Informed and Uninformed Search

8-puzzle Solution

Submitted to

Dr. Amira Youssef

Eng. Heba Abd El-Atty

Submitted By

Amira Hossam	5312
Mariam Medhat	5351
Basma Hesham	5432
Hana Magdy	5455
Toka Sherif	5492
Nada Elshafey	5612

Table of Contents

1.Problem Formulation ... **3**

Introduction ... 3

Algorithm Review ... 3

2. 8-puzzle using DFS search ... 4

DFS search Algorithm ... 4

DFS search pseudo-code ... 5

3. 8-puzzle using A*search ... **6**

Introduction ... 6

A* search Algorithm ... 7

Solving 8 puzzle using A* ... 8

A* search pseudo-code ... 9

4. Heuristic function ... **9**

5. Manhattan Distance ... **10**

Introduction ... 10

Pseudo-code ... 10

Example ... 10

6. Euclidean Distance ... **11**

Introduction ... 11

Pseudo-code ... 11

Example ... 11

7. Program Structure ... **12**

Source code ... 12

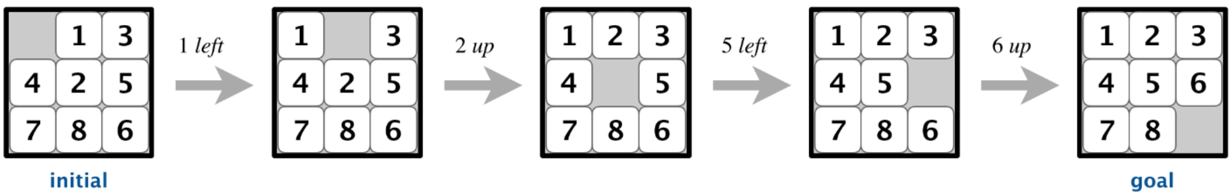
Sample Run ... 15

8. Conclusion ... **32**

I. Problem Formulation

➢ Introduction

8-puzzle It is 3x3 matrix with 8 square blocks containing 1 to 8 and a blank square. The main idea of 8 puzzle is to reorder these of the squares adjacent to blank block can squares into a numerical order of 1 to 8 and last square as blank. Each move up, down, left or right depending on the edges of the matrix.



➢ Generic Algorithm review

It begin by visiting the root node of the search tree, given by the initial state. Three major things happen in sequence in order to visit a node:

1. we remove a node from the frontier set.
2. we check the state against the goal state to determine if a solution has been found.
3. if the result of the check is false, we then expand the node. To expand a node, we generate successor nodes adjacent to the current node, and add them to the frontier set. If these successor nodes are already in the frontier set, or have already been visited, then they should not be added to the frontier again.

   This describes the life-cycle of a visit, and is the basic order of operations for search agents (1) remove, (2) check, and (3) expand.

II. DFS search

➢ <u>**DFS Search Algorithm**</u>.

DFS is an uninformed search algorithm. perform **<u>a depth-first search.</u>**
A List which is used as a (LIFO) Stack.

- Create an empty list (explored),this list will states that have been visited
- Create another list(frontier) that contains just the starting state, this list contains states that have not been visited
- Create an empty map contains previous state of each state has been visited
- While the 2nd-list not empty

I. Remove it from frontier list by using List.pop() and put it in explored-list
    - If the state is goal
        - Return path from goal state to initial state (recursively from constructed map)
    - If the state isn't goal
        - If next state has been visited before(check from explored)
            - ignore it
        - If next state isn't in visited list(explored list)
            - Put it in explored list

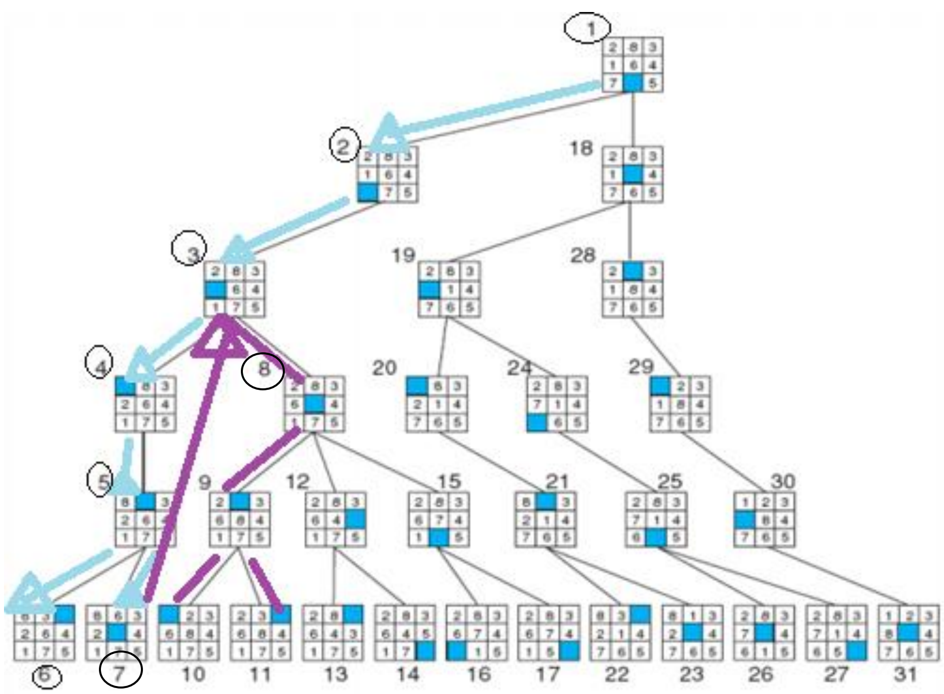
II. Repeat the loop

III. If the non-visited list(frontier-list) is empty and never found the goal, No solution possible

➢ **DFS Search PseudoCode**.

```
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
```

- DFS Searches branch by branch with each branch pursued to maximum depth.



III. A* search

➢ Introduction

A* is recursive algorithm that calls itself until a solution is found. In this algorithm we consider two heuristic functions, euclidean distance heuristic and manhattan distance heuristic. In this part we have implemented the state space generation using both the heuristics.

A* is an **informed search algorithm**. It is a combination of uniform cost search and best first search, which avoids expanding expensive paths.

