

Artificial Intelligence Assignment 1

Report

Using Informed and Uninformed Search Algorithms to
Solve 8-Puzzle

Name: Ahmed Gamal Mahmoud
ID: 18010083

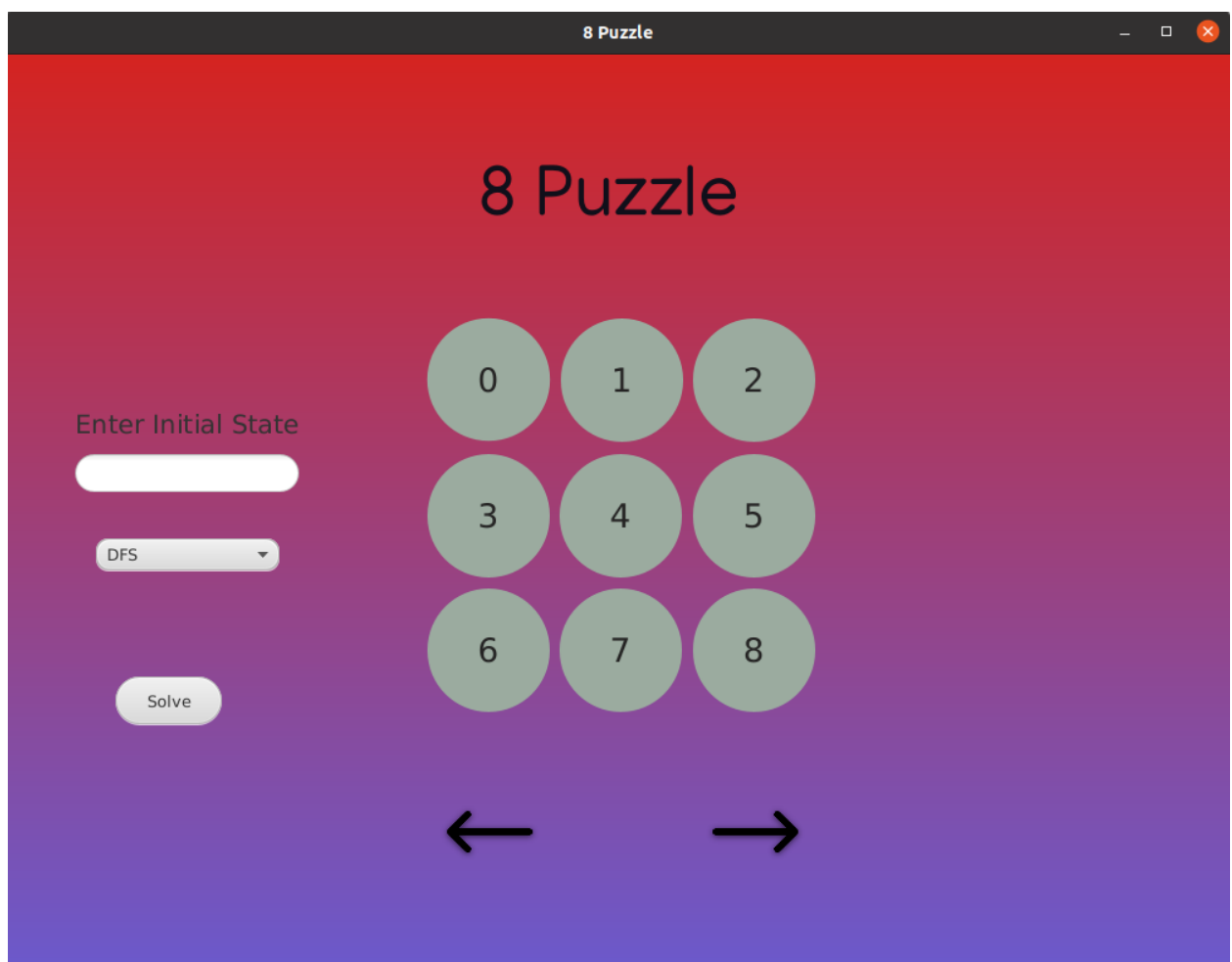
Name: Marwan Mohamed Saad Abougabal
ID: 18011736

Name: Mohamed Mofreh Abd El-monem El-gazzar
ID: 18011626

Name: Youssef Hany Fathy Shamsia
ID: 18015025

Assumptions and Details

- Once the given initial state is solved, the user can click on the 'Next' button to see the next state in the solution and the 'Previous' button to see the previous state. Once the user reaches the goal state, the buttons are disabled.
- Assuming the starting algorithm will be DFS if the user didn't change it



Common Data Structures

Node: Added data structure that contains state stored as string, depth, reference to parent node, and cost of the state according to the formula: $f(\text{state}) = g(\text{state}) + h(\text{state})$; where $g(\text{state})$ is the cost of reaching that state starting from the initial state and $h(\text{state})$ is the value of the heuristic function towards the goal (Manhattan or Euclidean).

