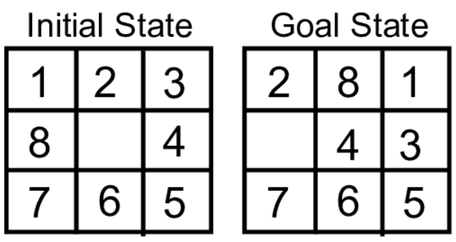
CS 554 Introduction to Artificial Intelligence

Homework 1

Name Surname (ID No)

Introduction

The N puzzle (8 - 16 puzzle) problem that emerged in the 1870s as a toy problem that causes the algorithms used to solve many conflicts today has an important place in the heuristic methods. The goal state is tried to be reached by giving the initial state with problem definition. The figure below is an excellent example of an 8-puzzle.



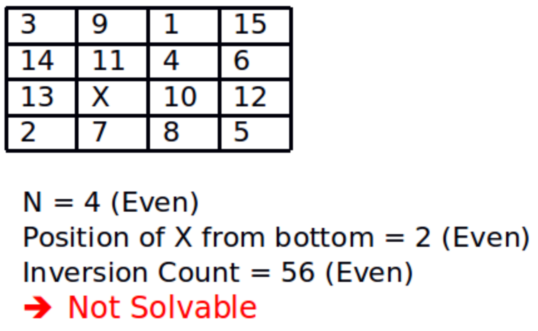
Initial State

1	2	3
8		4
7	6	5

Goal State

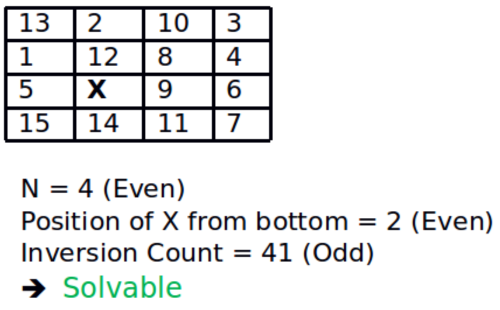
2	8	1
	4	3
7	6	5

On the other hand, the effort to prove that there are insoluble examples from the day the problem was defined started Johnson (1879). The definition of the N-puzzle problem as an NP-Hard problem was provided by Wilson's (1974) bi-connection graph. You can find one of the insoluble and solvable examples below. Today, 3x3, 8 puzzle versions have been accelerated to be solved with 31 single-tile moves (E.Korf, 2008).



3	9	1	15
14	11	4	6
13	X	10	12
2	7	8	5

N = 4 (Even)
Position of X from bottom = 2 (Even)
Inversion Count = 56 (Even)
➔ Not Solvable



13	2	10	3
1	12	8	4
5	**X**	9	6
15	14	11	7

N = 4 (Even)
Position of X from bottom = 2 (Even)
Inversion Count = 41 (Odd)
➔ Solvable

Algorithms

After the initial and goal states are given, a few questions should be asked to decide on the algorithm design. The first question is which search algorithms will be used. There are two search categories:

Uninformed:

1. Breadth-First Search
   a. Time Complexity: $O(b^d)$ → b: Branching Factor, d: depth of the shallowest solution.
2. Depth-First Search
   a. Time Complexity: $O(b*d)$ → b: Branching Factor, d: depth of the shallowest solution.
3. Uniform Cost Search
   a. Time Complexity: $O(b^{c/e})$ → b: Branching Factor, c: cost of the optimal solution,   e: min one-step cost.

Informed:

1. A* Search
2. Greedy

After making this selection, a heuristic function will be needed if the informed search algorithm is used. Three different distance functions support N-puzzle amongst the heuristic functions. The heuristic score will be omitted if you use Breadth-first, Depth-first or Iterative Deepening Search.

Heuristic Functions

1. Euclidean Distance
2. Manhattan Distance
3. Tile Out-of-Place

The last question is whether to make a tile move or multi-tile move in every move. Accordingly, the algorithm design will change significantly. One-tile is generally used in search methods like A *, Breadth-First, Depth-First for the teacher because it is an effective method to observe each step separately.

Implementation Details

As in the homework description document, the main file receives user inputs, learns which algorithm to use, and returns whether the algorithm spends time and results. I did not change this file as in Node and Graph files. Node.py defines the n matrix given as the initial state in the main file as the root node. It defines the target matrix as the target node. Graph.py creates the nodes that search algorithms must be produced from root node to target node and creates the relationship (Child, Parent) between these nodes. It provides move capabilities of algorithms with template nodes.

1. Breadth-First Search vs Depth First Search

The queue is used as a structure Breadth-First. The initial state is added to the queue. After the root is taken, the counter is incremented and continued for each node visited. The method checks whether the current node is the same as the target node. If the current state is not the target state, it turns neighbor nodes into candidates.