Calculating g(n) which is a measure of step cost for each move made from current state to next state, initially it is set to zero. For each of the heuristic we have implemented f(n)= g(n)+h(n), where h(n) is the heuristic function used.

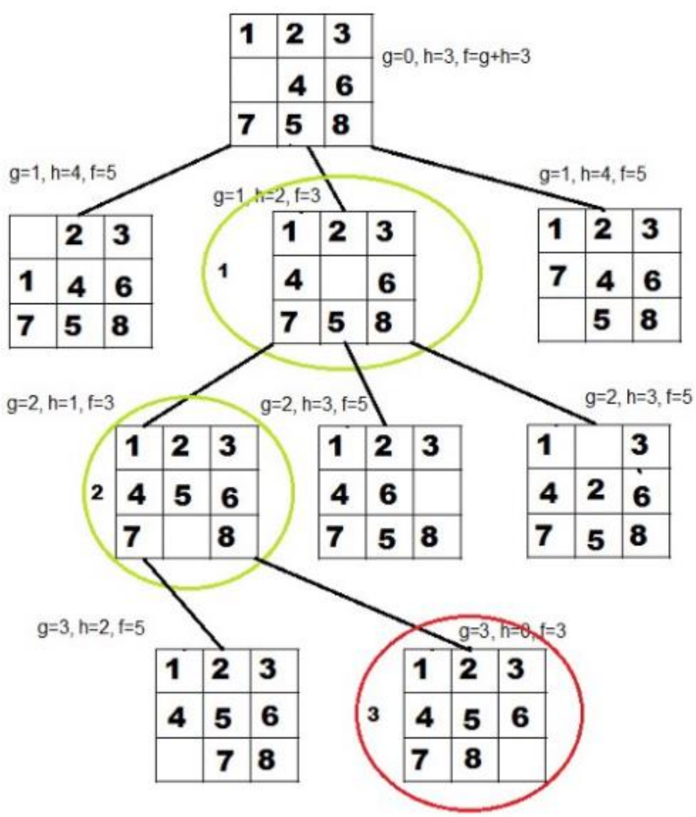
In this part, we have used the optimality of A* by not allowing any node generation of previously traversed nodes.Used two sets they are closed set and open set to implement this functionality of A* algorithm. Closed set stores all the previously expanded nodes and open set (priority queue) which stores all of the non-duplicate nodes and sorts them according to f(n) value.

➢ A* Algorithm

- A* algorithm keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time.
- A min heap which will take a node (cost, current state) as input, which will heapify it according to the cost calculated depending on the type of the Heuristic algorithm.
- It also has a list that holds all the nodes that are not visited to be explored and it chooses the most optimal node from this list, thus saving time not exploring less optimal nodes.
- we use two lists ‘frontier list‘ and ‘explored list‘ ,the frontier list contains all the nodes that are being generated and are not existing in the explored list and each node explored after it’s neighboring nodes are discovered is put in the explored list and the neighbors are put in the frontier list this is how the nodes expand
- Each node has a pointer to its parent to retrace the path to the parent
- Initially, the frontier list holds the Initial node. The next node chosen from the frontier list is based on its **cost**, the node with the least cost is picked up and explored.
- **F(n) = h(n) + g(n)** , we can calculate the **h(n)** by comparing the current state and goal state.

➢ **How A* solves the 8-Puzzle problem.**

- Move the empty space in all the possible directions in the start state and calculate the **f(n)** for each state (Expanding the current state).
- After expanding the current state, it is pushed into the **explored list** and the newly generated states are pushed into the **frontier list**.
- A state with the least f(n) is selected and expanded again.
- Loop this process until the goal state occurs as the current state.



➤ **A* Search PseudoCode**.

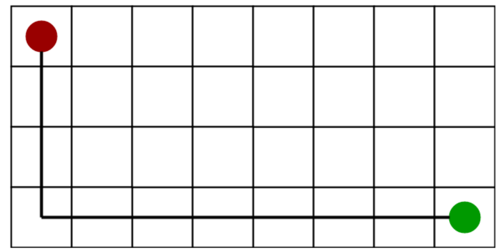
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    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
```

IV. Heuristic Functions

The heuristic function is a way to inform the search regarding the direction to a goal , It provides an information to estimate which neighboring node will lead to the goal.

V. <u>Manhattan Distance</u>

The Manhattan distance heuristic is used for its ability to estimate the number of moves required to bring a given puzzle state to the solution state.



Manhattan distance heuristic function measures the least steps needed for each of the tiles in the 8-puzzle initial or current state to arrive to the goal state position

It's the sum of absolute values of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively, i.e. **|i - l|+ |j - m|**

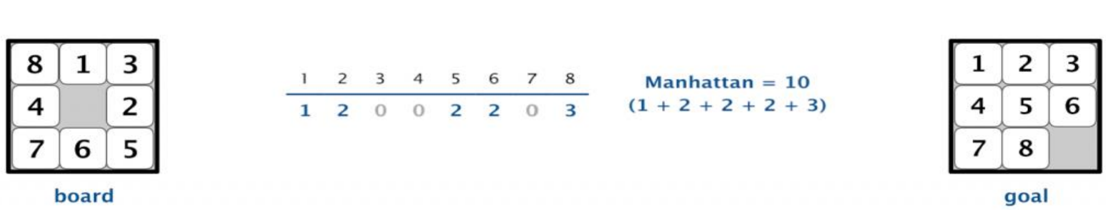
➢ <u>pseudo code</u>

```
function manhattan distance(node, goal) =

dx = abs(node.x - goal.x)

dy = abs(node.y - goal.y)

return dx + dy
```

➢ <u>Example</u>

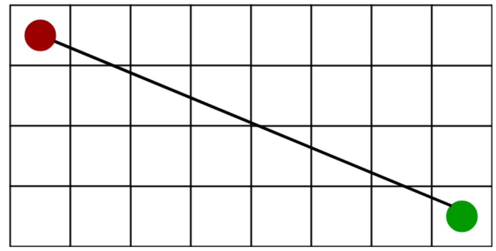
- In this heuristic, we are allowed to move only in four directions only (right, left, top, bottom)

we find the heuristic value required to reach the final state from initial state. The cost function, g(n) = 0 (initial state)
The value is obtained, as "1" in the current state is 1 horizontal distance away than the "1" final state. Same goes for all no .
So total value for h(n) is 1+2+0+0+2+2+0+3=10. Total cost function f(n) is equal to 10+ 0 = 10



VI. Euclidean Distance

Euclidean distance heuristic use this heuristic when we are allowed to move in any directions



It is the sqrt (sum of values of squaring of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively,) i.e. $\sqrt{(x_l - x_i)^2 + (y_m - y_j)^2}$

Pseudo code

```
function eclidean distance(node, goal) =

dx = (node.x - goal.x)

dy = (node.y - goal.y)

return sqrt(dx + dy)
```

Example

- In this heuristic, we are allowed to move in any directions
  we find the heuristic value required to reach the final state from initial state. The cost function, g(n) = 0 (initial state)
  The value is obtained, as "1" in the current state is 1 horizontal distance away than the "1" final state. Same goes for all no . , 2 can move in diagonal
  So total value for h(n) is 1+1+0+0+1+1+0+1=5. Total cost function f(n) is equal to 5+ 0 = 5.