HashMap: Built-in Java HashMap used for the explored list as well as a secondary data structure for checking if the entry exists in the priority queue faster (Find entry in $O(1)$ instead of searching the frontier list (Stack or Queue or Priority Queue) for the required entry).

Arraylist: Built-in Java Arraylist used to store all possible next states from the current states (all child nodes from the current node) as well as the path to the goal.

BFS search

Data Structures

Queue: Built-in Queue in java to track frontier list.

HashSet: Built-in HashSet in java to store elements in frontier list to get it faster ($O(1)$ theoretically) instead of looping on Queue.

Algorithms

Boolean SearchTech(String initialState, String goalState):

Resets holding variables for instance of BFS class and start searching using BFS technique.

DFS search

Data Structures

Stack: Built-in Stack in java to track frontier list.

HashSet: Built-in HashSet in java to store elements in frontier list to get it faster ($O(1)$ theoretically) instead of looping on Stack.

Algorithms

Boolean SearchTech(String initialState, String goalState):

Resets holding variables for instance of DFS class and start searching using DFS technique.

A* search

Data Structures

Priority Queue: Built-in Java Priority Queue used for the frontier list to hold state in the frontier where the state with the minimum value of $f(\text{state})$ is chosen next to be explored

Algorithms

Boolean SearchTech(String initialState, String goalState):
Resets holding variables for instance of Astar class and calls SearchAStarManhattan(initialState, goalState).

Boolean SearchTechEuclidean(String initialState, String goalState): resets holding variables for instance of Astar class and calls SearchAStarEuclidean(initialState,goalState).

Boolean SearchAStarManhattan(String initialState, String goalState): Create node for initial state and add it to the frontier list with cost using Manhattan Heuristic.

While the frontier list is not empty

- Remove the top of frontier (minimum $f(\text{state})$) and place it in the explored list

- Update maximum depth reached if necessary

- If goal state is reached

- Show search statistics and path to goal then return

- true

- Generate arraylist of child states (next states from current state)

- For each child in the arraylist

- Set cost of node using Manhattan heuristic

- If state is not in frontier and not in explored

- Add to frontier

- Else if state is in frontier

- Compare costs of both instances of the state

- And if the newer instance (Node n) has less

cost than the instance currently in the frontier
replace the version in the frontier
Return false if goal was not found

Boolean SearchAStarEuclidean(String initialState, String goalState): Create node for initial state and add it to the frontier list with cost using Euclidean Heuristic.

While the frontier list is not empty

Remove the top of frontier (minimum $f(\text{state})$) and place it in the explored list

Update maximum depth reached if necessary

If goal state is reached

Show search statistics and path to goal then return

true

Generate arraylist of child states (next states from current state)

For each child in the arraylist

Set cost of node using Euclidean heuristic

If state is not in frontier and not in explored

Add to frontier

Else if state is in frontier

Compare costs of both instances of the state

And if the newer instance (Node n) has less

cost than the instance currently in the frontier

replace the version in the frontier

Return false if goal was not found

Helper

Data Structures

N/A

Algorithms

ArrayList<Node> Gen_States(Node currentNode):

Generate all valid child states from the current state. Find index of empty block (zero) and generate states where it moves left, right, up and down where these moves are legal moves.

int Find_EmptySpace(String state): Find the index of the empty block (zero) in the state string.

String writeState(String state, int emptyIndex, int indexToSwap, int newDepth): Write the string of the new state by swapping the empty block (zero) with the block that would swap with it.

double ManhattanDist(String state, String goalState):

Calculate value of heuristic of current state from the goal state using Manhattan Distance Heuristic.

double EuclideanDist(String state, String goalState):

Calculate value of heuristic of current state from the goal state using Euclidean Distance Heuristic.

Sample Runs

Example	Category	BFS	DFS	A*	
N/A	Heuristic	N/A	N/A	Manhattan	Euclidean
EXAMPLE ONE "125340678"	Max Depth	3	29	3	3
	Cost of path	3	29	3	3
	Nodes expanded	16	30	4	4
	Search depth	3	29	3	3
	Running time(μs)	4020	12142	5600	729
EXAMPLE TWO "182043765"	Max Depth	21	28259	21	21
	Cost of path	21	28259	21	21
	Nodes expanded	63973	29580	1481	2249
	Search depth	21	28259	21	21
	Running time(μs)	153855	88993	3495	6941
EXAMPLE THREE "012435687"	Max Depth	18	66126	18	18
	Cost of path	18	53034	18	18
	Nodes expanded	18747	130413	565	739
	Search depth	18	53034	18	18
	Running time(μs)	50271	246582	12374	21083

path to goal.txt is attached with report which contains path to goal for each example and for each search technique.

