CSM 61B Fall 2017

Graphs & Searches

Mentoring 12: November 13, 2017

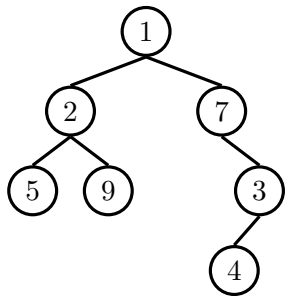
1 Tree Traversal

A **queue** is a data structure that orders items in a first-in-first-out (FIFO) manner, meaning that the first element you `add` will be at the front and the last item you `add` will be at the tail. Elements are `removed` from the front.

A **stack** is a data structure that orders items in a last-in-first-out (LIFO) manner, meaning that the first element you `add` will be at the tail and the last item you `add` will be at the front. Elements are `removed` from the front.

1.1
```
public void treeTraversal(Fringe<Node> fringe) {
    fringe.add(root);
    while (!fringe.isEmpty()) {
        Node node = fringe.remove();
        System.out.print(node.value);
        if (node.left != null) {
            fringe.add(node.left);
        }
        if (node.right != null) {
            fringe.add(node.right);
        }
    }
}
```

```
public class BinaryTree<T> {
    protected Node root;
    protected class Node {
        public T value;
        public Node left;
        public Node right;
    }
}
```



What would Java display?

(a) `tree.traversal(new Queue<Node>());`

1275934

Notice that this solution goes in breadth-first order. We see that we start at the root 1, and then proceed to traverse the tree level-by-level. The second level is 2 and 7, which are both one hop away from the root. The third level is 5, 9, and 3, which are all two hops away from the root. Lastly, we have 4, which is three hops away from the root.

(b) `tree.traversal(new Stack<Node>());`

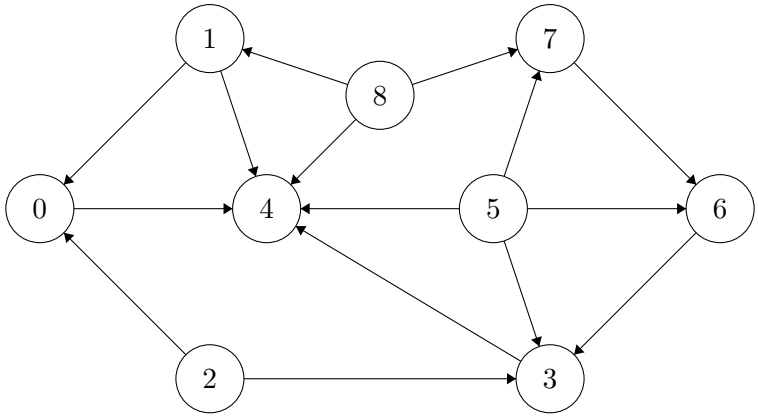
1734295

Notice that this solution goes in depth-first order. We first traverse all the way down the right branch: 1, 7, 3, 4. Once we reach 4, the leaf node, we go back up to the root and traverse all the way down the left branch: 2, 9, 5. We always traverse the right branch first due to the way the code is written (the right branch is added to the stack after the left branch, meaning the right branch would be explored first in LIFO ordering).

2 Graph Traversal

```
procedure GRAPH-TRAVERSAL(start, fringe)
    seen ← an empty set
    ADD(start, fringe)
    while fringe is not empty do
        node ← REMOVE(fringe)
        if node is not in seen then
            ADD(node, seen)
            for child in NEIGHBORS(node) do
                ADD(child, fringe)
```

2.1 Consider the following directed graph. Break ties numerically from least to greatest. For example, when iterating through the edges pointing from vertex 5, consider the edge $(5, 3)$ before the others.



(a) Give the depth-first *pre-order* traversal of the graph.

`0, 4, 1, 2, 3, 5, 6, 7, 8`

(b) Give the depth-first *post-order* traversal of the graph.

`4, 0, 1, 3, 2, 6, 7, 5, 8`

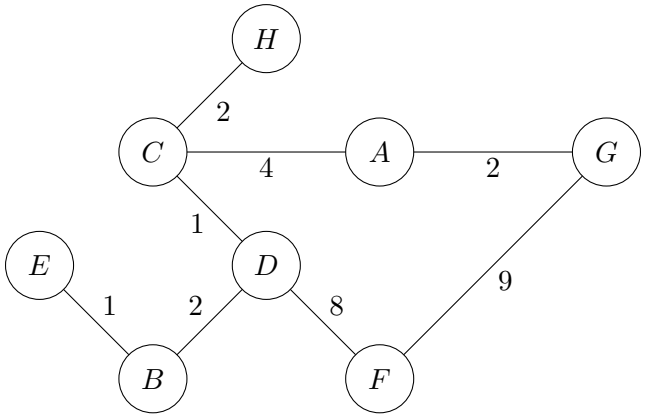
(c) Give the *reverse* depth-first *post-order* traversal.