```
8  1  3        1  2  3      1  2  3  4  5  6  7  8
4     2        4  5  6      ----------------------
7  6  5        7  8         1  1  0  0  1  1  0  1

initial          goal          Euclidean = 5 + 0
```

VII. Program Structure

➢ Source Code

- **DFS**

```python
def dfs(initialState):  # dfs search
    frontier = [initialState]  # first put the initial states in a forntier list
    explored = []   # create an empty explored list
    maxDepth = 0
    if_frontier = 0

    while len(frontier) > 0:  # pass over all the elements in the frontier list
        state = frontier.pop()
        if not state:
            continue

        depth = frontier.count(None)
        maxDepth = max(maxDepth, depth)
        explored.append(state.matrix_to_list())
        if check_if_goal(state):
            return reachGoal(state, len(explored), depth)

        depthLimit = 50  # max depth
        if depth < depthLimit:
            row = len(frontier)
            for neighbour in state.children():  # loop over all the neighbours of the state
                if not check_if_explored(neighbour, explored):  # check if the neighbour in the explored list
                    if not check_if_explored(neighbour, frontier):  # check if the neighbour in the frontier list
                        if_frontier = 1 # if the neighbour in the frontier list append it
                        frontier.append(neighbour)

            if if_frontier == 1:    # if frontier flag = 1 insert the row in the frontier list and toggle the flag
                frontier.insert(row, None)
                if_frontier = 0

    return False
```

- **A***

```python
def astar(initialState, type):  # A* search
    frontier = []   # create an empty frontier list
    explored = []   # create an empty explored list

    if type == "manhattan":
        cost = calculate_manhattan(initialState)
    else:
        cost = calculate_eculedian(initialState)

    node = (cost, initialState)  # create new node
    heapq.heappush(frontier, node)  # push the node to the frontier list
    while len(frontier) > 0:    # loop over all the nodes in the frontier list and explore them
        state = heapq.heappop(frontier)[1]
        explored.append(state.matrix_to_list())

        if check_if_goal(state):    # stop when reach the goal
            depth = 0
            temp = state
            while temp: # calculate the depth of the node
                depth += 1
                temp = temp.parent
            return reachGoal(state, len(explored), depth)

        for neighbour in state.children():  # check all the neighbours of the node
            if not check_if_explored(neighbour, explored):  # check if the node is not explored
                # if the node is not the frontier list calculate its cost and add it to the frontier list
                if not check_if_frontier(neighbour, [state for cost, state in frontier]):
                    if type == "manhattan":
                        cost = calculate_manhattan(neighbour)
                    else:
                        cost = calculate_eculedian(neighbour)
                    node = (cost, neighbour)
                    heapq.heappush(frontier, node)
    return False
```

- **Manhattan**

```python
    return False

# function to calculate the cost of the manhattan method
def calculate_manhattan(state):
    tiles = state.matrix_to_list()
    h = state.cost
    for i in range(len(tiles)):
        if tiles[i] != 0:
            currentCellX, currentCellY, goalCellX, goalCellY = get_dim(
                i, tiles[i], state.size)
            h += (abs(currentCellX - goalCellX) + abs(currentCellY - goalCellY))
    return h
```

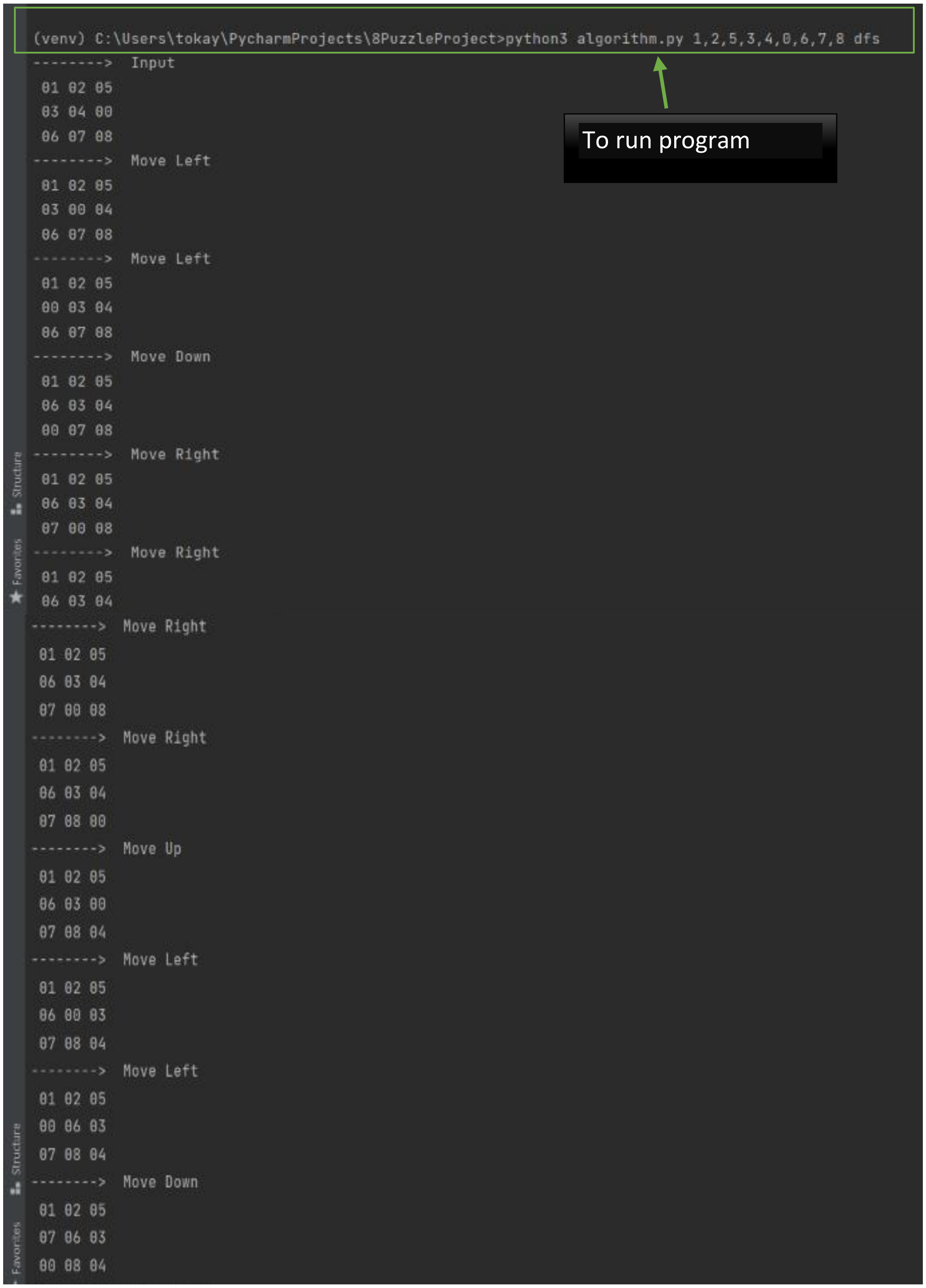
- **Euclidean**

