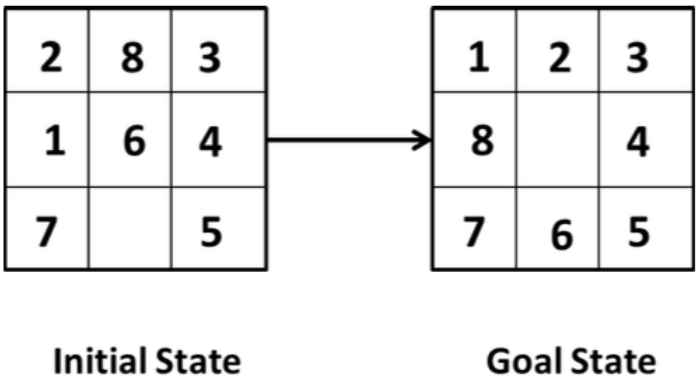
LAB 04 - Uninformed Searches

8 PUZZLE PROBLEM USING BFS

We will explore different search options for this problem.

8 Puzzle Problem

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.

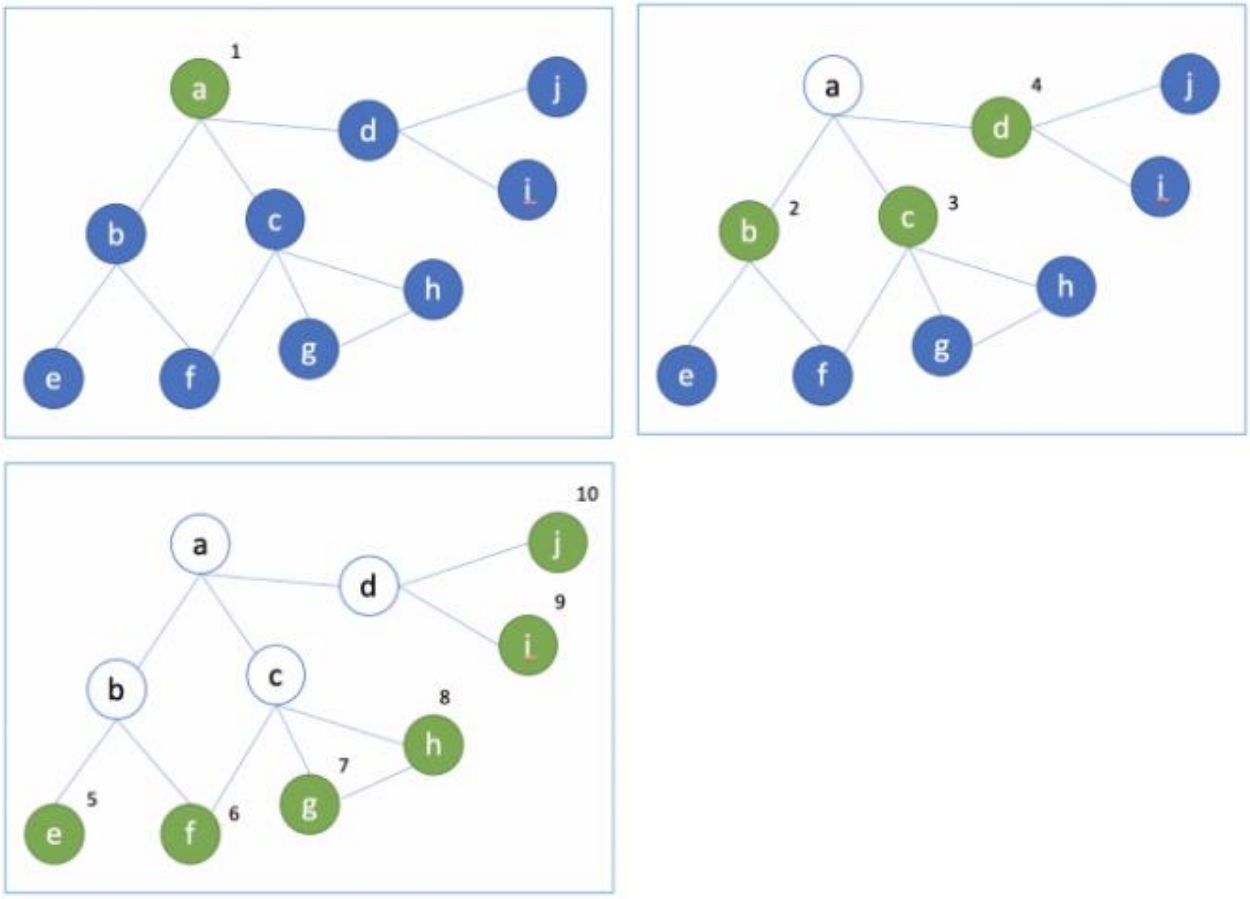


Breadth First Search

Breadth-First Search is a "blind" algorithm. It's called "blind" because this algorithm doesn't care about the cost between vertices on the graph. The algorithm starts from a root node (which is the initial state of the problem) and explores all nodes at the present level prior to moving on to the nodes at the next level. If the algorithm finds a solution, returns it and stops the search, otherwise extends the node and continues the search process. Breadth-First Search is "complete", which means that the algorithm always returns a solution if exists. More specifically, the algorithm returns the solution that is closest to the root, so for problems that the transition from one node to its children nodes costs one, the BFS algorithm returns the best solution. In addition, in order to explore the nodes level by level, it uses a queue data structure, so new nodes are added at the end of the queue, and nodes are removed from the start of the queue.

BFS and 8 Puzzle Problem

Breadth-First Search (BFS) starts by examining the first node and expands one layer at a time, for example, all nodes "one hop" from the first node; once those are exhausted it proceeds to all nodes "two hops" from the first node and so forth.



In order to accomplish this feat, BFS is implemented with a queue (first in, first out). The list of nodes "one level away" is often called the frontier:

```
const BFS = function (startNode, isGoal) {

    const frontier = [startNode]

    while(startNode.length > 0) {

       const currentNode = frontier.shift()

       if(isGoal(currentNode)) {
          return Solution(currentNode)
       }

       for(child of currentNode.children) {
          frontier.push(child)
       }
    }

    return NotFound()
}
```

The algorithm pushes the start node onto the frontier, and then proceeds by taking a node off of the

