Alexandria University

Faculty of Engineering

CSED 2025

Report

AI lab 1

Submitted to

Eng. Mohamed Elhabebe

Faculty of engineering Alex. University

Submitted by

Islam Yasser Mahmoud 20010312

David Michel Nagib 20010545

Marwan Yasser Sabry 20011870

Mkario Michel Azer 20011982

Faculty of engineering Alex. University

Table of Contents

s

Lab Overview:

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number 0. Given an initial state of the board, the search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8 . The search space is the set of all possible states reachable from the initial state. The blank space may be swapped with a component in one of the four directions ’Up’, ’Down’, ’Left’, ’Right’, one move at a time. The cost of moving from one configuration of the board to another is the same and equal to one. Thus, the total cost of path is equal to the number of moves made from the initial state to the goal state.

Data Structures:

DFS:

- A **stack** behaves as a frontier.
- An explored set which keeps track of all explored nodes, allowing a look up time of O(1).
- Another set which is the union of all the nodes in the frontier and the explored set.

BFS:

- A **queue** behaves as a frontier.
- An explored set which keeps track of all explored nodes, allowing a look up time of O(1).
- Another set which is the union of all the nodes in the frontier and the explored set.

A*:

- The frontier is a **min heap** allowing for retrieval of the nodes with the lowest cost + heuristic in O(lg n) time.
- An explored set which keeps track of all explored nodes, allowing a look up time of O(1).
- Another set which is the union of all the nodes in the frontier and the explored set.

Algorithms:

Same Algorithms in the lab overview papers.

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Assumptions:

There were few changes in the A*. Specifically:

- We don’t update the key in the frontier because that would involve searching, removing, and reinserting in a min heap. So, instead we assume that, when we take a state out of the frontier, if it wasn’t explored yet, then it is guaranteed to be in the optimal path from the root to that state. So, what we do is we keep the state in the min heap and if it happens that we take out a state from the heap such that the table of that state had already been explored then this one belongs to a non-optimal path. So, we skip this iteration of the loop.
- In the second condition inside the inner loop instead of:
  ```
  else if neighbor in frontier
  ```
  we check if the neighbor is not in the explored set:
  ```
  else if neighbor not in explored
  ```
  That is because looking up the neighbor state in a min heap is slower than looking for that state in a set which is done in O(1). And the reason this works is because we are guaranteed to reach this condition if the neighbor is either in the frontier, the explored set, or both, since failing the first condition means that it is not in frontier or explored.
- We have two functions for each heuristic function, one for the initial calculation of the heuristic of the state which takes O(n), and the other for the calculation of the heuristic of all the neighbors of that state which we implemented in O(1) by simply giving the child the same heuristic array of the parent and only changing the value of the moved element. (The heuristic array contains the heuristic of each element in the state separate).

Technologies used:

- Python3 as a programming language.
- PyQt5 library for implementing the GUI.

Sample runs:

Solvable states:

8	6	7
2	5	4
3		1

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	63351	27	27	27
Nodes expanded	111400	178065	3065	7682
Search depth	66123	27	27	27
Running time	0.382085 secs	1.163736 secs	0.0160036 secs	0.055012 secs

---

6	4	7
8	5	
3	2	1

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	65713	25	25	25
Nodes expanded	93115	153638	2675	4021
Search depth	65716	25	25	25
Running time	0.3730834 secs	1.117266 secs	0.0140035 secs	0.029007 secs

---

1		2
7	5	4
8	6	3

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	27265	23	23	23
Nodes expanded	158861	112996	1629	2732
Search depth	66123	23	23	23
Running time	0.4820881 secs	0.694156 secs	0.0100024 secs	0.0190051 secs

1	2	3
4	5	6
7	8	

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	10978	22	22	22
Nodes expanded	172560	81207	712	1700
Search depth	65982	22	22	22
Running time	0.6173141002655029 secs	0.5077316761016846 secs	0.004509687423706055 secs	0.012538909912109375 secs

---

1	4	2
6	5	8
7	3	

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	48564	8	8	8
Nodes expanded	54706	242	11	11
Search depth	48564	8	8	8
Running time	0.22032666206359863	0.0020041465759277344	0.0	0.0

A simulation for this test case using A* with Manhattan distance heuristic:

- **Up, left, down, left, up, right, up, left.**

1	4	2
6	5	8
7	3	

1	4	2
6	5	
7	3	8

1	4	2
6		5
7	3	8

1	4	2
6	3	5
7		8

1	4	2
6	3	5
	7	8

1	4	2
	3	5
6	7	8

1	4	2
3		5
6	7	8

1		2
3	4	5
6	7	8

	1	2
3	4	5
6	7	8

← Goal State.