```python
# function to calculate the cost of the eculedian method
def calculate_eculedian(state):
    tiles = state.matrix_to_list()
    h = state.cost
    for i in range(len(tiles)):
        if tiles[i] != 0:
            currentCellX, currentCellY, goalCellX, goalCellY = get_dim(
                i, tiles[i], state.size)
            h += ((currentCellX - goalCellX) ** 2 + (currentCellY - goalCellY) ** 2) ** 0.5
    return h
```

➢ Sample Run

❖ TestCase:1

- **DFS**



```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 1,2,5,3,4,0,6,7,8 dfs
-------->  Input
 01 02 05
 03 04 00
 06 07 08
-------->  Move Left
 01 02 05
 03 00 04
 06 07 08
-------->  Move Left
 01 02 05
 00 03 04
 06 07 08
-------->  Move Down
 01 02 05
 06 03 04
 00 07 08
-------->  Move Right
 01 02 05
 06 03 04
 07 00 08
-------->  Move Right
 01 02 05
 06 03 04
-------->  Move Right
 01 02 05
 06 03 04
 07 00 08
-------->  Move Right
 01 02 05
 06 03 04
 07 08 00
-------->  Move Up
 01 02 05
 06 03 00
 07 08 04
-------->  Move Left
 01 02 05
 06 00 03
 07 08 04
-------->  Move Left
 01 02 05
 00 06 03
 07 08 04
-------->  Move Down
 01 02 05
 07 06 03
 00 08 04
```

```
--------> Move Right
 01 02 05
 07 06 03
 08 00 04
--------> Move Right
 01 02 05
 07 06 03
 08 04 00
--------> Move Up
 01 02 05
 07 06 00
 08 04 03
--------> Move Left
 01 02 05
 07 00 06
 08 04 03
--------> Move Left
 01 02 05
 00 07 06
 08 04 03
--------> Move Down
 01 02 05
 08 07 06
 00 04 03
--------> Move Right
 01 02 05
 08 07 06
 04 00 03
--------> Move Right
 01 02 05
 08 07 06
 04 03 00
--------> Move Up
 01 02 05
 08 07 00
 04 03 06
--------> Move Left
 01 02 05
 08 00 07
 04 03 06
--------> Move Left
 01 02 05
 00 08 07
 04 03 06
--------> Move Down
 01 02 05
 04 08 07
 00 03 06
```

```
-------->  Move Right
 01 02 05
 04 08 07
 03 00 06
-------->  Move Right
 01 02 05
 04 08 07
 03 06 00
-------->  Move Up
 01 02 05
 04 08 00
 03 06 07
-------->  Move Left
 01 02 05
 04 00 08
 03 06 07
-------->  Move Left
 01 02 05
 00 04 08
 03 06 07
-------->  Move Down
 01 02 05
 03 04 08
 00 06 07
-------->  Move Right
 01 02 05
 03 04 08
 06 00 07
-------->  Move Up
 01 02 05
 03 00 08
 06 04 07
-------->  Move Up
 01 00 05
 03 02 08
 06 04 07
-------->  Move Left
 00 01 05
 03 02 08
 06 04 07
-------->  Move Down
 03 01 05
 00 02 08
 06 04 07
-------->  Move Right
 03 01 05
 02 00 08
 06 04 07
```

```
-------->  Move Down
 03 01 05
 00 02 08
 06 04 07
-------->  Move Right
 03 01 05
 02 00 08
 06 04 07
-------->  Move Down
 03 01 05
 02 04 08
 06 00 07
-------->  Move Right
 03 01 05
 02 04 08
 06 07 00
-------->  Move Up
 03 01 05
 02 04 00
 06 07 08
-------->  Move Left
 03 01 05
 02 00 04
 06 07 08
```

```
-------->  Move Left
 03 01 05
 02 00 04
 06 07 08
-------->  Move Left
 03 01 05
 00 02 04
 06 07 08
-------->  Move Up
 00 01 05
 03 02 04
 06 07 08
-------->  Move Right
 01 00 05
 03 02 04
 06 07 08
-------->  Move Right
 01 05 00
 03 02 04
 06 07 08
-------->  Move Down
 01 05 04
 03 02 00
 06 07 08
```

```
--------->  Move Left
 01 05 04
 03 00 02
 06 07 08
--------->  Move Up
 01 00 04
 03 05 02
 06 07 08
--------->  Move Right
 01 04 00
 03 05 02
 06 07 08
--------->  Move Down
 01 04 02
 03 05 00
 06 07 08
--------->  Move Left
 01 04 02
 03 00 05
 06 07 08
--------->  Move Up
 01 00 02
 03 04 05
 06 07 08
--------->  Move Left
 01 04 02
 03 00 05
 06 07 08
--------->  Move Up
 01 00 02
 03 04 05
 06 07 08
--------->   Move Left
 00 01 02
 03 04 05
 06 07 08
**** DONE ****
```

