**COT4400: Analysis of Algorithms**

**Final Project Report**

**Dr. Korzhova**

**Keval Solanki**

**Sidharth Anandkumar**

**Ahmed El Maliki**

**4/20/2022**

**Problem Description**

The task given was that that there is an 8-tile problem given on a 3x3 grid where each tile has a number ranging from 1-8 and one empty space. The only moves available are to slide the tiles either left, right, upwards, or downwards. There is a starting state provided that has the tiles in a random order and the objective being that using BFS, DFS and Dijkstra’s algorithm, the shortest number of steps needed be found to reach the final state while making use of the empty space.

**Our Task**

Our task was to find the shortest/cheapest number of steps needed to go from the initial state to the final state and hence solve the 8-puzzle using either BFS, DFS or Dijkstra’s algorithm.

**Performance Evaluation**

Depth First Search

Considering that the entire matrix is traveresed then the temporal complexity of DFS is O(V) where V are the values of the matrix. The value or node’s neighbors can be found by traversing the adjacency list only once in linear time. Considering sum of the adjacency list sizes of all these values which are essentially vertices in a graph is E. The complexity resolves to **O(V) + O(E) = O(V + E).**

Breadth First Search

Breadth First Search gave a complexity of **O(V+E)** space and time compelxity. This complexity is because of the fact that the implementation of the tree and nodes.

The tree branches for a worst case of “n” times so that it can reach the “m” final state assuming it is at the worst (and most liekly) bottom most node.

Dijkstra’s Algorithm

Dijkstra’s algorithm gave us a total complexity of **O(V + ElogV).** This complexity is because in the worst case, the code will have to go to every vertex (V) and every edge (E) to give us a complexity given.

**Conclusion: Comparison of Solutions**

Breadth First Search

After running the algorithm, it produced the solution with a total cost of **20** moves to go from the starting state to the final state for the first sample file.

A brief into how the BFS approach was taken is as follows. In this approach, a tree structure seemed like the best structure to utilize. This was mainly because a level-based approach starting from the root node all the way to the final node would be a much simpler way of solving the problem. The reason behind the choice of using a tree and nodes was that in this approach, each node would be representing different states of the puzzle whereby, the root node would be the initial state and the final node would represent the puzzle in its solved/final state.

To create this program, the choice of the programming language used was C++. This was primarily because of the multitude of data structures that were easily implementable using this language and majority of the pseudocode taught in class was based on C++ and its implementations of data structures.

In order to implement the BFS, there was the need of using a queue so that the algorithm could be implemented as closely to the pseudocode learnt earlier in class. Additionally, to suit the approach to solve the 8-puzzle, there was also the need of using a stack. C++ was able to easily provide these by making use of the readily available Standard Template Library.

Another important addition within the C++ program was to make use of structs. This was mainly because of the approach of using a tree where every node would be made up of a struct and each struct was representative of different stages of the puzzle. The struct compromised of a pointer to the parent node (this was to keep track of the child nodes that were different states of the puzzles). The struct also contained a 2-dimensional array that was the board where the tiles were. There were also some integer variables to keep track of each level and the cost for each node.

The program also consisted of multiple functions (helper function) that were responsible for different tasks such as printing out every state of the board i.e printing out each node, creating a new node etc. One of the most important helper functions were to create a new node. This was essential for the whole algorithm to work.

Depth first search

The DFS approach, where the initial state and final state is kept the same for all algorithms, took a total of more than **986** total steps when using the first sample file.

In this approach, we made of C++. This language was chosen to develop this solution to come up with a code because the pseudocode provided in class was making use of C++ and the code was based off the pseudocode. This language was also preferred as most of us had more experience using this language.

This solution was different from the BFS approach where it made use of nodes that were used in making the tree. Where in that approach you would traverse to the child node from the root node and the number of levels would represent the cost of moves needed to produce the solution. To come up with the DFS solution, there were no additional data structures used. The only item used was a multidimensional array to hold the puzzle at different states. The approach also makes use of multiple helper functions. These helper functions were crucial in doing repetitive tasks such as printing the contents of the multidimensional array. To do that there was a printer helper function. There was also another helper function that was responsible for performing swaps. The swaps were performed to represent moves that were made each time. The DFS algorithm was also implemented as a helper function. This function would take in the initial state and the final state of the matrix and produce the solution.

Dijkstra’s Algorithm

Using Dijikstra’s algorithm, we were able to go from the staring state to the final state within **30** steps when using the first sample file.

This approach was very different from the BFS and DFS approach. There were major changes that were need to produce the code needed to make the code work efficiently. The most important change was for Dijikstra’s algorithm, Python was used instead of C++. This was because desipite having pseduocode in C++, it was difficult to implement the code accurately in C++. Python has a multitude of built in libraries that could be directly imported. This was essential because we were able to leverage these in our solution.

**Summary of Results**

Breadth First Search, Depth First Search and Dijkstra’s algorithm were all ran to find the number of steps each took to solve the same 8-puzzle problem. This was to ensure that they were tested with the same test cases and to ensure that there were no differing variables. After running all 3 algorithms to find the shortest number of steps, it was found that BFS gave the final state within 4 steps, DFS was able to get to the final state with more than 3000 steps and Dijkstra’s algorithm was able to do so in 11 steps.

Conclusively, the best algorithm in terms of worst-case complexity and number of steps needed to get to the final states, Breadth First Search (BFS) was the best approach to the 8-puzzle problem. It had an averagely good worst-case complexity and took relatively the least number of steps to solve the puzzle.

**Discoveries**

The most interesting discovery in this entire project was how differently the different algorithms could be implemented. Especially for BFS, there were numerous implementations, and you could choose the implementation you wanted depending on your target complexity.