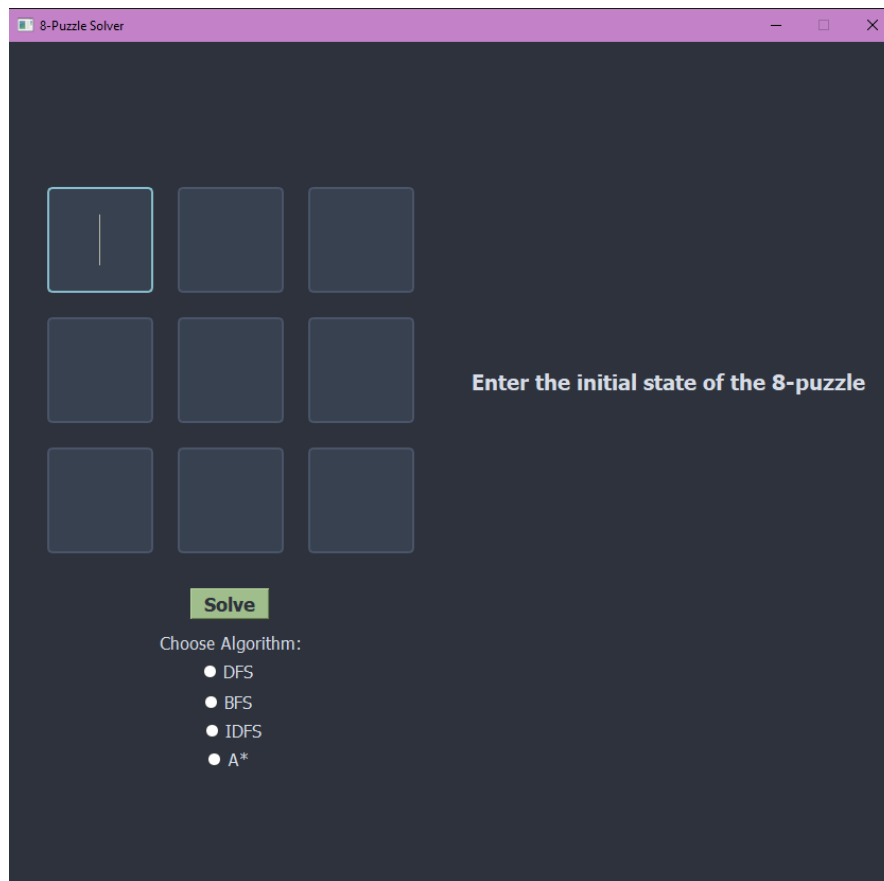


AI Lab (1)

8-Puzzle Solver

Name	ID
Ranime Ahmed Elsayed Shehata	21010531
Rowan Gamal Ahmed Elkhoully	21010539
Karene Antoine Nassif	21010969



The screenshot shows a web application titled "8-Puzzle Solver" in a purple header bar. The main area has a dark blue background. On the left, there is a 3x3 grid of square tiles. The top-left tile is highlighted with a light blue border and contains a vertical line. To the right of the grid, the text "Enter the initial state of the 8-puzzle" is displayed. Below the grid, there is a green "Solve" button. Underneath the button, the text "Choose Algorithm:" is followed by four radio button options: "DFS", "BFS", "IDFS", and "A*".

Algorithms Implemented:

- 1- DFS
 - 2- BFS
 - 3- Iterative DFS
 - 4- A*
 - Manhattan
 - Euclidean
-

Data Structures Used:

1- DFS:

- **frontier**: A stack (LIFO) to hold states to be explored.
- **explored**: A set to track visited states and avoid revisiting.
- **frontier_set**: A set for fast lookup of states in the frontier.
- **cost**: A dictionary to track the depth (cost) of each state.
- **parent**: A dictionary mapping each state to its predecessor, used to reconstruct the solution path.
- **path**: A list holding the final path from the initial to the goal state.

2- BFS:

- **frontier**: A queue (FIFO).
- **expanded**: A list to track visited states.
- **Added_nodes**: A set to track the states that have been added to the frontier before.
- **parent**: A dictionary that maps each state to its predecessor, it is used to get the path.
- **depth**: A dictionary to track the depth (cost) of each state.
- **path**: A list holding the final path from the initial to the goal state.