```
Path:   -> Input -> Move Left -> Move Left -> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -
> Move Left -> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -> Mov
e Right -> Move Up -> Move Up -> Move Left -> Move Down -> Move Right -> Move Down -> Move Right -> Move Up -> Move Left -> Move Left -> Move Up -> Move Right -> Move Right -> Move Down -
> Move Left -> Move Up -> Move Right -> Move Down -> Move Left -> Move Up -> Move Left
```

```
Cost:  49
Nodes Expanded:  10765
Depth:  49
Running Time: 38.7660 sec
```

- **A* Manhattan**

```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 1,2,5,3,4,0,6,7,8 a*
manhattan / euclidean ? manhattan
-------->  Input
 01 02 05
 03 04 00
 06 07 08
-------->  Move Up
 01 02 00
 03 04 05
 06 07 08
-------->  Move Left
 01 00 02
 03 04 05
 06 07 08
-------->   Move Left
 00 01 02
 03 04 05
 06 07 08
**** DONE ****

Path:   -> Input -> Move Up -> Move Left -> Move Left
Cost:  3
Nodes Expanded:  4
Depth:  4
Running Time: 6.5198 sec
```

To run program

- **A* Euclidean**

```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 1,2,5,3,4,0,6,7,8 a*
manhattan / euclidean ? euclidean
-------->  Input
 01 02 05
 03 04 00
 06 07 08
-------->  Move Up
 01 02 00
 03 04 05
 06 07 08
-------->  Move Left
 01 00 02
 03 04 05
 06 07 08
-------->   Move Left
 00 01 02
 03 04 05
 06 07 08
**** DONE ****

Path:   -> Input -> Move Up -> Move Left -> Move Left
Cost:  3
Nodes Expanded:  4
Depth:  4
Running Time: 4.6071 sec
```

To run program

❖ <u>TestCase:2</u>

- **<u>DFS</u>**

```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 3,1,2,0,4,5,6,7,8 dfs
-------->  Input
 03 01 02
 00 04 05
 06 07 08
-------->  Move Right
 03 01 02
 04 00 05
 06 07 08
-------->  Move Right
 03 01 02
 04 05 00
 06 07 08
-------->  Move Down
 03 01 02
 04 05 08
 06 07 00
-------->  Move Left
 03 01 02
 04 05 08
 06 00 07
-------->  Move Left
 03 01 02
 04 05 08
 00 06 07
-------->  Move Up
 03 01 02
 00 05 08
 04 06 07
-------->  Move Right
 03 01 02
 05 00 08
 04 06 07
-------->  Move Right
 03 01 02
 05 08 00
 04 06 07
-------->  Move Down
 03 01 02
 05 08 07
 04 06 00
-------->  Move Left
 03 01 02
 05 08 07
 04 00 06
-------->  Move Left
 03 01 02
 05 08 07
 00 04 06
-------->  Move Up
 03 01 02
 00 08 07
 05 04 06
-------->  Move Right
 03 01 02
 08 00 07
 05 04 06
-------->  Move Right
 03 01 02
 08 07 00
 05 04 06
-------->  Move Down
 03 01 02
 08 07 06
 05 04 00
-------->  Move Left
 03 01 02
 08 07 06
 05 00 04
-------->  Move Left
 03 01 02
 08 07 06
 00 05 04
```

To run program

```
-------->  Move Up
 03 01 02
 00 07 06
 08 05 04
-------->  Move Right
 03 01 02
 07 00 06
 08 05 04
-------->  Move Right
 03 01 02
 07 06 00
 08 05 04
-------->  Move Down
 03 01 02
 07 06 04
 08 05 00
-------->  Move Left
 03 01 02
 07 06 04
 08 00 05
-------->  Move Left
 03 01 02
 07 06 04
 00 08 05
-------->  Move Up
 03 01 02
 00 06 04
 07 08 05
-------->  Move Right
 03 01 02
 06 00 04
 07 08 05
-------->  Move Right
 03 01 02
 06 04 00
 07 08 05
-------->  Move Down
 03 01 02
 06 04 05
 07 08 00
-------->  Move Left
 03 01 02
 06 04 05
 07 00 08
-------->  Move Up
 03 01 02
 06 00 05
 07 04 08
-------->  Move Up
 03 01 02
 06 00 05
 07 04 08
-------->  Move Up
 03 00 02
 06 01 05
 07 04 08
-------->  Move Right
 03 02 00
 06 01 05
 07 04 08
-------->  Move Down
 03 02 05
 06 01 00
 07 04 08
-------->  Move Left
 03 02 05
 06 00 01
 07 04 08
-------->  Move Down
 03 02 05
 06 04 01
 07 00 08
```

```
-------->  Move Left
 03 02 05
 06 04 01
 00 07 08
-------->  Move Up
 03 02 05
 00 04 01
 06 07 08
-------->  Move Right
 03 02 05
 04 00 01
 06 07 08
-------->  Move Right
 03 02 05
 04 01 00
 06 07 08
-------->  Move Up
 03 02 00
 04 01 05
 06 07 08
-------->  Move Left
 03 00 02
 04 01 05
 06 07 08
-------->  Move Left
 00 03 02
 04 01 05
 06 07 08
-------->  Move Down
 04 03 02
 00 01 05
 06 07 08
-------->  Move Right
 04 03 02
 01 00 05
 06 07 08
-------->  Move Up
 04 00 02
 01 03 05
 06 07 08
-------->  Move Left
 00 04 02
 01 03 05
 06 07 08
-------->  Move Down
 01 04 02
 00 03 05
 06 07 08
-------->  Move Down
 01 04 02
 00 03 05
 06 07 08
-------->  Move Right
 01 04 02
 03 00 05
 06 07 08
-------->  Move Up
 01 00 02
 03 04 05
 06 07 08
-------->  Move Left
 00 01 02
 03 04 05
 06 07 08
**** DONE ****
```

