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…



43

Initializing Iterative BFS

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

```
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**!

44

Iterative BFS Loop

While there are nodes we have not explored from…

Explore from the **closest** discovered node…

Look at all neighbors of current node…

If we haven't seen them before…

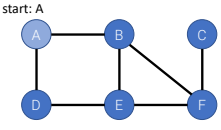
Then:
1. Note how we got here
2. Note we have seen
3. Mark to explore later

```
38    while (!toExplore.isEmpty()) {
39        current = toExplore.remove();
40        for (char neighbor : aList.get(current)) {
41            if (!visited.contains(neighbor)) {
42                previous.put(neighbor, current);
43                visited.add(neighbor);
44                toExplore.add(neighbor);
45            }
46        }
47    }
```


45

Initialize search at A

start: A



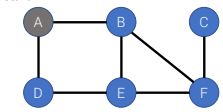
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

46

Remove A from the queue

start: A



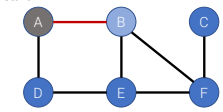
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

47

Find B from A

start: A



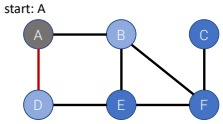
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

48

Find D from A

start: A



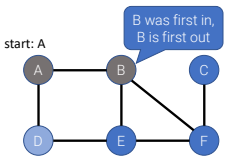
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

Note the difference, add to end of queue!

49

Remove B from queue

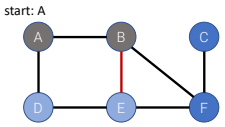


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

50

Find E from B

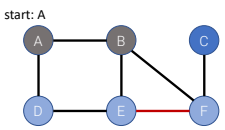


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

51

Find F from B



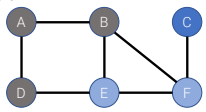
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

52

Remove D from queue

start: A



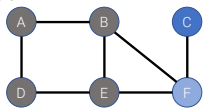
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

53

Remove E from queue

start: A



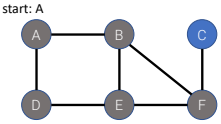
Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

54

Remove F from queue

start: A

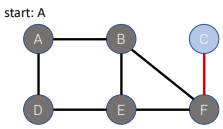


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

55

Find C from F

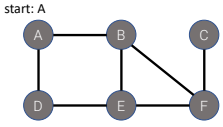


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

56

Remove C from queue

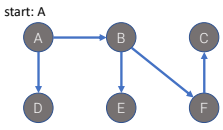


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

57

BFS Search Tree

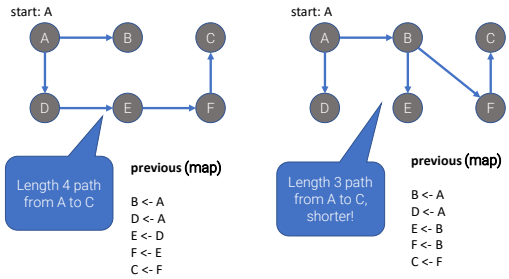


Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

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

58

Comparing DFS and BFS Search Trees



59

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.

60

L23-WOTO2-BFS-Sp24

Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

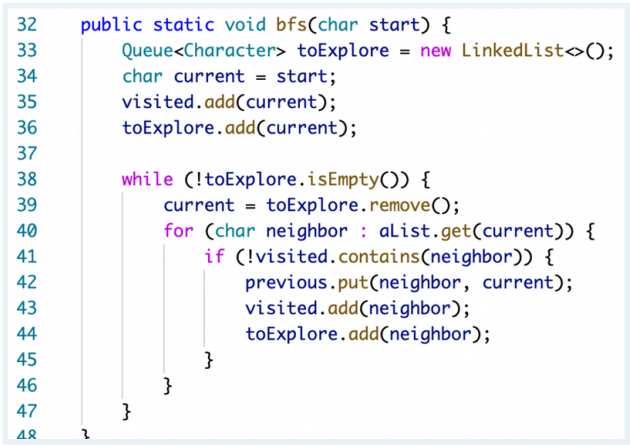
True or false: These global data structures will not work for / need to be changed for BFS vs DFS. *

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

- ○ True
- ◉ False

3

Which line of code best explains what is different about BFS vs. DFS algorithmically? *



```
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);
37
38      while (!toExplore.isEmpty()) {
39          current = toExplore.remove();
40          for (char neighbor : aList.get(current)) {
41              if (!visited.contains(neighbor)) {
42                  previous.put(neighbor, current);
43                  visited.add(neighbor);
44                  toExplore.add(neighbor);
45              }
46          }
47      }
48  }
```

- ◉ Line 33
- ○ Line 38
- ○ Line 40

- ◯ Line 41

4

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

```
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);
37
38        while (!toExplore.isEmpty()) {
39            current = toExplore.remove();
40            for (char neighbor : aList.get(current)) {
41                if (!visited.contains(neighbor)) {
42                    previous.put(neighbor, current);
43                    visited.add(neighbor);
44                    toExplore.add(neighbor);
45                }
46            }
47        }
48    }
```

- ◯ Because Queues do not store duplicates
- ◯ Because we only consider each node as a "neighbor" once
- ◉ Because of the visited Set

5

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? *

```
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);
37
38      while (!toExplore.isEmpty()) {
39          current = toExplore.remove();
40          for (char neighbor : aList.get(current)) {
41              if (!visited.contains(neighbor)) {
42                  previous.put(neighbor, current);
43                  visited.add(neighbor);
44                  toExplore.add(neighbor);
45              }
46          }
47      }
48  }
```

- ○ 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

7

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

- ○ True
- ◉ False