3- Iterative DFS:

- **frontier**: A list (stack) that holds the path of states explored up to the current depth.
- **explored**: A set to track visited states and avoid revisiting them.
- **nodes_expanded**: Tracks the number of nodes (states) expanded at each depth limit.

4- **A***:

- **frontier**: min heap (key is $f(n)$, value is node state)
- **expanded**: list (ordered)
- **frontier_dict**: dictionary for $f(n)$ maps frontier state $\rightarrow f(n)$
- **parent**: dictionary maps child \rightarrow parent
- **depth_per_node**: dictionary maps node to its depth

How does the Algorithm work?

1- **DFS**:

- **Initial Check**: It first checks if the puzzle is solvable using an inversion count. If unsolvable, it returns -1.
- **DFS Loop**: States are explored by popping from the frontier, generating neighboring states (possible moves), and adding unexplored neighbors to the frontier.
- **Goal Check**: If the current state matches the goal, the path is reconstructed, and statistics like nodes expanded, depth, and runtime are returned.
- **Termination**: The search ends when the goal is found or all states are explored.

DFS explores deep paths first and backtracks when necessary, ensuring all possible paths are explored (Expand deepest first).

2- **BFS**:

- **Initial Check**: It first checks if the puzzle is solvable using an inversion count. If unsolvable, it returns -1.
- **BFS Loop**: Initializes with the starting state in the queue. In each loop iteration, the algorithm dequeues the current state, adds it to the expanded list, and checks if it's the goal. If so, it records the path, nodes expanded, search depth, runtime, and path cost.
Otherwise, neighboring states (possible moves) of the current state are generated. Unexplored neighbors are added to the frontier, tracking each one's parent and depth.
- **Goal Check and Termination**: The search ends if the goal state is reached, and it provides the path from the initial state to the goal state; otherwise, it concludes with "No solution found."

BFS explores level by level, ensuring the shortest path to the goal is found.

3- Iterative DFS:

- **Initial Check:** The algorithm checks if the puzzle is solvable by counting inversions. If unsolvable, it returns -1.
- **Iterative Deepening:** It starts with a depth limit of 0, performing a depth-limited DFS at each depth and incrementing the limit until the goal is found.
- **Depth-Limited Search:** Recursively explores all states up to the current depth limit. If the goal is found, the search ends. If not, increment the depth and re-explore all the states from the root node up to the new current depth limit.
- **Termination:** When the goal is reached, it returns the solution path, nodes expanded, depth, and running time.

This method combines the completeness of BFS with the memory efficiency of DFS.

4- A*:

- The first step before the algorithm is to add the initial state to the frontier
 - While the frontier is not empty, we pop min out of the heap (frontier):
 - Update max depth so far (if it changed)
 - Check if it's the goal or not:
 - If yes, break out of the loop.
 - If not, we calculate the zero position to get all possible neighbors
 - For each neighbor, we calculate $f(n)$
 - $f(n) = g(n) + h(n)$
 - $\rightarrow g(n)$: cost of path so far (level of node)
 - $\rightarrow h(n)$: heuristic calculated by Manhattan/Euclidean rules.
 - We only add to frontier if it's not in expanded or if it's in frontier but $f(n)$ now is smaller \rightarrow update it
 - After adding it to Frontier we update the parent dictionary
-

Heuristic Functions with A*:

Using:

```
x_current = (i%3)
y_current = int(i//3)
x_goal = (int(str_state[i])%3)
y_goal = int(str_state[i]) // 3
```

1- Manhattan:

```
h += abs(x_current - x_goal) + abs(y_current - y_goal)
```

2-Euclidean:

```
h += ((x_current - x_goal)**2 + (y_current - y_goal)**2)**0.5
```

Which one is better? “**Manhattan vs. Euclidean**”:

- Manhattan is faster.
 - Both take the same space but Manhattan time is better. Therefore, Manhattan is better.
 - The **reason** is that Manhattan calculation is easier/faster than Euclidean, it only requires a different operation. Euclidean needs powers and square root computations.
 - The **reason in this puzzle especially** is that it requires moving in up, down, left, right only. Euclidean helps if moving diagonally was allowed, but in this case, manhattan will be faster to compute and more accurate
-

Sample Runs:

Enter the initial state of the 8-puzzle

Solve

Choose Algorithm:

- ☐ DFS
- ☐ BFS
- ☐ IDFS
- ☐ A*

8	6	7
2	5	4
3	0	1

Enter the initial state of the 8-puzzle

Solve

Choose Algorithm:

- ☐ DFS
- ☒ BFS
- ☐ IDFS
- ☐ A*

8	6	7
2	5	4
3		1

Previous

Play Again

Next

Step: 1/28

Search Depth: 27

Nodes Expanded: 176297

Total Cost to reach the Goal: 27

Running Time: 0.9065 seconds

8	6	7
2		4
3	5	1

8	6	7
2	4	
3	5	1

8	6	7
2	4	1
3	5	

8	6	7
2	4	1
3		5

8	6	7
2		1
3	4	5

8		7
2	6	1
3	4	5

	8	7
2	6	1
3	4	5

2	8	7
	6	1
3	4	5

2	8	7
3	6	1
	4	5

2	8	7
3	6	1
4		5

2	8	7
3		1
4	6	5

2		7
3	8	1
4	6	5

	2	7
3	8	1
4	6	5

3	2	7
	8	1
4	6	5

3	2	7
4	8	1
	6	5

3	2	7
4	8	1
6		5

3	2	7
4		1
6	8	5

3	2	7
4	1	
6	8	5

3	2	
4	1	7
6	8	5

3		2
4	1	7
6	8	5

3	1	2
4		7
6	8	5

3	1	2
4	7	
6	8	5

3	1	2
4	7	5
6	8	

3	1	2
4	7	5
6		8

3	1	2
4		5
6	7	8

3	1	2
	4	5
6	7	8

	1	2
3	4	5
6	7	8

Algorithm Comparisons:

1. Total Cost to reach goal:

	Test 1 ("123456078")	Test 2 ("867254301")	Test 3 ("876543201")	Test 4 ("536247108")
DFS	49124	63413	4581	45039
BFS	22	27	27	27
IDS	22	27	27	27
A* (Manhattan)	22	27	27	27
A* (Euclidean)	22	27	27	27

2. Search Depth:

	Test 1 ("123456078")	Test 2 ("867254301")	Test 3 ("876543201")	Test 4 ("536247108")
DFS	65982	66123	4581	45039
BFS	22	27	27	27
IDS	22	27	27	27
A* (Manhattan)	22	27	27	27
A* (Euclidean)	22	27	27	27

3. Nodes Expanded:

	Test 1 ("123456078")	Test 2 ("867254301")	Test 3 ("876543201")	Test 4 ("536247108")
DFS	136088	111245	4703	49551
BFS	75375	176297	177176	171576
IDS	141621	6469949	4853421	1534860
A* (Manhattan)	463	2513	190	1644
A* (Euclidean)	116136	433985	433985	433985

4. Running Time:

	Test 1 ("123456078")	Test 2 ("867254301")	Test 3 ("876543201")	Test 4 ("536247108")
DFS	0.5023	0.4126	0.00164	0.1875
BFS	0.4780	0.9497	0.8781	0.9153
IDS	1.3974	32.6821	28.7334	24.1377
A* (Manhattan)	0.0050	0.0309	0.0076	0.0205
A* (Euclidean)	0.0333	0.6988	0.1775	0.8974

Time and Space Complexities:

	Time Complexity	Space Complexity
BFS	b^d	b^d
DFS	b^m	$b \cdot m$
IDS	b^d	$b \cdot d$
A*	b^d	b^d

Solvability Test:

The solvability test for the 8-puzzle problem determines whether a given puzzle configuration can be solved by counting the number of inversions in the tile arrangement. An inversion occurs when a tile with a higher number precedes a tile with a lower number as you read the puzzle grid left to right, top to bottom (ignoring the empty tile, usually represented by 0).