```
Path:   -> Input -> Move Right -> Move Right -> Move Down -> Move Left -> Move Left -> Move Up -> Move Right -> Move Right -> Move Down -> Move Left -> Move Left -> Move Up -> Move Right
-> Move Right -> Move Down -> Move Left -> Move Left -> Move Up -> Move Right -> Move Right -> Move Down -> Move Left -> Move Left -> Move Up -> Move Right -> Move Right -> Move Down -> M
ove Left -> Move Up -> Move Up -> Move Right -> Move Down -> Move Left -> Move Down -> Move Left -> Move Up -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -
> Move Right -> Move Up -> Move Left -> Move Down -> Move Right -> Move Up -> Move Left
```

```
Cost:  49
Nodes Expanded:  9925
Depth:  49
Running Time: 43.1230 sec
```

- **A* Manhattan**

```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 3,1,2,0,4,5,6,7,8 a*
manhattan / euclidean ? manhattan
-------->  Input
 03 01 02
 00 04 05
 06 07 08
-------->   Move Up
 00 01 02
 03 04 05
 06 07 08
**** DONE ****

Path:   -> Input -> Move Up
Cost:  1
Nodes Expanded:  2
Depth:  2
Running Time: 5.8353 sec
```

To run program

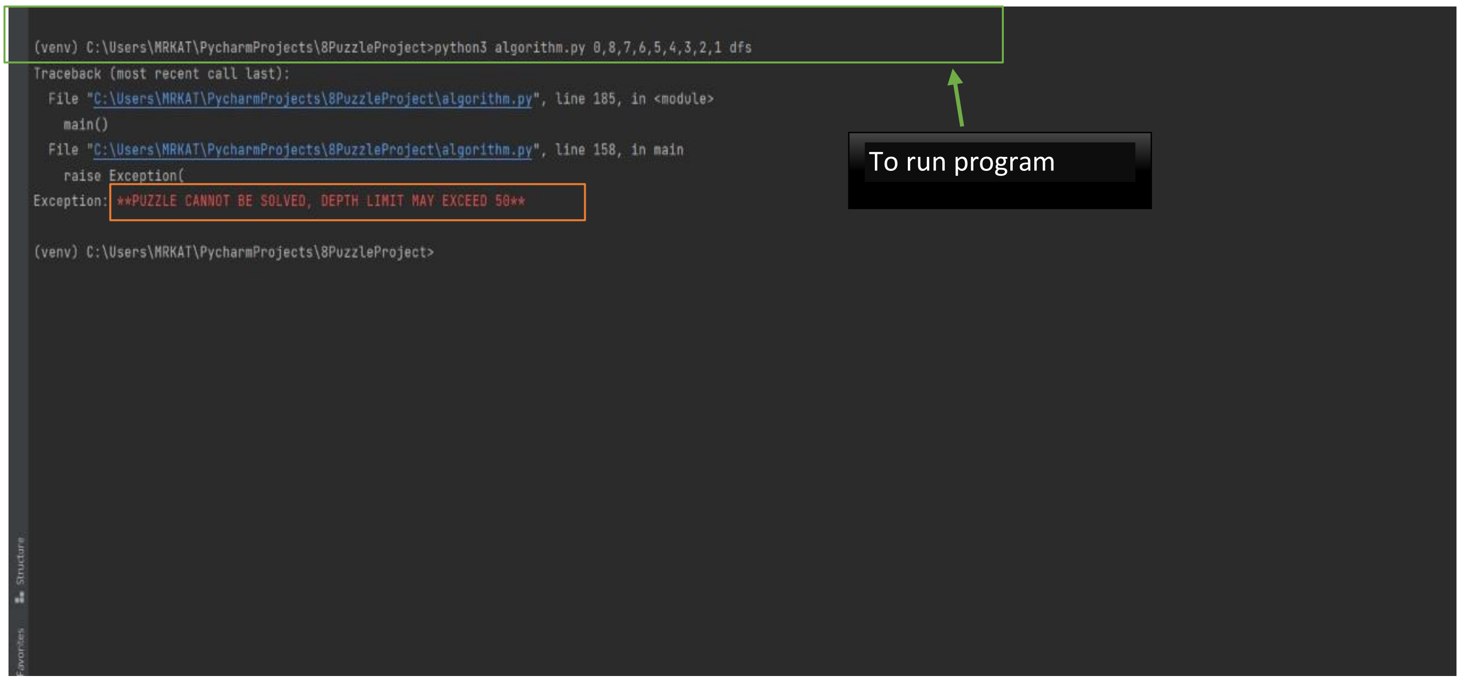
- **A* Euclidean**

```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 3,1,2,0,4,5,6,7,8 a*
manhattan / euclidean ? euclidean
-------->  Input
 03 01 02
 00 04 05
 06 07 08
-------->   Move Up
 00 01 02
 03 04 05
 06 07 08
**** DONE ****