8	7	6
3	4	5
	1	2

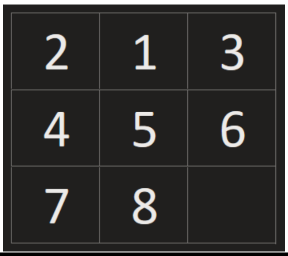
	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Cost of path	59022	30	30	30
Nodes expanded	71333	181425	11578	27647
Search depth	59022	30	30	30
Running time	0.35106873512268066	1.4053053855895996	0.09201979637145996	0.2990584373474121

Unsolvable States:

Note: any state with odd # of inversions is unsolvable.

1	2	3
4	5	6
8	7	

	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Nodes expanded	181440	181440	181440	181440
Search depth	65982	31	31	31
Running time	0.7329237461090088	1.1686410903930664	2.261714220046997	1.5829191207885742



	DFS	BFS	A* (Manhattan)	A* (Euclidean)
Nodes expanded	181440 31	181440 31	181440 31	181440 31
Search depth	65982	31	31	31
Running time	0.7679488658905029	1.2265865802764893	1.4797108173370361	2.311427116394043

Analysis:

More examples (used in our analysis):

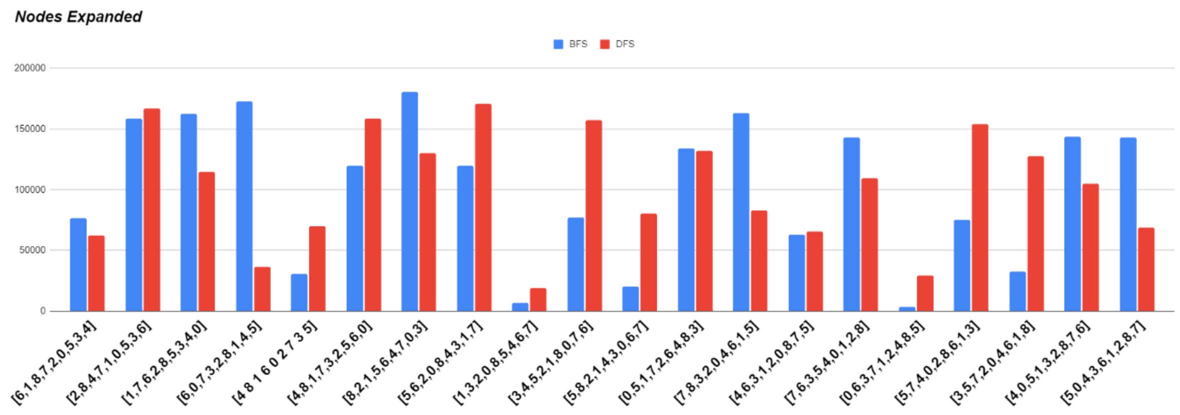
Initial State	*Cost Of Path*			
	BFS	***DFS***	***A* Manhattan***	***A* Euclidean***
['6', '1', '8', '7', '2', '0', '5', '3', '4']	*21*	*53887*	*21*	*21*
['2', '8', '4', '7', '1', '0', '5', '3', '6']	*25*	*17907*	*25*	*25*
['1', '7', '6', '2', '8', '5', '3', '4', '0']	*26*	*62186*	*26*	*26*
['6', '0', '7', '3', '2', '8', '1', '4', '5']	*27*	*34159*	*27*	*27*
['4', '8', '1', '6', '0', '2', '7', '3', '5']	*18*	*58036*	*18*	*18*
['4', '8', '1', '7', '3', '2', '5', '6', '0']	*24*	*27842*	*24*	*24*
['8', '2', '1', '5', '6', '4', '7', '0', '3']	*29*	*53083*	*29*	*29*
['5', '6', '2', '0', '8', '4', '3', '1', '7']	*23*	*13119*	*23*	*23*
['1', '3', '2', '0', '8', '5', '4', '6', '7']	*15*	*18355*	*15*	*15*
['3', '4', '5', '2', '1', '8', '0', '7', '6']	*22*	*29302*	*22*	*22*
['5', '8', '2', '1', '4', '3', '0', '6', '7']	*18*	*63114*	*18*	*18*
['0', '5', '1', '7', '2', '6', '4', '8', '3']	*24*	*51848*	*24*	*24*
['7', '8', '3', '2', '0', '4', '6', '1', '5']	*26*	*63672*	*26*	*26*
['4', '6', '3', '1', '2', '0', '8', '7', '5']	*21*	*55837*	*21*	*21*
['7', '6', '3', '5', '4', '0', '1', '2', '8']	*25*	*63531*	*25*	*25*
['0', '6', '3', '7', '1', '2', '4', '8', '5']	*14*	*27800*	*14*	*14*
['5', '7', '4', '0', '2', '8', '6', '1', '3']	*21*	*32351*	*21*	*21*
['3', '5', '7', '2', '0', '4', '6', '1', '8']	*18*	*55116*	*18*	*18*
['4', '0', '5', '1', '3', '2', '8', '7', '6']	*25*	*65215*	*25*	*25*
['5', '0', '4', '3', '6', '1', '2', '8', '7']	*25*	*57607*	*25*	*25*