❖ Steps of the Solvability Test:

1. Convert the Puzzle to a Linear Form:
 - The 8-puzzle is represented as a 3x3 grid. To test for solvability, the grid is "flattened" into a single list of numbers (excluding the 0 or empty tile).
2. Count Inversions:
 - Iterate through the list and count how many times a larger numbered tile comes before a smaller numbered tile. For each such instance, increment the inversion count.
3. Determine Solvability:
 - Even number of inversions: The puzzle is solvable.
 - Odd number of inversions: The puzzle is unsolvable.

```
def is_solvable(self):
    inversions = 0
    initial = str(self.initial_state)
    if len(str(self.initial_state)) == 8:
        initial = "0" + initial
    for i in range(len(initial)):
        if initial[i] == "0":
            continue
        for j in range(i+1, len(initial)):
            if initial[j] == "0":
                continue
            if int(initial[i]) > int(initial[j]):
                inversions += 1
    print(inversions)
    return inversions % 2 == 0
```

Unsolvable Example: “345216780”

The screenshot shows a web-based 8-puzzle solver interface. On the left, there is a 3x3 grid of tiles representing the puzzle state. The tiles contain the numbers 3, 4, 5 in the top row; 2, 1, 6 in the middle row; and 7, 8, 0 in the bottom row. To the right of the grid, the text "Enter the initial state of the 8-puzzle" is displayed. Below the grid is a green "Solve" button. Underneath the button, the text "Choose Algorithm:" is followed by four radio button options: DFS, BFS, IDFS, and A*. The A* option is currently selected. On the right side of the interface, a small dialog box with a purple title bar and a yellow warning icon is open. The dialog box contains the text "No Solution" and "No solution found for this puzzle." with an "OK" button at the bottom.

Bonus Section: (GUI) - with every step printed:

8	6	7
2	5	4
3	0	1

Enter the initial state of the 8-puzzle

Solve

Choose Algorithm:

- ☐ DFS
- ☒ BFS
- ☐ IDFS
- ☐ A*

8	6	7
2	5	4
3		1

Previous **Play Again** **Next**

Step: 1/28

Search Depth: 27

Nodes Expanded: 176297

Total Cost to reach the Goal: 27

Running Time: 0.9065 seconds

Some Validations:

3	1	5
2	1	6
7	8	0

Solve

Choose Algorithm:

- ☐ DFS
- ☒ BFS
- ☐ IDFS
- ☐ A*

Enter the initial state of the 8-puzzle

Input Error

⚠ Please enter distinct numbers (no duplicates).

OK

3	1	5
2	9	6
7	8	0

Solve

Choose Algorithm:

- ☐ DFS
- ☒ BFS
- ☐ IDFS
- ☐ A*

Enter the initial state of the 8-puzzle

Input Error

⚠ Please enter numbers within the range of 0 to 8.

OK

3

1

5

2

75

6

7

8

0

Solve

Choose Algorithm:

☐ DFS

☒ BFS

☐ IDFS

☐ A*

Enter the initial state of the 8-puzzle

Input Error

!

Please enter numbers within the range of 0 to 8.

OK

3

1

5

6

7

8

0

Solve

Choose Algorithm:

☐ DFS

☒ BFS

☐ IDFS

☐ A*

Enter the initial state of the 8-puzzle

Input Error

!

Please fill the grid before solving.

OK

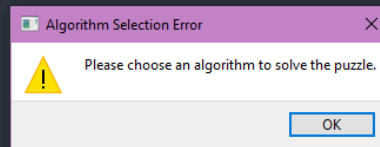
1	2	8
3	0	4
5	6	7

Solve

Choose Algorithm:

- ☒ DFS
- ☐ BFS
- ☐ IDFS
- ☐ A*

Enter the initial state of the 8-puzzle



1	2	8
3	0	4
5	6	7

Solve

Choose Algorithm:

- ☐ DFS
- ☐ BFS
- ☐ IDFS
- ☐ A*

Choose Distance Method:

- ☐ Manhattan
- ☐ Euclidean

Enter the initial state of the 8-puzzle

