1. Introduction

Basic search algorithms constitute the fundamentals of Artificial Intelligence (AI). So learning their concept and being able to implement them are extremely crucial to do more advanced research on AI.

In this report I will be explaining the details of my solution algorithms, implemented programs, and the results I have concluded.

2. AI Search Methods for Sliding Puzzle

The very basics of almost anything related to AI can be found in the book of Steward Russell and Peter Norvig; "Artificial Intelligence: A Modern Approach" [1].

In this take home examination, we were asked to implement Breadth First Search (BFS), Depth First Search (DFS), Iterative Deepening Depth Limited Search DFS (IDFS), and A* algorithm with "Misplaced Tiles Heuristic" (A* Mis) and "Total Manhattan Distance Heuristic" (A* Man) on an adjustable size "Sliding Puzzle". Among these algorithms, A*, IDFS, and BFS give optimal paths, on the other hand, DFS is not optimal in the sense of detecting the shortest solution path.

Sliding puzzle requires an agent to solve the problem, which is the program written by us in that take home examination. It is a deterministic, episodic, and fully observable problem. In other words, it is one of the simplest problems to realize in the computer environment. However, sliding puzzle has a very loopy structure. The smallest loop is constructed at only two steps, i.e., moving a tile forward and backward. Although finding that two step loop is very easy, some larger loops are not trivial to detect.

While solving this puzzle, a history of opened states should be constructed. Otherwise, the search space unboundedly expands and solving even the easiest puzzles becomes a burden and sometimes impossible. I kept that history in a Sorted List structure of C#. It basically sorts the inserted nodes with a key and when asked to retrieve a node, the node is requested with that key. So search in the history becomes very efficient and fast.

For further reference, a 3x3 sliding puzzle has 181,440 solvable states, a 4x4 one has ~13 trillion steps, and a 5x5 case has ~10^{25} states. So, 3x3 is almost the only one that can be solved with all algorithms in a reasonable time.

The best way to explain a written code is to draw its flowchart. So I will be explaining the algorithms used with the flowchart. Hopefully, they will make this report easier to understand and more clear.

I will be explaining my methods in more detail in the corresponding sections in this report.

2.1. Implemented Uninformed Search Methods

Uninformed search, as the name implies, has no idea to visit which successor node would be beneficial. BFS, DFS, and IDFS are the ones that are implemented in this take home examination.

2.1.1. Breadth First Search (BFS)

The implemented algorithms are exactly the same as explained in [1] except the history keeping. BFS algorithm also works without history, but due to the loopy structure of the sliding puzzle, the search space becomes unbounded. Hence, filling up the memory before the solution is found to become extremely possible. On the algorithm tests, it is seen that BFS cannot solve puzzles exceeding 15 steps in a reasonable time (2-3 minutes). However, after implementing history, it can reach to 25 steps in a reasonable time. My BFS always gives an optimal (shortest) solution path.