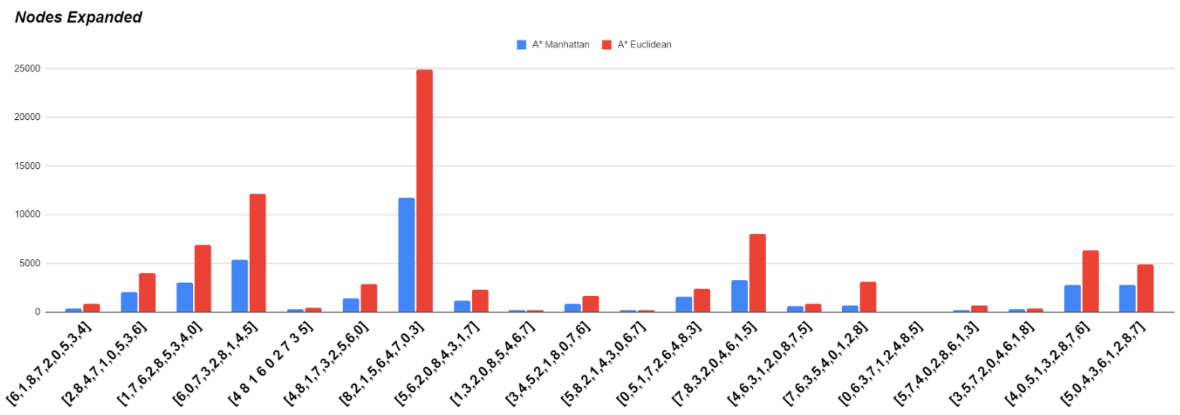
Initial State	Nodes Expanded			
	BFS	DFS	A* Manhattan	A* Euclidean
['6', '1', '8', '7', '2', '0', '5', '3', '4']	76237	62379	357	807
['2', '8', '4', '7', '1', '0', '5', '3', '6']	158598	166899	2024	4022
['1', '7', '6', '2', '8', '5', '3', '4', '0']	162593	114524	2989	6877
['6', '0', '7', '3', '2', '8', '1', '4', '5']	172572	36217	5375	12111
['4', '8', '1', '6', '0', '2', '7', '3', '5']	30670	69768	295	457
['4', '8', '1', '7', '3', '2', '5', '6', '0']	120001	158293	1367	2862
['8', '2', '1', '5', '6', '4', '7', '0', '3']	180541	130339	11706	24923
['5', '6', '2', '0', '8', '4', '3', '1', '7']	119915	170818	1184	2321
['1', '3', '2', '0', '8', '5', '4', '6', '7']	6963	18989	153	169
['3', '4', '5', '2', '1', '8', '0', '7', '6']	77088	157079	818	1610
['5', '8', '2', '1', '4', '3', '0', '6', '7']	20568	80492	165	226
['0', '5', '1', '7', '2', '6', '4', '8', '3']	134053	132090	1583	2391
['7', '8', '3', '2', '0', '4', '6', '1', '5']	163288	83006	3224	8007
['4', '6', '3', '1', '2', '0', '8', '7', '5']	63150	65639	587	850
['7', '6', '3', '5', '4', '0', '1', '2', '8']	143161	109734	658	3122
['0', '6', '3', '7', '1', '2', '4', '8', '5']	3157	29079	19	26
['5', '7', '4', '0', '2', '8', '6', '1', '3']	75119	154199	193	663
['3', '5', '7', '2', '0', '4', '6', '1', '8']	32256	127358	246	361
['4', '0', '5', '1', '3', '2', '8', '7', '6']	143603	104815	2771	6325
['5', '0', '4', '3', '6', '1', '2', '8', '7']	143355	69049	2744	4891

