INTRODUCTION TO ARTIFICIAL INTELLIGENCE

HW-3

Ayben GÜLNAR-191180041

November 2022

1.Introduction

In this work compares the performance of popular AI techniques, namely the Breadth First Search, Depth First Search, A* Search, IDA, Greedy Best First Search in approaching the solution of a N-Puzzle of different size. It looks at the complexity of each algorithm as it tries to approaches the solution in order to evaluate the operation of each technique and identify the better functioning one in various cases. The N Puzzle is used as the test scenario and an application was created to implement each of the algorithms to extract results. The paper also depicts the extent each algorithm goes through while processing the solution and hence helps to clarify the specific cases in which a technique may be preferred over another.

2.Algorithms

2.1. Breadth First Search

Complete? No. if the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate.

Time? O(b^m) → m is max depth of state space terrible if m is much larger than d

Space? O(b^m), i.e., linear space!

Optimal? No e.g. if node C is a goal node, depth-first search will explore the entire left subtree

2.2. Depth First Search

Complete? No. if the left subtree were of unbounded depth but

contained no solutions, depth-first search would

never terminate.

Time? O(b^m) → m is max depth of state space terrible if m is much larger than d

Space? O(bm), i.e., linear space!

Optimal? No e.g. if node C is a goal node, depth-first search will explore the entire left subtree

2.3. Depth Limited Search

Time ? If you can access each node in O(1) time, then with branching factor of b and max depth of m, the total number of nodes in this tree would be worst case = 1 + b + b2 + … + bm-1. Using the formula for summing a geometric sequence (or even solving it ourselves) tells that this sums to = (bm - 1)/(b - 1), resulting in total time to visit each node proportional to bm. Hence the complexity = O(b^m).

Space ? The length of longest path = m. For each node, you have to store its siblings so that when you have visited all the children, and you come back to a parent node, you can know which sibling to explore next. For m nodes down the path, you will have to store b nodes extra for each of the m nodes. That's how you get an O(bm) space complexity.

Compete: if cutoff chosen appropriately then it is guaranteed to find a solution.

Optimal: it does not guarantee to find the least-cost solution

2.4. Uniform Cost Search

Complete? No. if the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate.

Time? O(b^m) → m is max depth of state space terrible if m is much larger than d

Space? O(b^m), i.e., linear space!

Optimal? No e.g. if node C is a goal node, depth-first search will explore the entire left subtree

2.5. Iterative Deepening Search:

Complete? Yes

Time? (d+1)b0 + d b1 + (d-1)b2 + … + bd = O(bd)

Space? O(bd)

Optimal? No, unless step costs are constant

2.6. Greedy Search

Complete? No – can get stuck in loops, Complete in finite space with repeated-state checking

Time? O(b^m), but a good heuristic can give dramatic improvement – m: max. depth

Space? O(b^m) – keeps all nodes in memory

Optimal? No – it can start down an infinite path and never return to try other possibilities.

2.7. A* Search

Complete? Yes

Time? Exponential O(b^m)

Space? Keeps all nodes in memory

Optimal? Yes

3. My Code Outputs

```
for n in n_inputs:
    root = random.sample(range(0, n*n), n*n)

    while not solvable(root):
        root = random.sample(range(0, n*n), n*n)

    # burayı silince random çalışıyor
    if n == 3:
        root = [1,2,3,4,5,6,7,0,8]
    if n == 4:
        root = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,15]
    if n == 5:
        root = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,24]

    #
```

Figure 1

In my homework, I defined a random initial state and a random goal state at first. If a solveable solution does not come, it changes the state, and the user can determine the numbers of the n values. Both how many n values will be entered and how many n values will be, but since it is necessary to wait for a very long time, I defined the states that require only a single move as an example, as an example, and showed the 3x3 4x4 5x5 puzzle as an example. As I said before, it works in all of them, but it takes too long, so I showed examples in these. In Figure 1, you can see the part where the inputs are given. In Figure 2, you can examine the value outputs for all algorithms and, finally, the graph in Figure 3.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 0, 24]
BFS Solution is  ['Right']
Number of explored nodes is  1
BFS Time: 6.198883056640625e-05