It differs from the Depth-First Search method because it goes to child nodes before the neighbors in Depth-First Search. If the child nodes are finished, they go back to the Neighbor nodes. In addition, Depth-First Search uses the stack structure to go to their children instead of neighbors. Thus, we have used LIFO instead of FIFO.

2. Uniform Cost Search

By its very nature, uniform cost search defines cost between nodes. So he decides whether to go to a child or neighbor by distributing the cost. The main difference of UCS from BFS and DFS is that it only uses Cost and priority Queue.

3. A* Search

As I mentioned in the introduction, A * often performs better than other search methods we use for N-puzzle. In that case, it is done using Manhattan distance. In other words, it differs from others as the informed search method. While A-star decides where to go with every step he makes, he decides by looking at the total cost of the node it is located and the cost of the node it will go to. It uses priority as in Uniform Cost Search because it has a cost-oriented structure. Manhattan distance was used unchanged.

Results

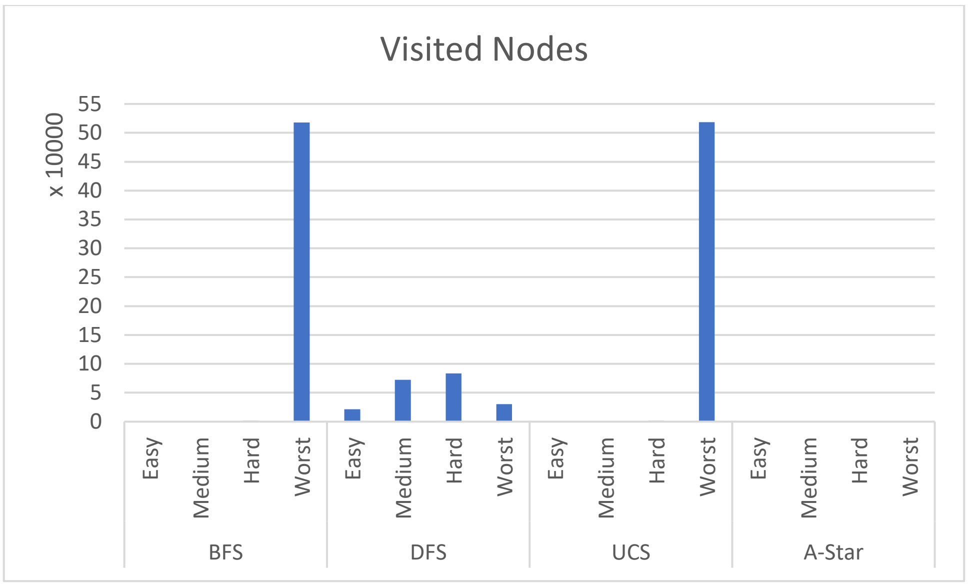
Even though n = 8 will be used in the homework explanation, I tested that the system is working in n = 3 and n = 15 cases. However, the main evaluation was tested as easy, medium, hard and the hardest in n = 8 domains.

Possible Game States:

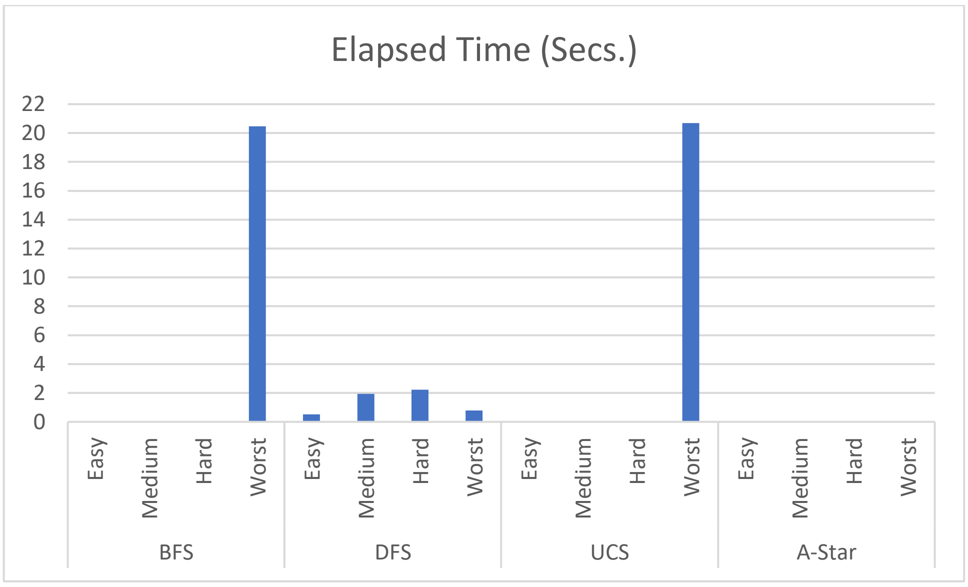
Goal:	**Easy:**	**Medium:**	**Hard:**	**Worst:**
1 2 3	1 3 4	2 8 1	2 8 1	5 6 7
8 0 4	8 6 2	0 4 3	4 6 3	4 0 8
7 6 5	7 0 5	7 6 5	0 7 5	3 2 1

