8-Puzzle Project Report

<u>Group Members</u>

Jaylen Brown

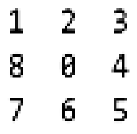
Eric Kemmer

Randall Hunt

Abstract

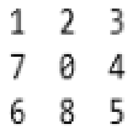
The 8-puzzle is a puzzle that typically consists of seven tiles and a blank space. To solve the puzzle, one must swap an adjacent tile with the blank space until a goal state is reached. Creating a simulation of this game in C++ would mean that our tiles and blank space would be represented using a two-dimensional array with zero representing the blank space and numbers one through eight representing the other tiles as shown below:

Goal State:



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

Initial State:

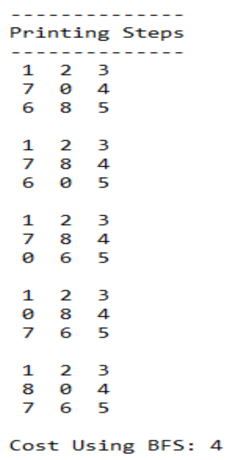


```
1  2  3
7  0  4
6  8  5
```

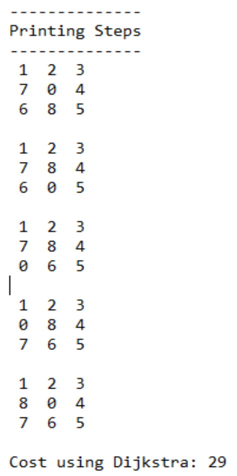
All the possible moves, up, down, left, right, can be represented in a graph. Each move being a child of the initial state eventually leading to the goal state. We can use three different algorithms to traverse this graph, breadth first search, depth first, and Dijkstra’s.

Results

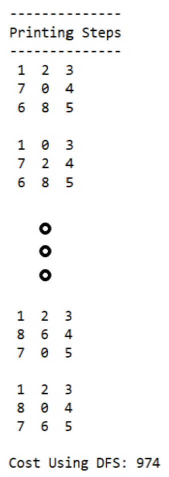
Our results show that DFS can have inferior performance compared to the BFS and Dijkstra's algorithms as shown below:



```
--------------
Printing Steps
--------------
 1  2  3
 7  0  4
 6  8  5

 1  2  3
 7  8  4
 6  0  5

 1  2  3
 7  8  4
 0  6  5

 1  2  3
 0  8  4
 7  6  5

 1  2  3
 8  0  4
 7  6  5

Cost Using BFS: 4
```



```
--------------
Printing Steps
--------------
 1  2  3
 7  0  4
 6  8  5

 1  2  3
 7  8  4
 6  0  5

 1  2  3
 7  8  4
 0  6  5
|
 1  2  3
 0  8  4
 7  6  5

 1  2  3
 8  0  4
 7  6  5

Cost using Dijkstra: 29
```



```
-------------
Printing Steps
-------------
 1  2  3
 7  0  4
 6  8  5

 1  0  3
 7  2  4
 6  8  5

       ○
       ○
       ○

 1  2  3
 8  6  4
 7  0  5

 1  2  3
 8  0  4
 7  6  5

Cost Using DFS: 974
```

Looking at these results, we can see that Dijkstra's and BFS both follow the same pathing to get the same result, however Dijkstra's cost more because it is dependent on the actual tile, we are moving rather than the number of moves. Moreover, it is clear in these results that DFS gives the worst answer out of the three algorithms. We implemented the following algorithms as such:

Breadth first search traverses a graph in level order and has a runtime of $O(|V + E|)$. We utilized two lists to represent an open and closed list of nodes. Each time we create a node that has not been explored we add it to the open list. When we choose to explore a node, we add it to the closed list. We use a while loop that will iterate under the condition that the goal state has not been reached and there are still nodes in the open list. If either of these conditions are true, then we continue to search for nodes. When a node is chosen to be explored, we try all possible moves up, left, down, right. A for loop then checks the current node's children to see if any reach the goal. If this is the case a path leading from goal state to the initial state is printed and a cost is incremented, one for every move. If none of the children match the goal state and if none of the children are already in the open list, then they are added to it.

For DFS, we use a recursive approach effectively utilizing the stack to keep track of where it has currently traveled to, and it has a worst-case runtime of $O(|V + E|)$. To start, we first send in the root node or the initial state of the puzzle and then the pathing will follow the same order of up, left, down, right. In other words, the algorithm will try to move the blank space up as much as possible until it can't anymore, then it will go left until it can either go up again or neither of the following two options are possible. If it can't go up or left then it will try to go down, and finally if it can't do those three moves then it will go right. It is important to note that once a puzzle goes left then it can never go right the next move as it would cause an infinite loop, so protections were placed to prevent that by using a list with all unique puzzle states already checked during the DFS search. With that said, this pattern of moves will continue until the goal state is found or a stack overflow occurs, which tends to happen often due to the nature of this algorithm.

Dijkstra's algorithm operates similarly to Prim's MST, as we create a SPT (shortest path tree) with a given source as the root. The SPT is used to keep track of the vertices that are within the shortest path. We then assign all vertex distance values to infinity and we initialize the root vertex that we start with to 0. While we have not included all vertices to the targeted position, we must pick a vertex that is not in the SPT and has the minimum distance value. The minimum vertex that was selected should then be added to the SPT and then update all adjacent vertices with distance values. Distance values are updated by iterating through all adjacent vertices and adding the sums of the weighted edges from the root to the current position. A priority queue is implemented to determine the next shortest path to take in our program. Dijkstra's has a worst-case runtime of $O(|V + Elog(V)|)$.

Conclusion

In the scenario that we have a goal state that is closer to the root of the graph then breadth first search would be the ideal algorithm as it traverses in level order. We found that the results show breadth first search incurring a much smaller cost in most cases when compared to depth first search since depth first search would often go down the graph as far as possible before continuing to the right of the tree. This incurs a much greater cost. Even so if the goal state was very far away from the root, we could encounter memory leaks as our implementation of breadth first search used the new keyword when creating nodes and so if enough nodes are created onto the heap this could lead to memory leakage.

If it was the case that the goal state was extremely far from the root, then it could be argued that depth first search would be a more ideal algorithm but still depending on where the goal state is and where the depth first search starts it may also incur a much larger cost that a breadth first search would. Regularly there would be stack overflows when using depth first search as it added an exceptionally substantial number of recursive calls to the stack when attempting to find the goal state.

Dijkstra’s algorithm functioned very similarly to breadth first search as it operated in the same fashion with the only addition being it took cost into account when making decisions. This would have been beneficial had the costs of eight puzzles not been artificial. To clarify, choosing a lower cost move does not necessarily lead to the goal state quicker, it only incurs less artificial cost when solving the puzzle.