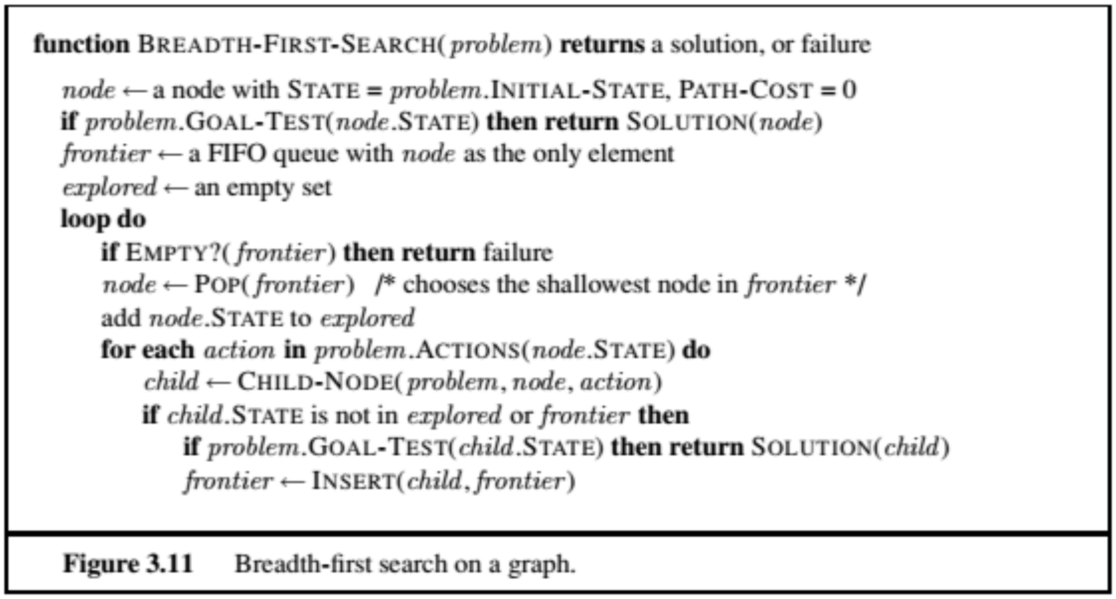
CS1571
Fall 2019
8/28 In-Class Worksheet

Name: Simran Gidwani

Where were you sitting in class today: Center towards the front

A. Breadth-First Search

The following is pseudocode for breadth-first search, one uninformed search strategy. Based on the pseudocode, answer the following questions.



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

1. What data structure is used to represent the frontier?

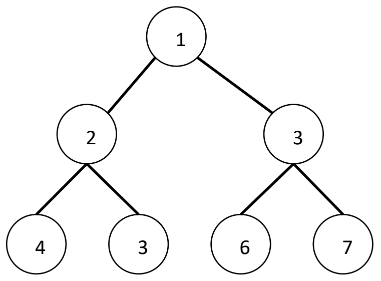
Frontier would be a list in FIFO queue form

2. What is the role of *explored* in this algorithm?

Explored represents whether the node has been visited making sure you don't visit states multiple times.

3. Follow the path of breadth-first search by filling in the following table, based on the search tree below. Numbers in the tree are abstract representations of particular states. Each iteration represents one cycle in the do loop; you should fill in the table .

Iteration	**Node**	**Frontier**	**Explored**
1	1	2, 3	1
2	2	3, 4	1, 2, 3
3	4	3, 4	1, 2, 3, 4
4	3		1, 2, 3, 4
5	6	7	1, 2, 3, 4, 6, 7



B. The following is pseudocode for depth-first search, another uninformed search strategy. Based on the pseudocode, answer the following questions.

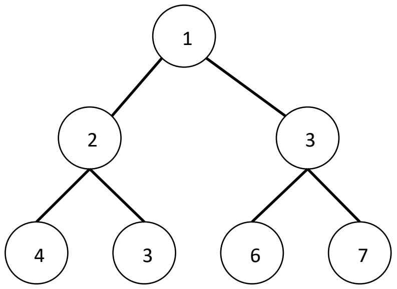
```
function DFS(node, problem) returns a solution, or failure
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else
        for each action in problem.ACTIONS(node.STATE) do
            child <- CHILD-NODE(problem, node, action)
            result <- RECURSIVE-DLS(child, problem)
            if result != failure then return result
        return failure
```

4. If you were to implement DFS non-recursively, what data structure would be used to represent the frontier?

LIFO queue/stack

5. What is the order the nodes are visited in for DFS?

1, 2 ,4, 3, 3, 6, 7



6. Evaluate the performance of the above BFS and DFS algorithms as they apply to a solvable Sudoku problem based on completeness, optimality, time complexity, and space complexity.

For BFS it is guaranteed to find a solution because you know the height of the tree so in terms of completeness, the Sudoku problem is solvable. For DFS, it could go down a path that is an infinite loop and not end up finding a solution. But the solution will be somewhere in the tree so yes.

In terms of optimality, I think it depends on the way the algorithm is interpreted. Because there are many alternative DFS algorithms, there can be a way that using this algorithm is not optimal. But altering it to avoid repeated paths and redundant paths than it would be finding the optimal solution. BFS I think doesn't find the most optimal solution because it finds all the solutions to the sudoku problem but also finds all the solutions that don't work. For that reason, the amount of time and space BFS takes up is extremely larger. For both there is no sense of optimality because there is no path cost and the number of steps don't matter.

Time complexity of BFS would be higher because you are searching through every algorithm even the ones that you know are not right.

In terms of time and space complexity because it is a 9 by 9 board, it would be O(9^81) at the worst case. But BFS uses a lot more space and memory due to the fact that it produces every possible combination even the ones that aren't solutions.