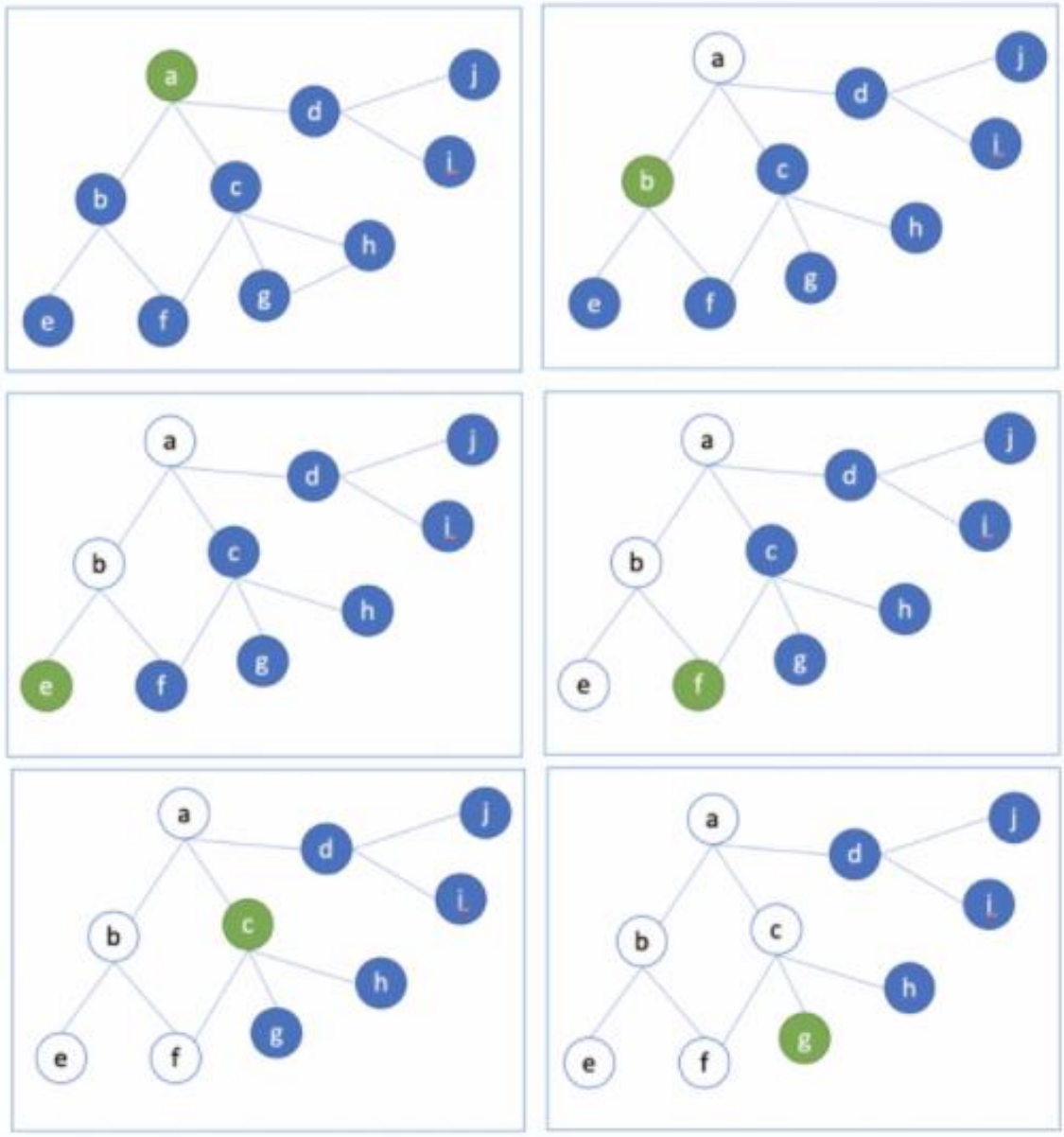
frontier, seeing if it solves the goal, if it doesn't add its children to the frontier. If the frontier ever reaches an empty state you have a mismatch between the goal and the search space.
BFS is a complete solution (will find a node that meets the goal if it exists) and does well when the space is small or the solution/goal is shallow (close to the startNode). It also has challenges with memory and time.
And that's where met problems in solving eight-puzzle. I certainly could have rewritten the graph to be more efficient, but in general, it wouldn't have created meaningful change in performance.

Depth First Search

The Depth-first Search is a "blind" search algorithm that can be used to find a solution to problems that can be modelled as graphs.
It's called "blind" because this algorithm doesn't care about the cost between vertices on the graph. The algorithm starts from a root node and explores as far as possible along each branch before backtracking. if the algorithm finds a solution then it returns the solution and stops the search.

DFS and 8 Puzzle Problem

Depth First Search (DFS) starts at a node and proceeds down the left-most node until it reaches a leaf. It then backs up to the leaf's parent and checks it next left-most node, and so on



DFS is also a complete solution that will ultimately find the goal if it exists, but it is implemented

recursively.

```
const DFS = function (node, isGoal) {

      if (isGoal(node)) {
        return Solution(node)
      }

      for (child of node.children.reverse) {

         if (notVisited(child)) {

            const resultOfRecursion = BFS(child, isGoal)

            if (resultOfRecursion instanceof Solution) {
                return resultOfRecursion
            }

         }
      }

      return NotFound()
}
```

It should be noted that searching graphs is a different challenge than searching trees. With trees, you can be certain that your search will not enter into any cycles (loops), but in a graph, nodes can cycle back to itself, which is a problem. To prevent that you simply track whether a node has been visited or not.

TASK:
Find solution to the 8 puzzle problem using iterative depth first search.