Results		**Visited nodes:**	**Depth of the Graph:**	**Elapsed Time (Secs):**
BFS	**Easy**	44	5	0,001
	Medium	416	9	0,009
	Hard	1401	12	0,030
	Worst	517.721	30	20,477
DFS	**Easy**	21.250	20.891	0,521
	Medium	72.275	68.395	1,929
	Hard	83.276	77.430	2,227
	Worst	30.185	29.606	0,776
UCS	**Easy**	39	5	0,001
	Medium	344	9	0,010
	Hard	1.388	12	0,040
	Worst	518.388	30	20,692
A-Star	**Easy**	6	5	0,000
	Medium	16	9	0,001
	Hard	26	12	0,040
	Worst	719	30	0,080

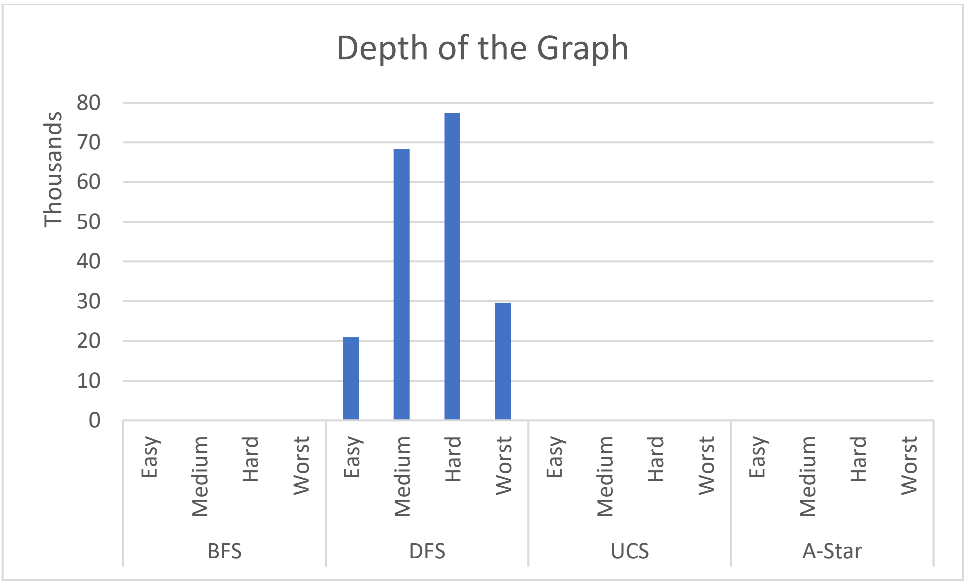
Above, there are 4 different puzzles in which the initial state is in 4 different forms, and the goal is to be achieved. These were run 3 times with BFS, DFS, UCS, A-Star Search methods and averaged. Although you can see the results in detail, I reach the tables below to clear the data according to the decision variables.



While in the table above, visiting nodes generally get the same numbers outside of DFS. In the most challenging case, DFS decreased, and BFS and UCS dramatically increased. The reason for this is made sense by defining the opposite of the desired state as the initial state in the given case. In other words, these search methods performed inefficiently as they moved away from the initial node in terms of similarity. On the other hand, A * always performs very well compared to others.



In the table above, the time to find the solution was measured in seconds. This table is observed to be proportional to the number of visited node tables. Here the exact inference is supported. Likewise, A * overperformed its competitors.



While the depth of the graph table clearly varies between 5 and 30 in all other algorithms, DFS has been found to find solutions at very deep points by far. Another inference is that the other three algorithms reach the same number of patches.