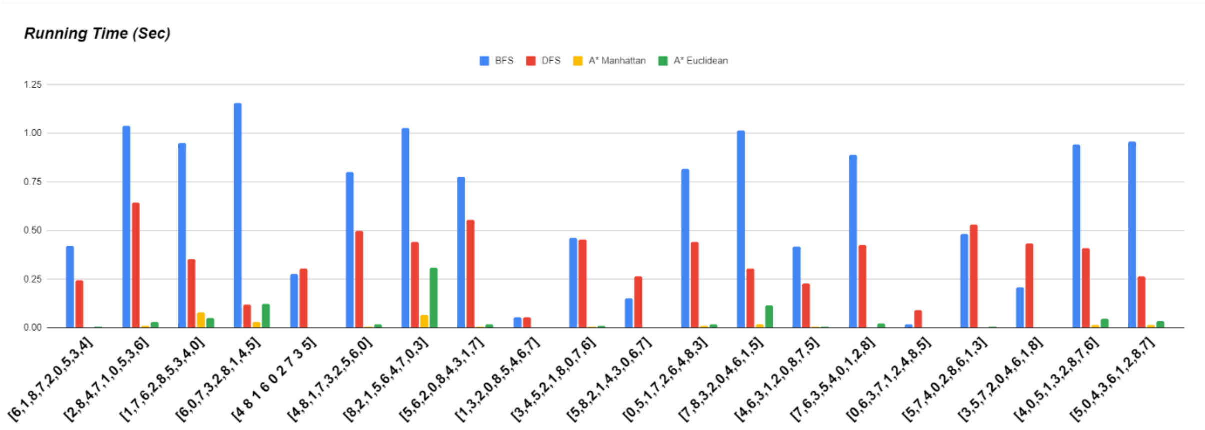
Initial State	Maximum Depth			
	BFS	DFS	A* Manhattan	A* Euclidean
['6', '1', '8', '7', '2', '0', '5', '3', '4']	21	53887	21	21
['2', '8', '4', '7', '1', '0', '5', '3', '6']	25	66056	25	25
['1', '7', '6', '2', '8', '5', '3', '4', '0']	26	65982	26	26
['6', '0', '7', '3', '2', '8', '1', '4', '5']	27	34159	27	27
['4', '8', '1', '6', '0', '2', '7', '3', '5']	18	58036	18	18
['4', '8', '1', '7', '3', '2', '5', '6', '0']	24	65982	24	24
['8', '2', '1', '5', '6', '4', '7', '0', '3']	29	66123	29	29
['5', '6', '2', '0', '8', '4', '3', '1', '7']	23	66125	23	23
['1', '3', '2', '0', '8', '5', '4', '6', '7']	15	18355	15	15
['3', '4', '5', '2', '1', '8', '0', '7', '6']	22	66488	22	22
['5', '8', '2', '1', '4', '3', '0', '6', '7']	18	63114	18	18
['0', '5', '1', '7', '2', '6', '4', '8', '3']	24	66126	24	24
['7', '8', '3', '2', '0', '4', '6', '1', '5']	26	63672	26	26
['4', '6', '3', '1', '2', '0', '8', '7', '5']	21	55837	21	21
['7', '6', '3', '5', '4', '0', '1', '2', '8']	25	66056	25	25
['0', '6', '3', '7', '1', '2', '4', '8', '5']	14	27800	14	14
['5', '7', '4', '0', '2', '8', '6', '1', '3']	21	66125	21	21
['3', '5', '7', '2', '0', '4', '6', '1', '8']	18	66122	18	18
['4', '0', '5', '1', '3', '2', '8', '7', '6']	25	66123	25	25
['5', '0', '4', '3', '6', '1', '2', '8', '7']	25	57611	25	25