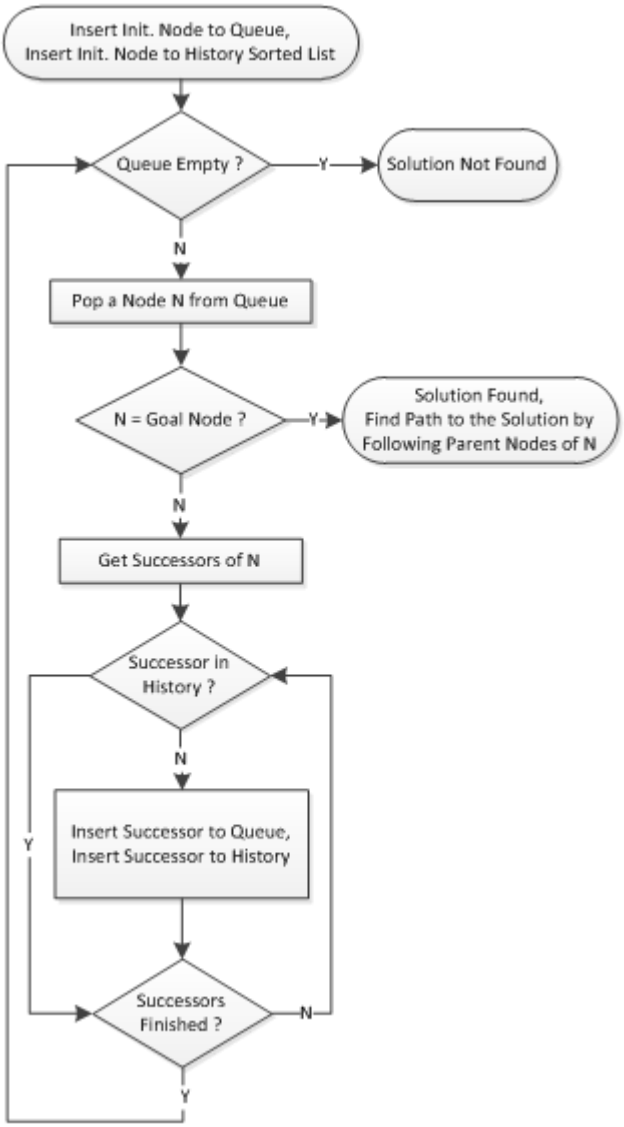

Figure 1: Implemented BFS algorithm

2.1.2. Depth First Search (DFS)

DFS is usually implemented in a recursive manner. I have implemented the recursive version too. However, I did not use it in the final version of my code. The iterative version still can be found in the source code, but the executable program uses non-recursive DFS. It is almost exactly the same as BFS, except for the fact that it uses a stack instead of a queue. In a 3x3 problem, if one is lucky, DFS can find the solution in a few 100 steps. But most of the time it searches almost the entire search space and takes around a minute to solve a 3x3 problem. Due to the fact that 4x4 and larger puzzles have much more states, DFS can be said to fail to find a solution in a reasonable time unless the user is lucky.

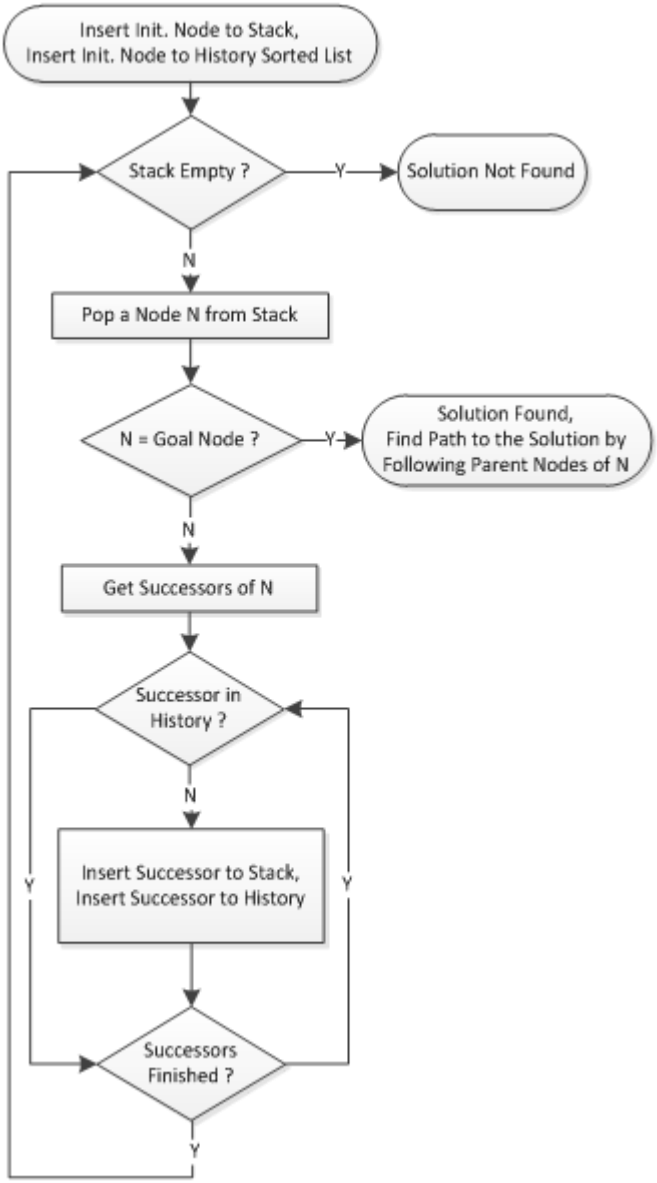

Figure 2: Implemented DFS algorithm

2.1.3. Iterative Deepening Depth Limited Depth First Search (IDFS)

I have implemented IDFS in both recursive and non-recursive forms. But only the non-recursive algorithm is used in the final program due to ease of understanding and better ability to track the code.