Manhattan Vs Euclidean

After reviewing the table shown above and comparing the Manhattan heuristic and the Euclidean heuristic, it appears that the Manhattan heuristic expands less nodes and has a shorter running time in general. This aligns with the hypothesis that Manhattan heuristic would be better as the heuristic is a better model of how moves are made in the 8-puzzle. In conclusion, Manhattan Distance is a more admissible heuristic than Euclidean Distance.

Sample Runs photos

Example : 125340678

BFS sequence:

8 Puzzle

Enter Initial State

125340678

BFS

Solve

0	1	2
3	4	5
6	7	8

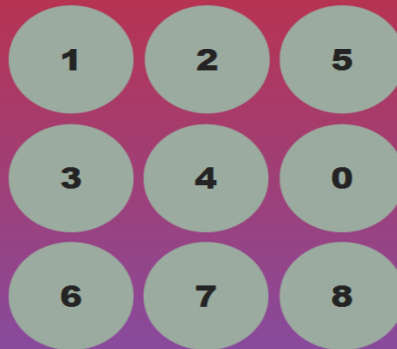
← →

8 Puzzle

Enter Initial State

BFS

Solve



Cost of Path = 3

Nodes Expanded = 16

Running time = 3769 micros

Max Depth = 3

Search Depth = 3

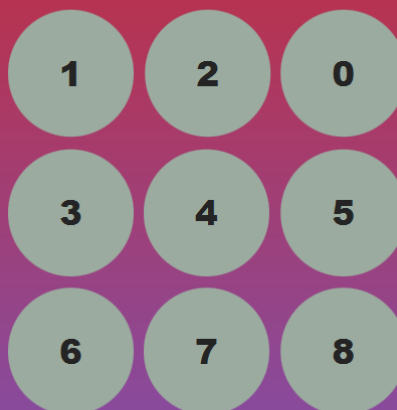


8 Puzzle

Enter Initial State

BFS

Solve



Cost of Path = 3

Nodes Expanded = 16

Running time = 3769 micros

Max Depth = 3

Search Depth = 3

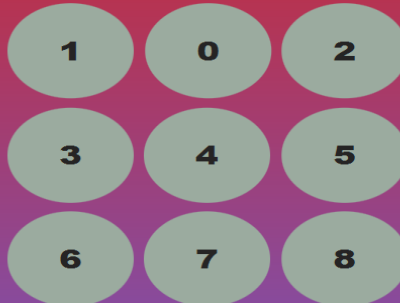


8 Puzzle

Enter Initial State

BFS

Solve



Cost of Path = 3

Nodes Expanded = 16

Running time = 3769 micros

Max Depth = 3

Search Depth = 3

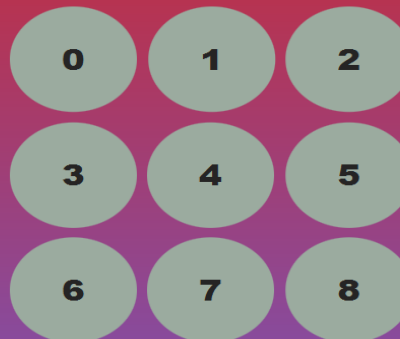


8 Puzzle

Enter Initial State

BFS

Solve



Cost of Path = 3

Nodes Expanded = 16

Running time = 3769 micros

Max Depth = 3

Search Depth = 3



SUCCESS

DFS start

8 Puzzle

Enter Initial State

DFS

Solve

1 2 5

3 4 0

6 7 8

← →

Cost of Path = 29

Nodes Expanded = 30

