AI Assignment - 1

Implementation and Performance Comparison of Classical Search Algorithms Using N-Puzzle Problem

Sakib Hasan
Roll 149
February 2020

Introduction

N-Puzzle is a popular problem used by AI researchers for testing new AI algorithms. In the puzzle there is an N X N grid and N^2 - 1tiles, each marked with a number from 1 to N^2 - 1. One of the positions in the grid is kept empty.

The goal of the AI is to move the tiles in such a way such that the position of the tiles reach a particular goal state, in as few steps as possible. Any tile adjacent to the empty grid position horizontally or vertically can move to the empty place.

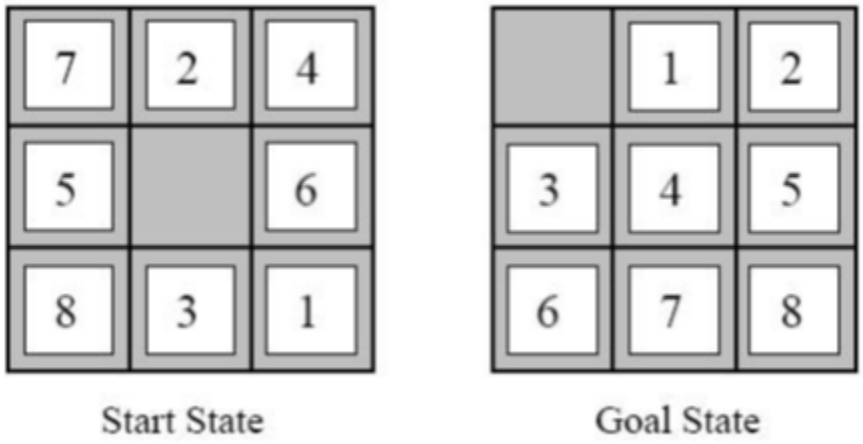


Figure 1: An Example of N-Puzzle Problem

In the assignment, we try to use 6 AI algorithms to solve the N-puzzle Problem. They are:

1. Breadth First Search (BFS)
2. Uniform Cost Search (UCS)
3. Depth Limited Search (DLS)
4. Iterative Depth First Search (IDS)
5. Greedy Best First Search (GBFS)
6. A* search

Among all these algorithms, the first 4 are uninformed search algorithms, while the last 2 are informed search algorithms. For heuristics, we use 2 heuristic functions:

1. Number of misplaced tiles
2. Manhattan distance between source and destination of all tiles

Result analysis:

Pictures of the graphs obtained from outputs for 8-puzzle (N-puzzle with 8 tiles) is given below:

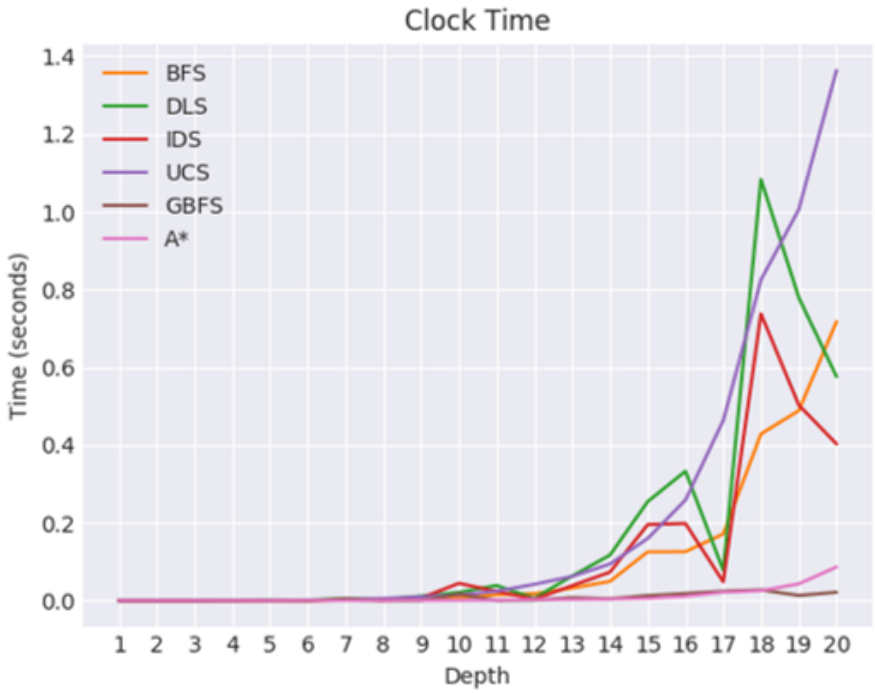


Figure 2: Time required vs depth

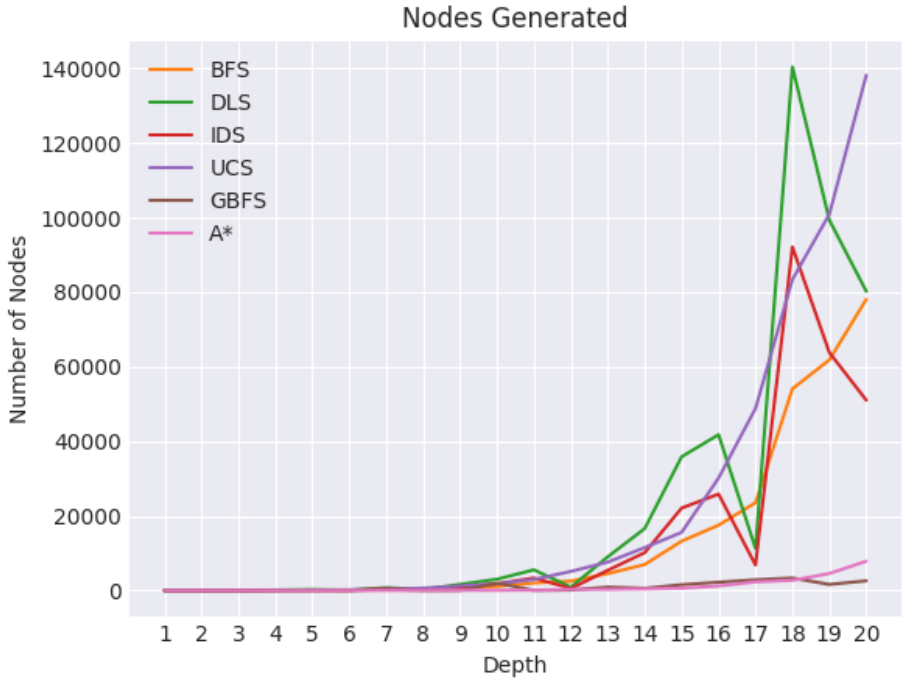


Figure 2: Node processed vs depth

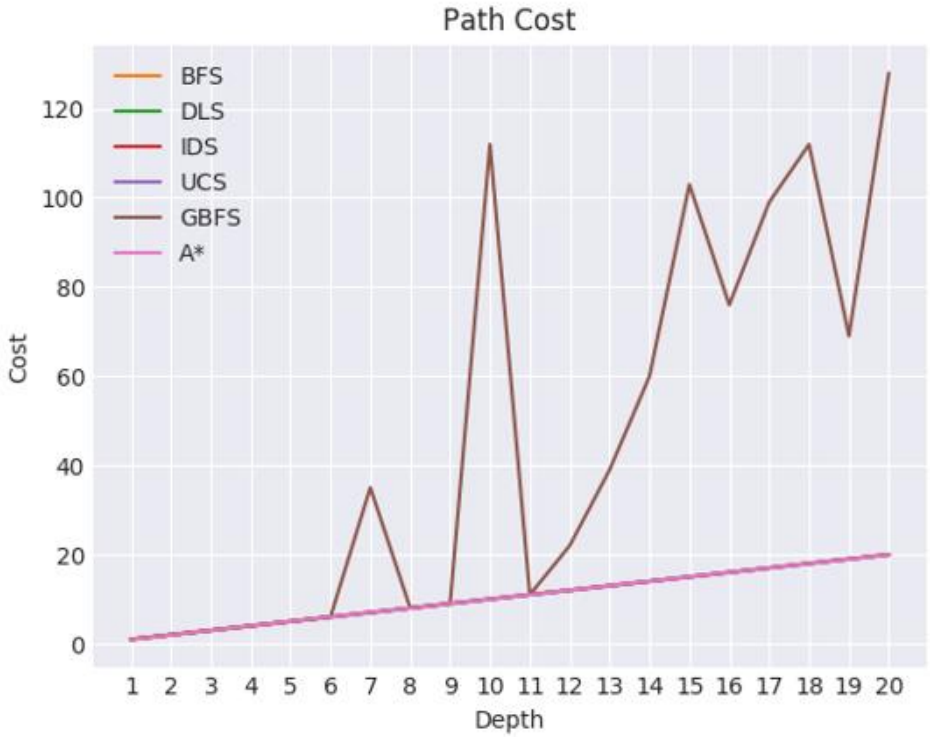


Figure 2: Solution path length vs depth

From the graph, we can reach the following conclusions:

1. The heuristic algorithms, i.e. GBFS and A* take less time than the uninformed search algorithms. This is because, the heuristics 'guide' the algorithms to the goal state, while the uninformed algorithms check all possible paths from the search tree.
2. From the 'Time Required' graph, we can see that the DLS algorithms gives random performance, i.e. sometimes it is the best algorithm, and sometimes it is the worst algorithm, based on time. DLS behaves in such a way because DLS uses depth first search. If the first subtrees analyzed by DLS contain the goal state, the result is obtained very quickly. But if the goal state is in the subtrees that are analyzed later by DLS, then DLS will find that later, and so it will take more time. But the other algorithms' performances based on time are more or less predictable.
3. Most of the algorithms solution path is exactly equal to the actual shortest path, except for DLS, GBFS and sometimes for BFS. The reason for this is DLS and GBFS are not optimal, i.e. they will find a solution, but that solution may not be the exact solution. BFS is not optimal for cases where the path cost for different paths are different. But here, the path cost is same for all paths (i.e. cost is always 1). So BFS is also optimal in this case.
4. GBFS gives a very suboptimal path length for some problems. This is because, GBFS is a greedy algorithm, i.e. it will always select the node closest to the goal state at present, but it may not eventually lead to the goal state directly.

5. Among all the 6 AI algorithms implemented, A* is the best. This is because of many reasons. One reason is that A* is heuristic, so it can take the help of the heuristic function to reach the goal state faster. Also, A* is always guaranteed to reach the optimal solution. It also uses the least amount of nodes in the search tree. So, overall, A* is the best suited algorithm for N-puzzle.

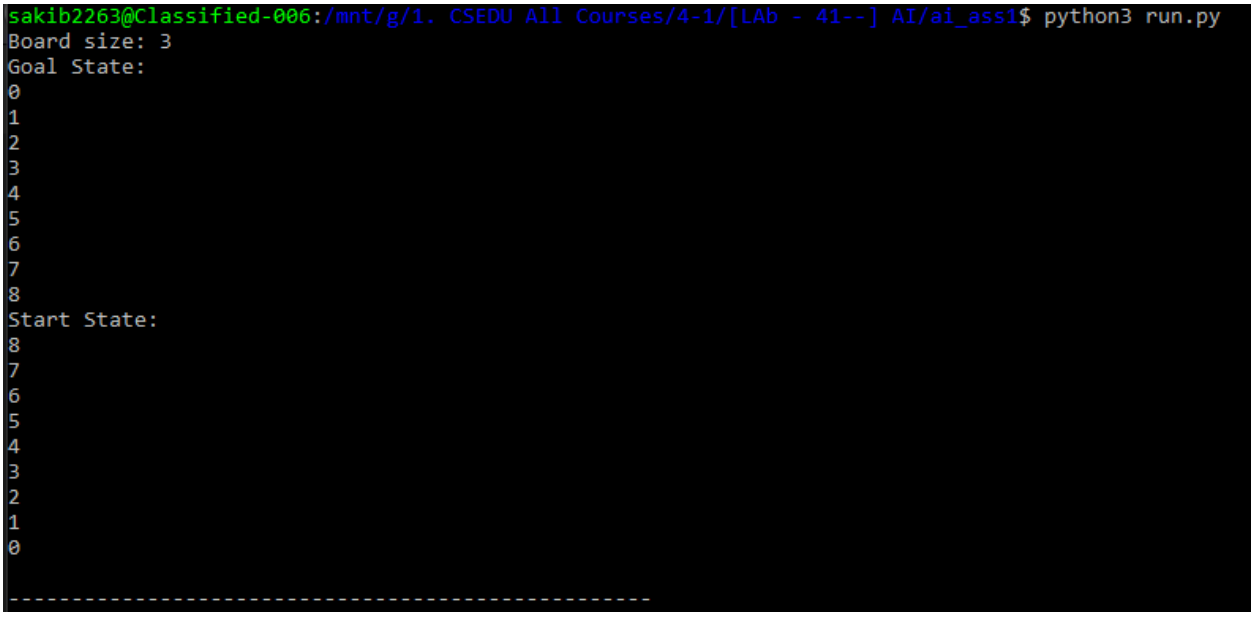
Challenges faced:

The challenges faced by me during implementation of the assignment is given below:

1. It was difficult to think about how to implement the Node class, i.e., which variables I should keep in each node.
2. It was also difficult to select which data structures to use, i.e., should I use a tuple or ***numpy*** array to store the state, should I use an array or set for 'explored' data structure.
3. Some of the algorithms were very challenging, for example, A* or GBFS or IDS, as they were not properly implemented/elaborated in the book and I had to use the help of internet.
4. Testing the program and generating the graph was also very difficult, as it took a lot of time for some of the algorithms to give solution. Also, in case of 8-puzzle, for some algorithms, some cases which have a large shortest path takes a long time to run.