DFS Solution is  ['Right']
Number of explored nodes is  1
DFS Time: 6.890296936035156e-05

Greedy Solution is  ['Right']
Number of explored nodes is  2
Greedy Time: 0.00026607513427734375

A* Solution is  ['Right']
Number of explored nodes is  2
A* Time: 0.00035309791564941406

UCS Solution is  ['Right']
Number of explored nodes is  3
UCS Time: 0.00011396408081054688

IDS Solution is  ['Right']
Number of explored nodes is  1
IDS Time: 0.0001010894775390625

DFSWOL Solution is  ['Right']
Number of explored nodes is  1
DFSWOL Time: 6.008148193359375e-05
```

Figure 2 outputs for 5x5

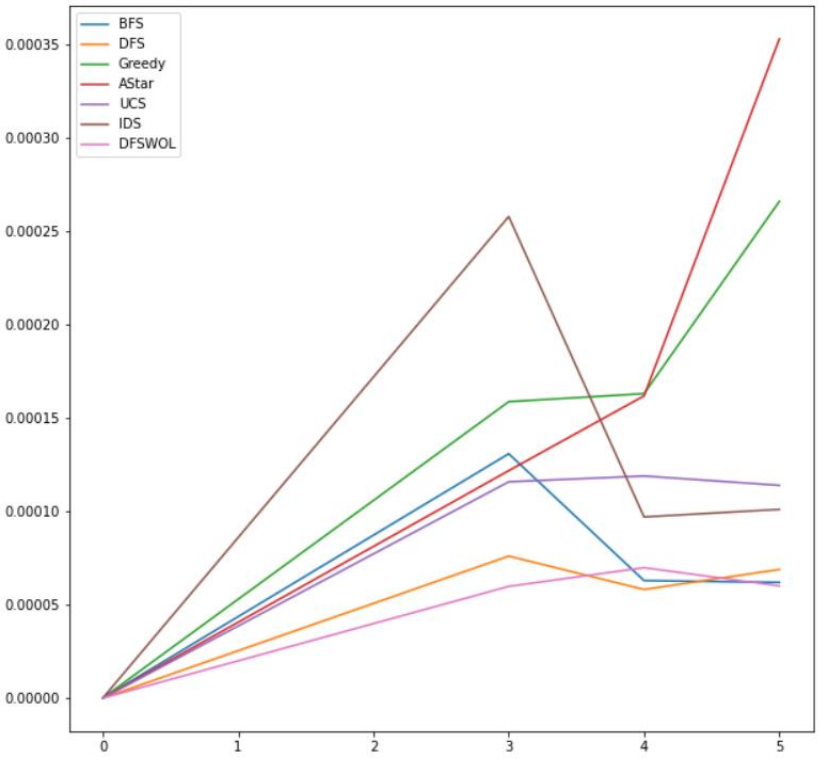

Figure 3 Line Charts

Normally, according to what I learn, the result of each algorithm differs according to each initial and target situation. Here I initialized the target states so that they only need to execute one step in order to compare. Accordingly, I have seen that heuristic algorithms generally work fast. Also, not every state is solvable. In fact, which one is faster in each case can vary.

When we run the code, you can see the individual time outputs for 3x3, 4x4 and 5x5.

4.Conclusion

In conclusion, the best approaches to apply to this dynamically scaling sliding puzzle will either be the A* or the Greed Best First Search. Greedy BFS is more memory efficient and matches the performance of A* for shorter solutions. For longer and more complex solutions, the A* is the best choice, in order to avoid possibilities of getting a suboptimal solution due to the Greedy BFS's disregard for path cost computations. A further improvement can be implemented in the form of a morphing algorithm that can opt to repeat the search in A* if the initial attempt with Greedy BFS turns out unusually long solution.