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)
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

```
public class MazeDemo {
    private int mySize;
    private boolean[] north;
    private boolean[] east;
    private boolean[] south;
    private boolean[] west;

    // Example: 10 x 10 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 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></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td></td>
<td></td>
<td></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().

```java
    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.

Recursive DFS on a General Graph: Visiting all nodes

Base case: If already visited, backtrack
Else, visit this node
And explore its neighbors, adjacent nodes

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]
Recursion on B

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)

Recursion on E

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)

Recursion on D

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)
Backtrack to E, recurse on F

Recurse on C

Did we really need recursion? preOrder Tree Traversal with Stack

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

public static void main() {
    String[] extra = {'compsci', 'java', 'wonderful'};
    Stack<String> st = new Stack<>();
    for (String x : extra) {
        st.push(x);
    }
    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.

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…
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\}

Pop A off the stack

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

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

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

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 <- A
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]

start: A
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 < A | {A, B, D, E, F, C}
B | D < A
D < A
E < D
F < E
C < F

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]

Start: A

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

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

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

To Explore (stack) | Previous (map) | Visited (set)
--- | --- | ---
B < A | {A, B, D, E, F, C} | {A, B, D, E, F, C}
D < A | |
E < D | |
F < E | |
C < F | |

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

DFS Complexity?

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

Seems like O(NM), but...

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).
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? * [ ]
The best explanation of the loop on line 22 is...

- 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

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

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);
        }
    }
}

O(N+M)

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

False
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

Level Order Tree Traversal using a Queue

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.

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…

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]
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]

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]

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]

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

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

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

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

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]

start: A

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

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]

Start: A

toExplore [queue] previous [map] Visited [set]
- F: B < A, D < A, E < B

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

toExplore [queue] previous [map] Visited [set]
- B: A, D < A, E < B, F < B
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 (queue) previous (map) Visited (set)
C B <- A
D <- A
E <- B
F <- B
C <- F

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

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) previous (map) Visited (set)
B <- A
D <- A
E <- B
F <- B
C <- F
Comparing DFS and BFS Search Trees

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.
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)

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