Example of one test case for the generated graphs (From terminal):

Step 1:



Step 2:

```
--------------------------------------------------
Choose and algorithm (1-6) or press 0 to quit:
1. Breadth-First Search (BFS)
2. Uninformed Cost Search (UCS)
3. Depth Limited Search (DLS)
4. Iterativee Deepening Depth-First Search (IDS)
5. Greedy Best-First Search (GBFS)
6. A* Search
7. Generate Graphs on 8-Puzzle (time, nodes generated, cost)
Choice: 7
Generating random states from
Goal: [0, 1, 2, 3, 4, 5, 6, 7, 8]
Board Size: 3x3
Unique states generated for all depth 1 to 20!
Picking a random state for each depth...
Depth #1  -> [1, 0, 2, 3, 4, 5, 6, 7, 8]
Depth #2  -> [1, 2, 0, 3, 4, 5, 6, 7, 8]
Depth #3  -> [1, 4, 2, 0, 3, 5, 6, 7, 8]
Depth #4  -> [1, 2, 5, 3, 0, 4, 6, 7, 8]
Depth #5  -> [3, 0, 1, 4, 5, 2, 6, 7, 8]
Depth #6  -> [1, 5, 4, 3, 0, 2, 6, 7, 8]
Depth #7  -> [3, 1, 2, 6, 5, 8, 7, 0, 4]
Depth #8  -> [4, 3, 0, 1, 5, 2, 6, 7, 8]
Depth #9  -> [4, 0, 2, 1, 3, 8, 6, 5, 7]
Depth #10 -> [0, 1, 4, 6, 3, 2, 7, 8, 5]
Depth #11 -> [1, 7, 2, 3, 5, 8, 4, 0, 6]
Depth #12 -> [4, 6, 1, 3, 0, 2, 7, 8, 5]
Depth #13 -> [3, 0, 5, 6, 1, 2, 7, 8, 4]
Depth #14 -> [0, 4, 2, 6, 7, 5, 1, 3, 8]
Depth #15 -> [3, 2, 8, 5, 7, 0, 4, 1, 6]
Depth #16 -> [3, 8, 1, 5, 7, 2, 0, 6, 4]
Depth #17 -> [1, 5, 8, 2, 3, 0, 4, 6, 7]
Depth #18 -> [1, 2, 7, 4, 0, 5, 8, 3, 6]
Depth #19 -> [1, 0, 8, 7, 2, 5, 3, 6, 4]
Depth #20 -> [0, 1, 5, 6, 8, 7, 2, 4, 3]
Running all the algos for each depth (1 to 20):

Generating stats for BFS.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/bfs.json

Generating stats for DLS.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/dls.json

Generating stats for IDS.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/ids.json

Generating stats for UCS.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/ucs.json

Generating stats for GBFS.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/gbfs.json

Generating stats for A*.. #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20,
written to data/astar.json

All statistics have been recorded and saved in .json files
```

Step 3:

```
All statistics have been recorded and saved in .json files
Choose which graphs to generate (1-3). To exit this loop please enter 0.
Exiting will take you back initial menu!
1. 'Time' Graph
2. 'Nodes Generated' Graph
3. 'Path Cost' Graph
Choice: 1
Reading data from .json files..
Reading complete!
Initiating graph..
Plotting the points..
Adding legend and labels
Completed! Generating graph
Choose which graphs to generate (1-3). To exit this loop please enter 0.
Exiting will take you back initial menu!
1. 'Time' Graph
2. 'Nodes Generated' Graph
3. 'Path Cost' Graph
Choice: 2
Choose which graphs to generate (1-3). To exit this loop please enter 0.
Exiting will take you back initial menu!
1. 'Time' Graph
2. 'Nodes Generated' Graph
3. 'Path Cost' Graph
Choice: 3
Choose which graphs to generate (1-3). To exit this loop please enter 0.
Exiting will take you back initial menu!
1. 'Time' Graph
2. 'Nodes Generated' Graph
3. 'Path Cost' Graph
Choice:
```

Then the program will generate chosen graph(s) in Graph folder.