`8, 5, 7, 6, 2, 3, 1, 0, 4`

Meta: This is a topological sort of the graph. Redraw the graph in this order to show all of the arrows pointing rightward.

3 Searches

3.1 For the graph below, write the order in which vertices are visited using the specified algorithm starting from A. Break ties by alphabetical order. Notice that we have now introduced edge weights to the graph.



(a) DFS

$A - C - D - B - E - F - G - H$

(b) BFS

$A - C - G - D - H - F - B - E$

(c) Dijkstra's

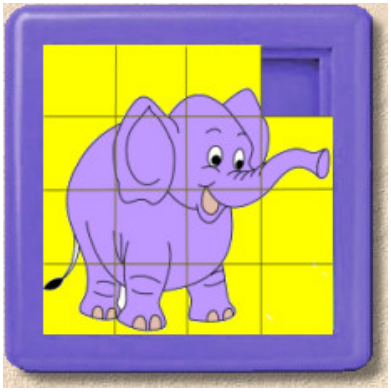
$A - G - C - D - H - B - E - F$

4 Elephants *Extra Practice*

A `State` of this puzzle is some permutation of the puzzle tiles. There are two things we can do with a `State`:

- Get the set of next possible states from the current `State`.
- Find out if the `State` is a goal: in this case when the puzzle is solved.

Imagine a game graph as a graph of all possible states where each state is a graph node and where a method, `getNextStates`, returns the neighbor nodes. Finding a solution is equivalent to finding a path from some start state to goal state.

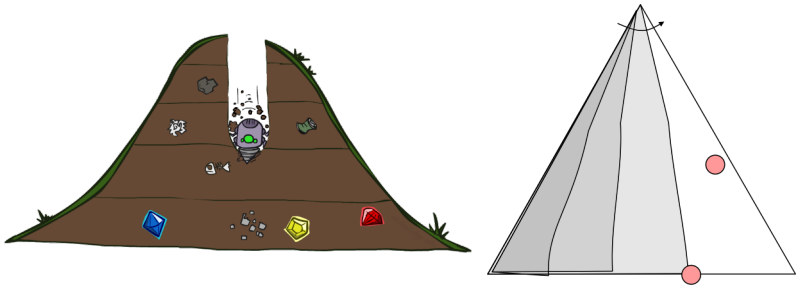


4.1 Would BFS or DFS be better for finding the solution that takes the least number of moves to solve the game?

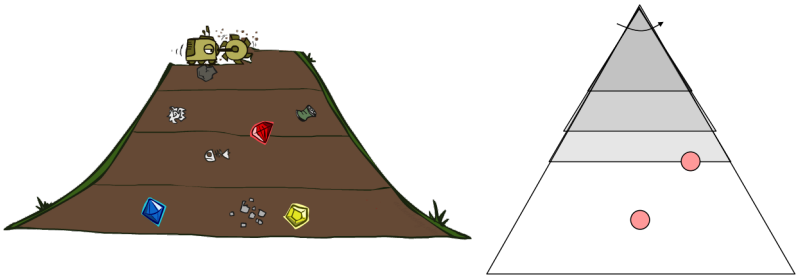
BFS

Each step in BFS is equivalent to one move, which is the metric we are using.

DFS:



BFS:



We start the search from the top of the triangle and move downwards. The circles represent possible solutions.

The first two pictures represent DFS. We see that here, DFS will find the "leftmost" solution, even if it is deeper than other solutions. For example, in the DFS triangle graphic, we see that DFS first finds the bottom left solution, even though this is more deep (and thus less efficient) than the solution on the right.

The second two pictures represent BFS. We see that here, BFS will find the most optimal solution. For example, in the BFS triangle graphic, the first solution returned is the most optimal one (the one closest to the root).

Images from Anca Dragan's CS 188 lecture slides.

4.2 Define the `solve` method in `PicturePuzzle` which returns the board `State` that is the solution with the least number of moves away from a given starting `State`.

```
public interface State {
    public Set<State> getNextStates();
    public boolean isGoal();
}


public class PicturePuzzle {
    public static State solve(State startState) {
        Set<State> seen = new HashSet<>();
        Queue<State> nextStates = new LinkedList<>();
        seen.add(startState);
        nextStates.add(startState);
        while (!nextStates.isEmpty()) {
            State currentState = nextStates.remove();
            if (currentState.isGoal()) {
                return currentState;
            }
            for (State nextState : currentState.getNextStates()) {
                if (!seen.contains(nextState)) {
                    seen.add(nextState);
                    nextStates.add(nextState);
                }
            }
        }
        return null;
    }
}
```