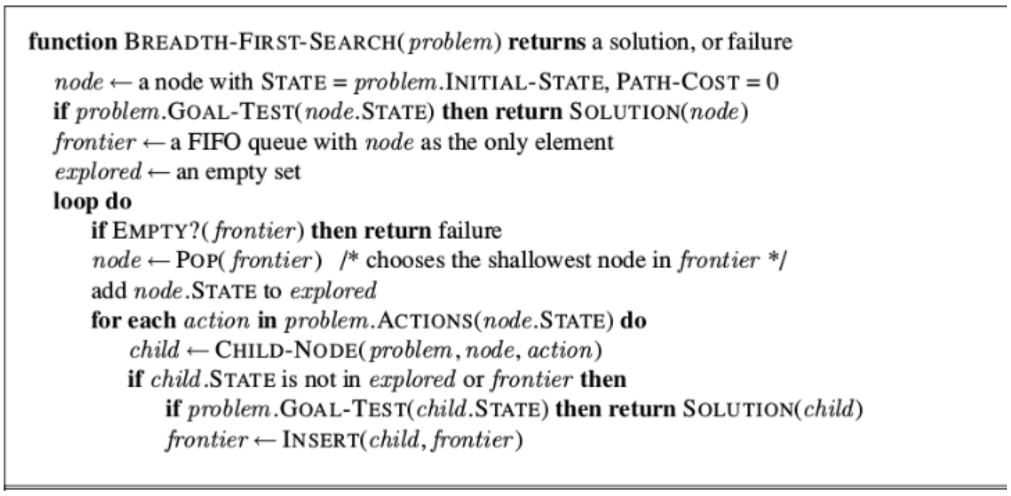
Conclusion

After all these analyses, A * is the best search method among them. In addition, the reason for this has been observed with the research made that it addresses the general search methods. On the other hand, although ranking generally seems to be A *> UCS> BFS> DFS, in some cases, it has been observed that UCS and BFS can perform poorly. On the other hand, it was observed when unsolvable puzzle setups with one tail move. In the 24 puzzle setups, it has been observed that there is a research direction towards development using different heuristic methods. In future studies, developing an approach to 24-puzzle processes will be considered a fancy research area.

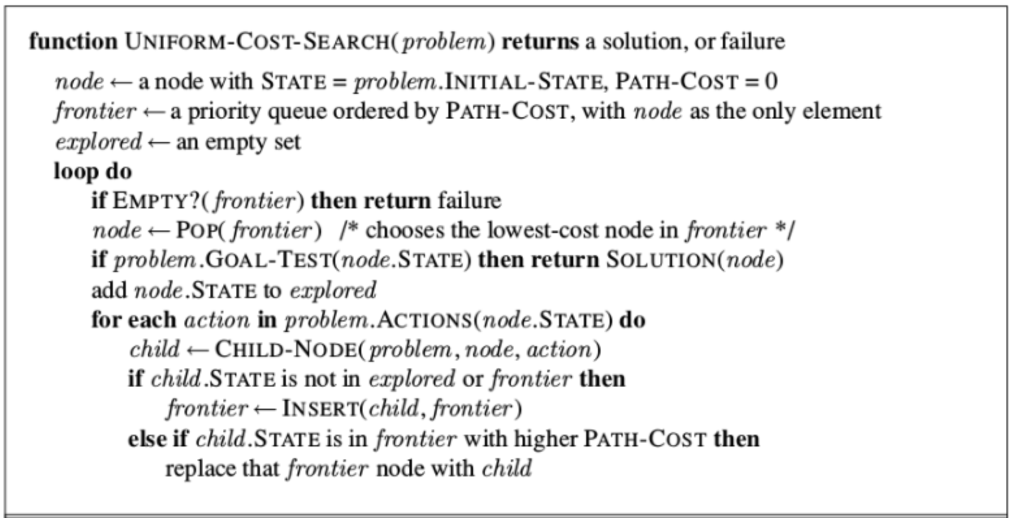
References

1. *Korf, R. E. (2000), "Recent Progress in the Design and Analysis of Admissible Heuristic Functions" (PDF), in Choueiry, B. Y.; Walsh, T. (eds.), Abstraction, Reformulation, and Approximation (PDF), SARA 2000. Lecture Notes in Computer Science, vol. 1864, Springer, Berlin, Heidelberg, pp. 45–55,*
2. Wilson, R. M. "Graph Puzzles, Homotopy, and the Alternating Group." *J. Combin. Th. Ser. B* **16**, 86-96, 1974.
3. Johnson, W. W. "Notes on the '15 Puzzle. I.' " *Amer. J. Math.* **2**, 397-399, 1879.

Appendix-1



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Appendix-2

Easy setup trial for N = 15 puzzle

Initial State	**Final State**
1 2 3 4	1 2 3 4
5 6 7 8	5 6 7 8
0 9 10 11	9 10 11 12
13 14 15 12	13 14 15 0

	BFS	**DFS**	**UCS**	**A-Star**
Visited Nodes:	5	50	41	39
Depth of the Graph:	4	4	40	4
Elapsed Time (Secs.):	0,001	0,002	0,002	0,002