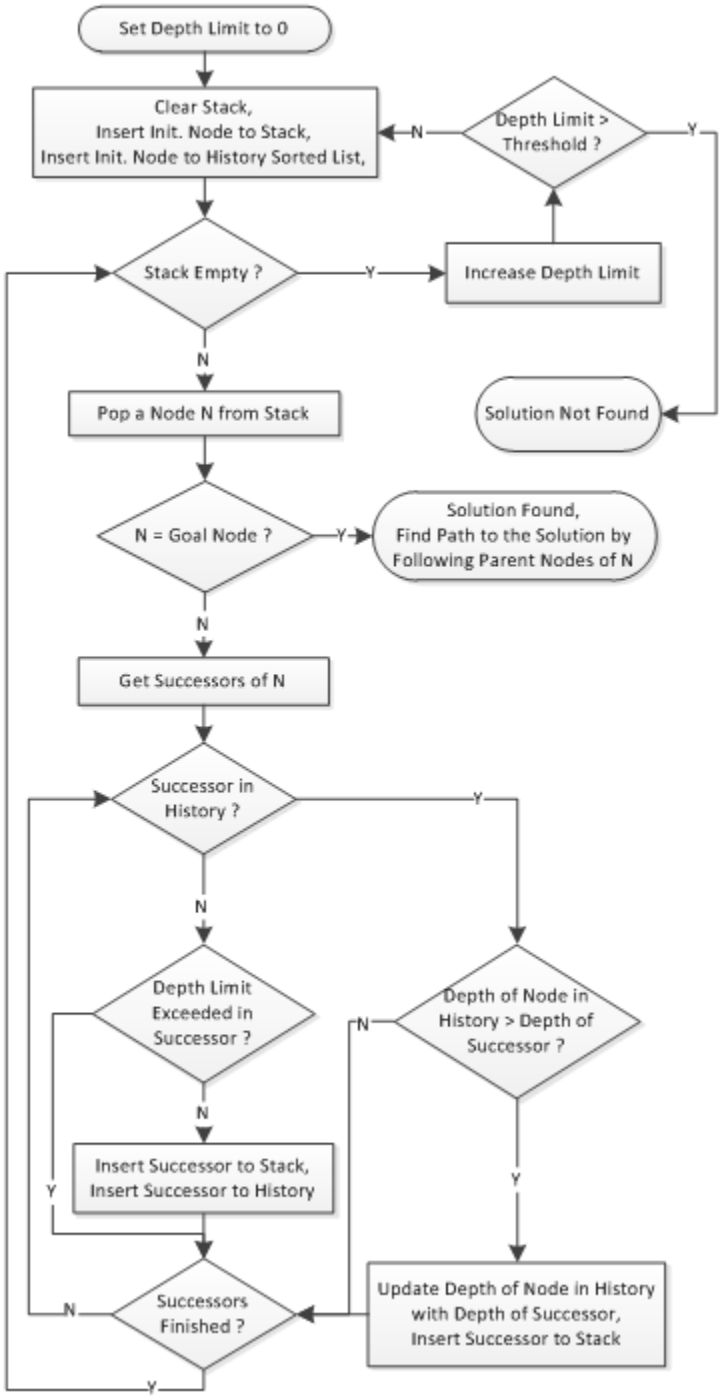

Figure 3: Implemented DFS algorithm

IDFS has a limit value, and the limit starts from 1. When no solution is found in the first iteration, the limit is increased by 1, and Depth Limited DFS starts again. The limit is increased until a threshold value. In my program that threshold value is larger than 200,000 so that for a 3x3 case a solution is always found. Actually, the limit can never reach that value in a real case since it takes really a lot of time to search the entire search space when the depth limit gets larger. Actually to search a specified depth, IDFS takes longer than BFS since IDFS iteratively searches for all levels. But the good thing with IDFS is that it requires very little memory. Storing an approximate number of nodes equal to the depth limit is enough.

For optimality, in my IDFS, I needed to keep the history but this time the history is slightly modified as seen in figure 3. Now, IDFS always gives an optimal solution. Because if a successor node is found in the history but the successor's depth is smaller than the one in the history list, it is pushed to the stack and the history is updated.