Path:   -> Input -> Move Up
Cost:  1
Nodes Expanded:  2
Depth:  2
Running Time: 7.7401 sec
```

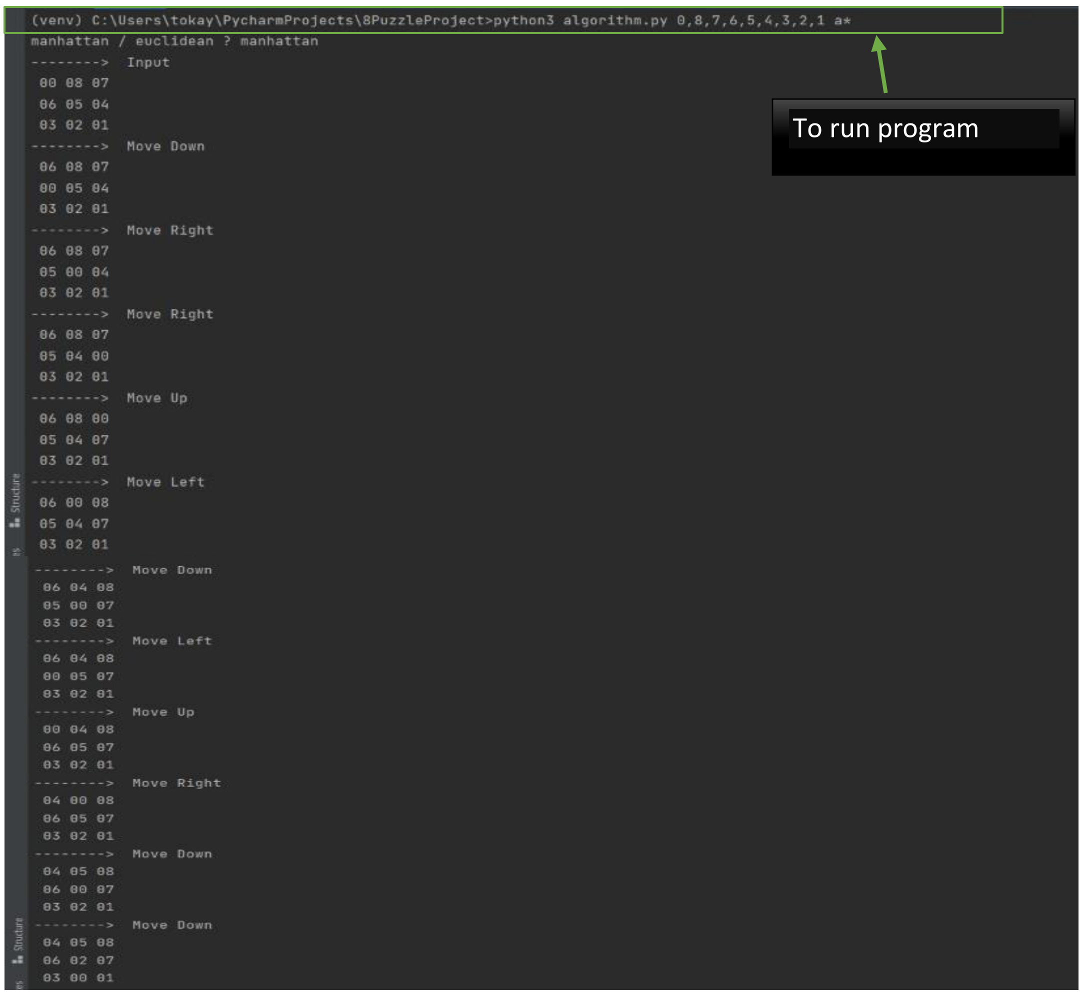
To run program

❖ TestCase:3

- **DFS**



- **A* Manhattan**



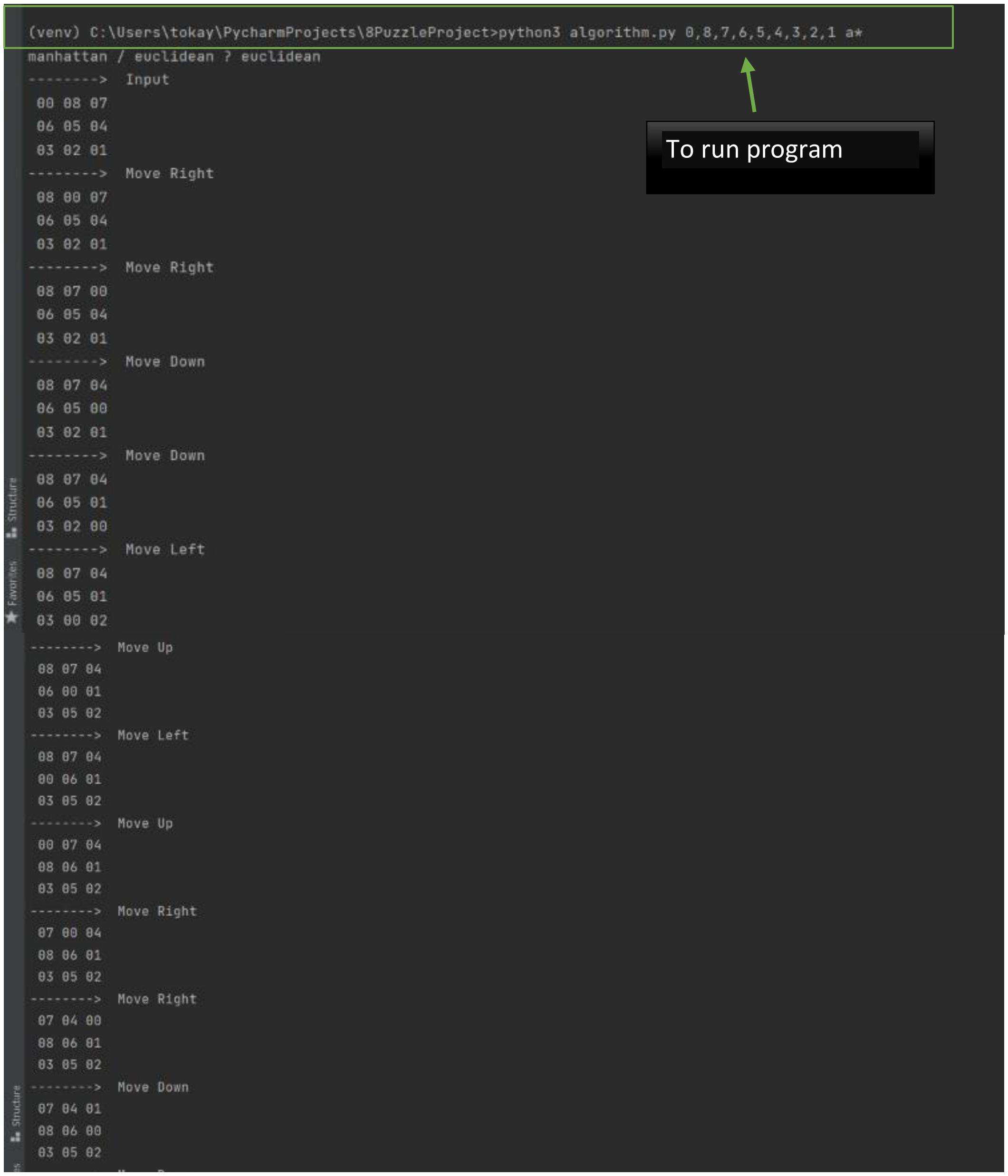
```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 0,8,7,6,5,4,3,2,1 a*
manhattan / euclidean ? manhattan
-------->  Input
 00 08 07
 06 05 04
 03 02 01
-------->  Move Down
 06 08 07
 00 05 04
 03 02 01
-------->  Move Right
 06 08 07
 05 00 04
 03 02 01
-------->  Move Right
 06 08 07
 05 04 00
 03 02 01
-------->  Move Up
 06 08 00
 05 04 07
 03 02 01
-------->  Move Left
 06 00 08
 05 04 07
 03 02 01
-------->  Move Down
 06 04 08
 05 00 07
 03 02 01
-------->  Move Left
 06 04 08
 00 05 07
 03 02 01
-------->  Move Up
 00 04 08
 06 05 07
 03 02 01
-------->  Move Right
 04 00 08
 06 05 07
 03 02 01
-------->  Move Down
 04 05 08
 06 00 07
 03 02 01
-------->  Move Down
 04 05 08
 06 02 07
 03 00 01
```