Initial State	Running Time			
	BFS	DFS	A* Manhattan	A* Euclidean
['6', '1', '8', '7', '2', '0', '5', '3', '4']	0.4210689068	0.242647171	0.0009727478027	0.005033016205
['2', '8', '4', '7', '1', '0', '5', '3', '6']	1.038071871	0.6434798241	0.01095271111	0.02865624428
['1', '7', '6', '2', '8', '5', '3', '4', '0']	0.9491944313	0.3523836136	0.0765542984	0.05100393295
['6', '0', '7', '3', '2', '8', '1', '4', '5']	1.157402039	0.1166605949	0.02798962593	0.1246051788
['4', '8', '1', '6', '0', '2', '7', '3', '5']	0.2765614986	0.3025608063	0.0009891986847	0.002999544144
['4', '8', '1', '7', '3', '2', '5', '6', '0']	0.8026366234	0.4980511665	0.007031679153	0.01900053024
['8', '2', '1', '5', '6', '4', '7', '0', '3']	1.028384209	0.4424083233	0.06561136246	0.3066852093
['5', '6', '2', '0', '8', '4', '3', '1', '7']	0.7780368328	0.553539753	0.006000041962	0.01600193977
['1', '3', '2', '0', '8', '5', '4', '6', '7']	0.05261683464	0.05202198029	0.0009927749634	0.0009939670563
['3', '4', '5', '2', '1', '8', '0', '7', '6']	0.4630115032	0.4550685883	0.003988981247	0.0100004673
['5', '8', '2', '1', '4', '3', '0', '6', '7']	0.1505782604	0.2631604671	0.0009989738464	0.001000642776
['0', '5', '1', '7', '2', '6', '4', '8', '3']	0.8161628246	0.4434447289	0.007993459702	0.01756501198
['7', '8', '3', '2', '0', '4', '6', '1', '5']	1.015086412	0.304725647	0.01759266853	0.1135571003
['4', '6', '3', '1', '2', '0', '8', '7', '5']	0.4169311523	0.2268121243	0.003992080688	0.004997014999
['7', '6', '3', '5', '4', '0', '1', '2', '8']	0.8903532028	0.4262974262	0.003027677536	0.02056956291
['0', '6', '3', '7', '1', '2', '4', '8', '5']	0.01899647713	0.08861851692	0	0
['5', '7', '4', '0', '2', '8', '6', '1', '3']	0.4824697971	0.5298581123	0.0009906291962	0.004036188126
['3', '5', '7', '2', '0', '4', '6', '1', '8']	0.2062618732	0.4321622849	0.0009906291962	0.003006696701
['4', '0', '5', '1', '3', '2', '8', '7', '6']	0.9438154697	0.4082670212	0.01499032974	0.04463434219
['5', '0', '4', '3', '6', '1', '2', '8', '7']	0.9578974247	0.2640662193	0.01500368118	0.03496670723

Leading to:







Observations:

Regarding BFS & DFS:

- It is tight between DFS and BFS in **time** manner, but the **cost** is much less in BFS as it always finds the optimal solution unlike DFS.
- Arguably the # of nodes expanded, and the path is less in the BFS case too.

Regarding BFS & A*:

- The usage of the heuristic makes us explore less states and this leads sometimes to less time too. but regarding the cost and the optimality they are equal.

Regarding A* (Manhattan) & A* (Euclidean):

First of all, they are both admissible as:

- Manhattan:
  - The Manhattan heuristic calculates the sum of the Manhattan distances (also known as "L1 distances") between each tile's current position and its goal position regardless that these movements should go through the free tile only (Which really happens and for sure cost more) that's why it is admissible because **it never overestimates the cost** to reach the goal. In other words, it provides **a lower-bound estimate** of the actual cost.
  - Consider the nature of the 8-puzzle problem: Moving a tile from its current position to its goal position requires a minimum of the Manhattan distance between these two positions. Therefore, the sum of these minimum distances for all tiles provides a lower-bound estimate of the total number of moves required to reach the goal state.
- Euclidean:
  - The Euclidean heuristic calculates the Euclidean distance (L2 distance) between each tile's current position and its goal position. Similar to the Manhattan heuristic, it is admissible because it also provides **a lower-bound estimate.**
  - In the 8-puzzle problem, tiles can move diagonally (as well as horizontally and vertically). The Euclidean distance accounts for these diagonal moves, making it a **smaller estimate** of the true cost to reach the goal.

Despite that both Manhattan and Euclidean are admissible heuristics in this game, but the Manhattan over perform Euclidean In the examples provided especially in time. We can also consider the following metrics:

- The Manhattan is better as:
  - Less complexity in the computation.
  - Much more accurate.

- The Euclidean is worse as:
    - More complexity in the computation (the square root)
    - Less accurate.

Another important reason for these observations is that Manhattan is more admissible than Euclidean and the reason behind it is that Manhattan always look for the sum of horizontal and vertical moves to the target unlike Euclidean which looks for the Diagonal moves also as it depends on the hypotenuse. And with a simple math calculation we can reach that h(Manhattan) will always be greater than or equal to h(Euclidean) as in a right-angled triangle the sum of both sides will always be greater than the hypotenuse.