Running time = 3652 micros

Max Depth = 29

Search Depth = 29

A* Euclidean start

8 Puzzle

Enter Initial State

A* Euclidean

Solve

1 2 5

3 4 0

6 7 8

← →

Cost of Path = 3

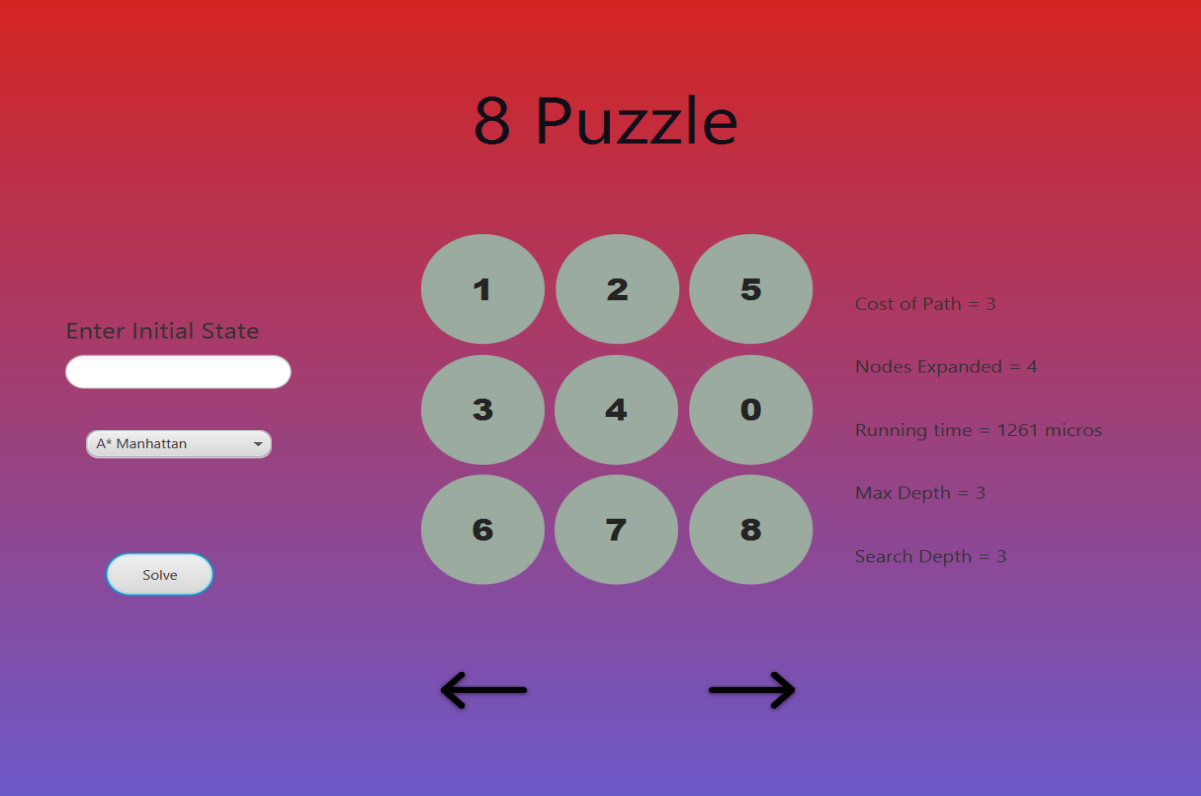
Nodes Expanded = 4

Running time = 4657 micros

Max Depth = 3

Search Depth = 3

A* Manhattan start



The image shows a web-based 8 Puzzle solver interface. At the top, the title "8 Puzzle" is displayed in a large, dark font. Below the title, the puzzle state is represented by a 3x3 grid of light green circles containing numbers. The numbers are: Row 1: 1, 2, 5; Row 2: 3, 4, 0; Row 3: 6, 7, 8. To the left of the grid, there is a section for entering the initial state, including a text input field, a dropdown menu set to "A* Manhattan", and a "Solve" button. To the right of the grid, several statistics are listed: "Cost of Path = 3", "Nodes Expanded = 4", "Running time = 1261 micros", "Max Depth = 3", and "Search Depth = 3". At the bottom of the grid, there are two large black arrows pointing left and right, indicating the possible moves for the puzzle.

8 Puzzle

Enter Initial State

A* Manhattan

Solve

Cost of Path = 3

Nodes Expanded = 4

Running time = 1261 micros

Max Depth = 3

Search Depth = 3

← →

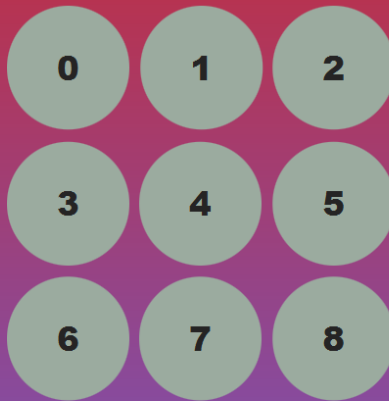
EXAMPLE : 432650781 **Unsolvable**

8 Puzzle

Enter Initial State

DFS

Solve

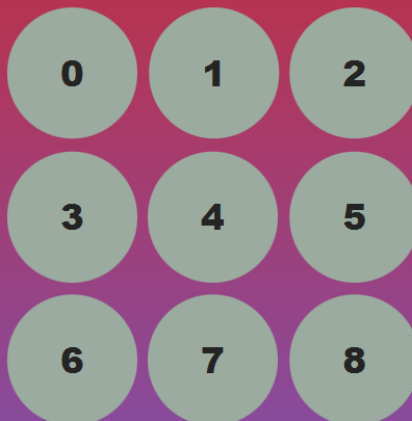


8 Puzzle

Enter Initial State

DFS

Solve



Unsolvable

Example : 01234 **Invalid**