2.2. Implemented Informed Search Methods

The informed search methods can make a prediction about opening which node will be more beneficial. In this problem the cost of a node can be thought of as its depth since each move costs only 1.

A* is implemented with two heuristic functions. One of them is counting misplaced tiles, the other is counting the total Manhattan distance of all tiles to their goal states. Both heuristics are admissible, i.e., smaller than the actual solving cost. But Manhattan distance gives a closer approximation to the remaining steps than misplaced tiles.

2.2.1. A* with Misplaced Tiles Heuristic (A* Mis)

First, A* is implemented with a total misplaced tiles heuristic. It is similar to the BFS algorithm with the history keeping improvement done in the IDFS algorithm. However, this time, the item with the smallest (Cost + Heuristic) function is popped out of the list. The list is kept in a sorted manner so that accessing the smallest element is much faster than checking all elements in the list. My implemented A* with misplaced tiles heuristic always gives an optimal result. And it is noticeably faster than BFS, DFS, and IDFS. First I will explain A* with Manhattan distance and then give a flowchart of the A* algorithm since it is the same for both heuristic functions, only the calculation of the heuristic changes.

2.2.2. A* with Manhattan Distance Heuristic (A* Man)

A* algorithm with Manhattan distance heuristic gives the most optimal results among all of the implemented algorithms. It stores very little nodes, finds the solution with expanding very little steps and time. With this algorithm, the efficiency of Manhattan distance heuristic for sliding puzzle is proved.

The flowchart of the implemented A* algorithm is as follows:

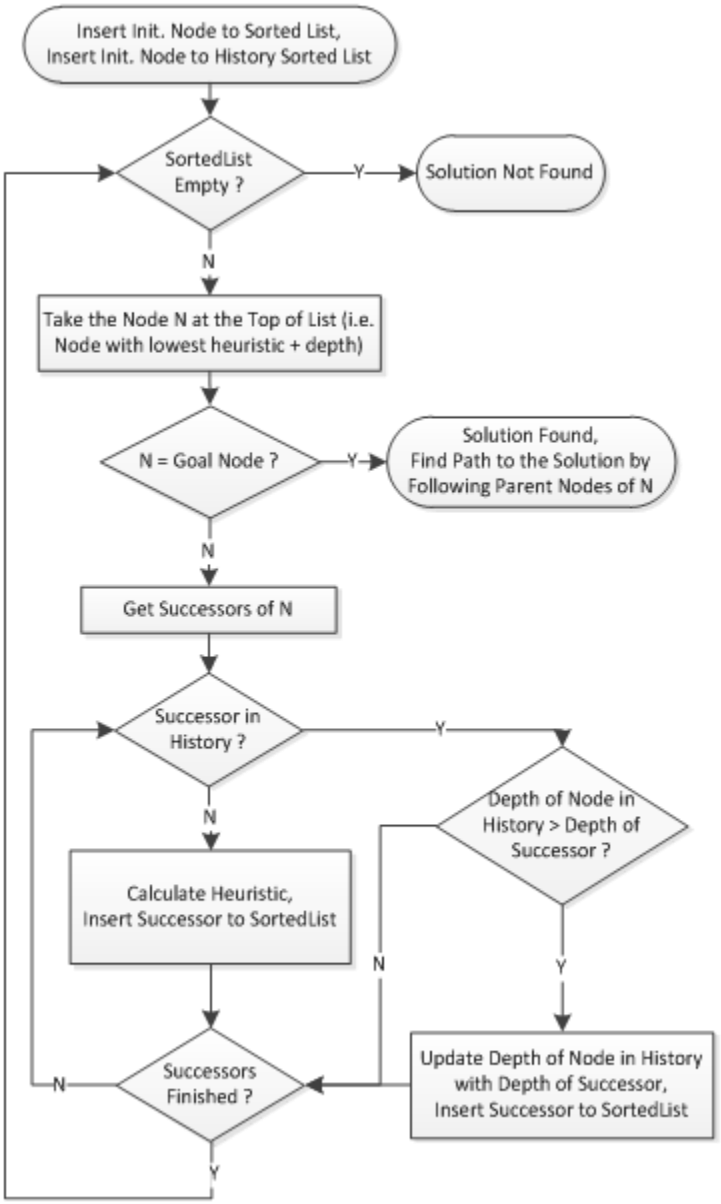

Figure 4: Implemented A* algorithm