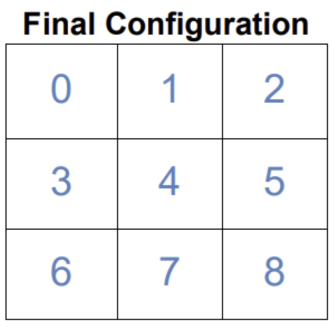
GroupNumber: 70
Group Members: Sabrina Johnson , Jessenta Drake
Date: Sept. 18, 2023
Course: CSE 368 Intro to Artificial Intelligence

<u>Assignment 1 Part 1 Report</u>

"Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space." (Geeks4Geeks)

In our implementation, the empty space corresponds to the tile labeled '0'. And the rules follow as described above. We are given an initial state configuration (this will typically be different from the goal/final configuration) and must find a set of moves to change that initial configuration into our goal configuration.



Final Configuration

0	1	2
3	4	5
6	7	8

<u>Breadth First Search (BFS)</u>

Optimization must still be made

The Breadth First Search (BFS) algorithm was implemented using a queue structure. This allows for a First-in First-Out (FIFO) technique to be used when traversing the tree of nodes (possible states from its "parent" state). Given the following diagram, we will show what the queue would look like for BFS algorithm

BFS Example:
Our goal is G

```
                 A                           depth = 0
                 /\
            B           C                    depth = 1
            /\          /\
       D    E            F    G              depth = 2
```

BFS Queue: [A, B, C, D, E, F, G ...]

Node A was added to the queue first as the root node. We then pop a node off of the front of the queue and expand the tree with its children (as long as that node's depth is not greater than 15); Expansion is done by adding the child nodes(s) to the queue. This procedure will be done until the queue is empty.

At this point, we have expanded the tree fully for 15 layers. If the solution was found during the expansion process, the algorithm would stop and 2 things; (1) the actions to find the solution for the given problem in a list form and (2) the cost of finding that solution (The cost of the initial state for that problem is 0 and for each move in that list of actions to find the solution, 1 is added to that cost). Otherwise, our algorithm exits and displays the performance metrics of the algorithm.

Depth First Search (DFS)

Briefly explain how your implementation works for each of the methods. Include any details if you used any optimization techniques.

Depth First Search (DFS) creates its search tree by expanding fully down one path until 'failure' and repeating this expansion process. 'Failure' may be defined as being unable to expand a path further because the child node causes a cycle or is a leaf that is not our solution state. One can consider DFS to metaphorically 'dig' in one spot until it can no longer, repeatedly digging deep holes until the goal is found or until algorithm termination.

One optimization technique used for DFS was the use of a stack. The stack's use of last in -first out helped to easily pop nodes off the stack.

Detail any parameters or configurations for each algorithm (e.g., depth limits for DFS).

For DFS, a layer limit was applied. If the program went to/beyond 16 layers the program would immediately exit.

Since DFS methods usually take the route of exploring via child nodes rather than neighboring nodes a condition to check whether or not the child nodes have already been explored was applied so the same search would occur more than once.

Lastly, A condition to quit the program was applied. When the goal solution is reached the program stops searching and returns the path it took to reach the goal, the cost, explored nodes, and the time it took to reach the solution.

Detail how DFS would work here. Then we go into our implementations

Method Structure (Stack):

The Depth First Search (DFS) algorithm was implemented using a stack structure to build its search tree. This allows for a Last-In-First-Out (LIFO) technique to be used. We initially create our root node and add it to the stack; We then begin our while loop. The top node on the stack is popped off. If this popped node is our goal state, stop; Otherwise, its children are assessed. If the child of our node is not an ancestor of the node (i.e. the child does not create a cycle) and the child has a depth <= 15 we add this node to the stack. Once all children are assessed, the while loop repeats the same process with the node on the top of the stack. This loop also checks that the child node isn't a member of the data structure that keeps track of whether or not that child node has been explored or is a part of the main stack being used. This cycle will occur until we have found our problem solution or the search tree has been expanded fully for 15 layers.

DFS Example:

Our goal is G

```
                    A                       depth = 0
                   /\
              B            C                depth = 1
             /\           /\
         D    E            F     G          depth = 2
```

BFS Stack: [A, B, D, E, C, F, G…]

Comparison of method's Performance/Solution Quality

Efficiency

Given a problem which was created with few shifts, BFS served as the more efficient algorithm because the first solution path found was with expanding all children at the same depth before moving to the next; allowing for a shorter path to be generated.

Given a randomly generated problem (farther from the goal state), DFS becomes the more efficient algorithm. Consider BFS on a problem where the shortest solution path is 15. BFS would fully expand the tree for 14 layers before reaching a solution path ; Creating many unneeded nodes/paths. DFS, on the other hand, could reach that path much quicker without creating as many unfruitful paths.

Thus, deciding which algorithm is more efficient would be dependent on the problem and amount of shifts used to create our starting grid.

Optimality

For the sliding grids problem, optimality could be defined by the length of the solution path. Considering the expansion methods of each algorithm, BFS would be more optimal than DFS in situations where the possible solution paths for a given problem vary greatly in length. This is because DFS searches for the shortest solution path of length k (arbitrary in >= 0) before expanding further. DFS could find a solution path of larger length early in its search and terminate before expanding another node completely that produces a shorter solution path. The two algorithms have the same optimality for problems with solution paths that do not vary in length (ex. All solution paths are of length 15).

Key Findings/Solution Quality

Based on the way that BFS expands its search tree (from a node to all its neighbors), we believe it to be the more optimal algorithm in comparison to DFS (which spreads fully down one path before expanding another). Depending on how far the initial state is from the goal state, either algorithm can perform faster and more effectively considering space complexity (i.e., nodes created/explored). BFS would produce a higher solution quality by inadvertently finding a shorter path in the general case of the sliding puzzle problem. You can reference the first solution paths provided by the two algorithms for the first test code problem.

Team Member	Assignment Part	Contribution (%)
Sabrina Johnson (50407988)	BFS (code), BFS report section, DFS report section (non-recursive), Solution Quality Section	

Jessenta Drake (50358141)	DFS, DFS report section (recursive)	

References/Work Cited

https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/
Date accessed: 9/18/2023