```
-------->  Move Right
 04 05 08
 06 02 07
 03 01 00
-------->  Move Up
 04 05 08
 06 02 00
 03 01 07
-------->  Move Up
 04 05 00
 06 02 08
 03 01 07
-------->  Move Left
 04 00 05
 06 02 08
 03 01 07
-------->  Move Down
 04 02 05
 06 00 08
 03 01 07
-------->  Move Down
 04 02 05
 06 01 08
 03 00 07
-------->  Move Left
 04 00 05
 06 02 08
 03 01 07
-------->  Move Down
 04 02 05
 06 00 08
 03 01 07
-------->  Move Down
 04 02 05
 06 01 08
 03 00 07
-------->  Move Left
 04 02 05
 06 01 08
 00 03 07
-------->  Move Up
 04 02 05
 00 01 08
 06 03 07
-------->  Move Right
 04 02 05
 01 00 08
 06 03 07
```

```
--------> Move Right
 04 02 05
 01 00 08
 06 03 07
--------> Move Down
 04 02 05
 01 03 08
 06 00 07
--------> Move Right
 04 02 05
 01 03 08
 06 07 00
--------> Move Up
 04 02 05
 01 03 00
 06 07 08
--------> Move Up
 04 02 00
 01 03 05
 06 07 08
--------> Move Left
 04 00 02
 01 03 05
 06 07 08
--------> Move Left
 04 00 02
 01 03 05
 06 07 08
--------> Move Left
 00 04 02
 01 03 05
 06 07 08
--------> Move Down
 01 04 02
 00 03 05
 06 07 08
--------> Move Right
 01 04 02
 03 00 05
 06 07 08
--------> Move Up
 01 00 02
 03 04 05
 06 07 08
-------->  Move Left
 00 01 02
 03 04 05
 06 07 08
**** DONE ****
```

```
Path:  -> Input -> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -> Move Down -> Move Left -> Move Up -> Move Right -> Move Down -> Move Down -> Move Right -> Move Up ->
Move Up -> Move Left -> Move Down -> Move Down -> Move Left -> Move Up -> Move Right -> Move Down -> Move Right -> Move Up -> Move Up -> Move Left -> Move Left -> Move Down -> Move Right
-> Move Up -> Move Left
Cost:  30
Nodes Expanded:  18441
Depth:  31
Running Time: 366.2906 sec
```

- **A* Euclidean**



```
(venv) C:\Users\tokay\PycharmProjects\8PuzzleProject>python3 algorithm.py 0,8,7,6,5,4,3,2,1 a*
manhattan / euclidean ? euclidean
-------->  Input
 00 08 07
 06 05 04
 03 02 01
-------->  Move Right
 08 00 07
 06 05 04
 03 02 01
-------->  Move Right
 08 07 00
 06 05 04
 03 02 01
-------->  Move Down
 08 07 04
 06 05 00
 03 02 01
-------->  Move Down
 08 07 04
 06 05 01
 03 02 00
-------->  Move Left
 08 07 04
 06 05 01
 03 00 02
-------->  Move Up
 08 07 04
 06 00 01
 03 05 02
-------->  Move Left
 08 07 04
 00 06 01
 03 05 02
-------->  Move Up
 00 07 04
 08 06 01
 03 05 02
-------->  Move Right
 07 00 04
 08 06 01
 03 05 02
-------->  Move Right
 07 04 00
 08 06 01
 03 05 02
-------->  Move Down
 07 04 01
 08 06 00
 03 05 02
```

```
-------->  Move Down
 07 04 01
 08 06 02
 03 05 00
-------->  Move Left
 07 04 01
 08 06 02
 03 00 05
-------->  Move Up
 07 04 01
 08 00 02
 03 06 05
-------->  Move Left
 07 04 01
 00 08 02
 03 06 05
-------->  Move Down
 07 04 01
 03 08 02
 00 06 05
-------->  Move Right
 07 04 01
 03 08 02
 06 00 05
-------->  Move Up
 07 04 01
 03 00 02
 06 08 05
-------->  Move Up
 07 00 01
 03 04 02
 06 08 05
-------->  Move Left
 00 07 01
 03 04 02
 06 08 05
-------->  Move Down
 03 07 01
 00 04 02
 06 08 05
-------->  Move Right
 03 07 01
 04 00 02
 06 08 05
-------->  Move Up
 03 00 01
 04 07 02
 06 08 05
```

```
--------> Move Right
 03 01 00
 04 07 02
 06 08 05
--------> Move Down
 03 01 02
 04 07 00
 06 08 05
--------> Move Down
 03 01 02
 04 07 05
 06 08 00
--------> Move Left
 03 01 02
 04 07 05
 06 00 08
--------> Move Up
 03 01 02
 04 00 05
 06 07 08
--------> Move Left
 03 01 02
 00 04 05
 06 07 08
--------> Move Up
 00 01 02
 03 04 05
 06 07 08
**** DONE ****
```

```
Path:  -> Input -> Move Right -> Move Right -> Move Down -> Move Down -> Move Left -> Move Up -> Move Left -> Move Up -> Move Right -> Move Right -> Move Down -> Move Down -> Move Left -
> Move Up -> Move Left -> Move Down -> Move Right -> Move Up -> Move Up -> Move Left -> Move Down -> Move Right -> Move Up -> Move Right -> Move Down -> Move Down -> Move Left -> Move Up
-> Move Left -> Move Up
Cost:  30
Nodes Expanded:  29464
Depth:  31
Running Time: 1975.5996 sec
```

VIII. Conclusion

Based on our result, in case of Euclidian’s algorithm for A* is best search algorithm compared to DFS algorithm and Manhattan’s algorithm