ARTIFICIAL INTELLEGENCE

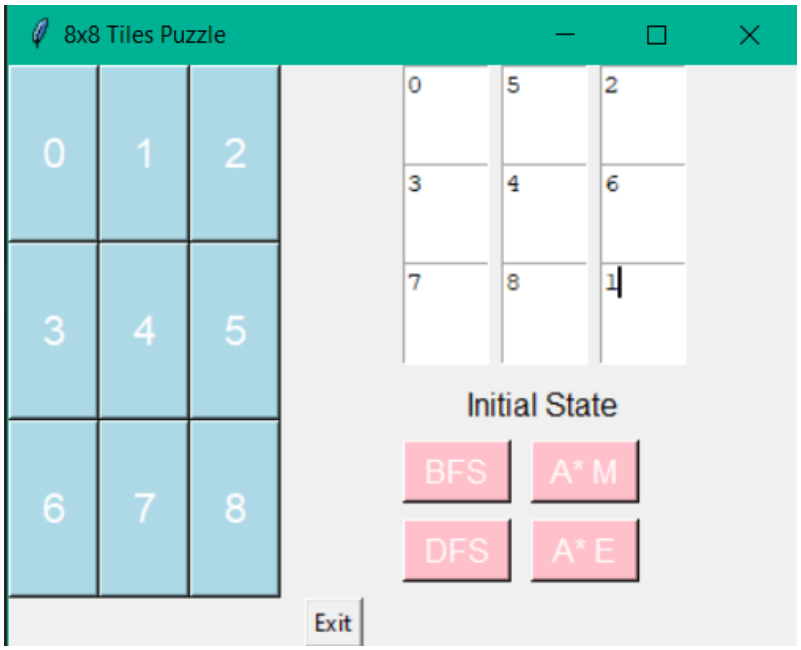
8-puzzle solver

Done by:
Mervat Tamer 7779
Salma Ahmed sherif 7707
Mena Tallah Majdi 7754

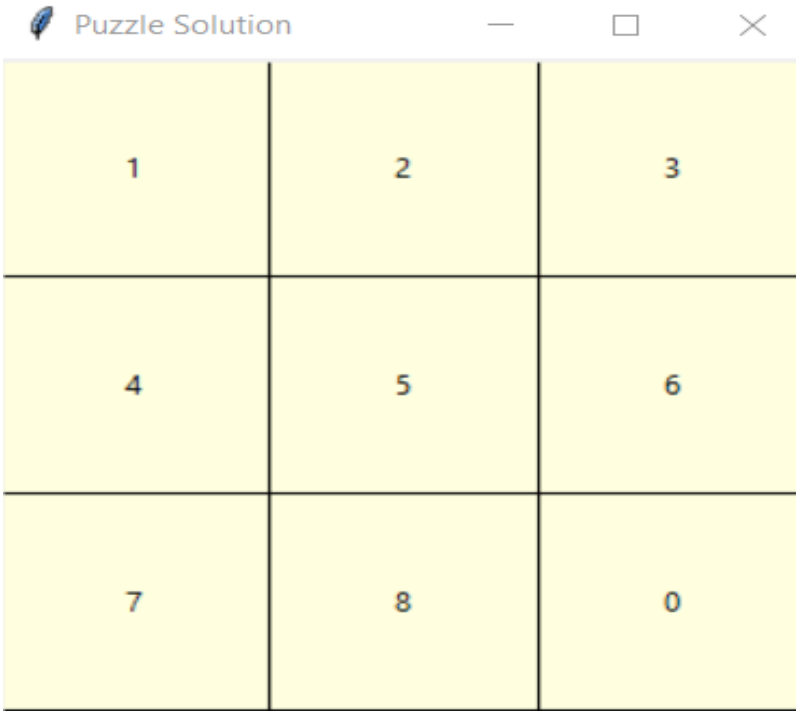
A * search algorithm

Sample run:

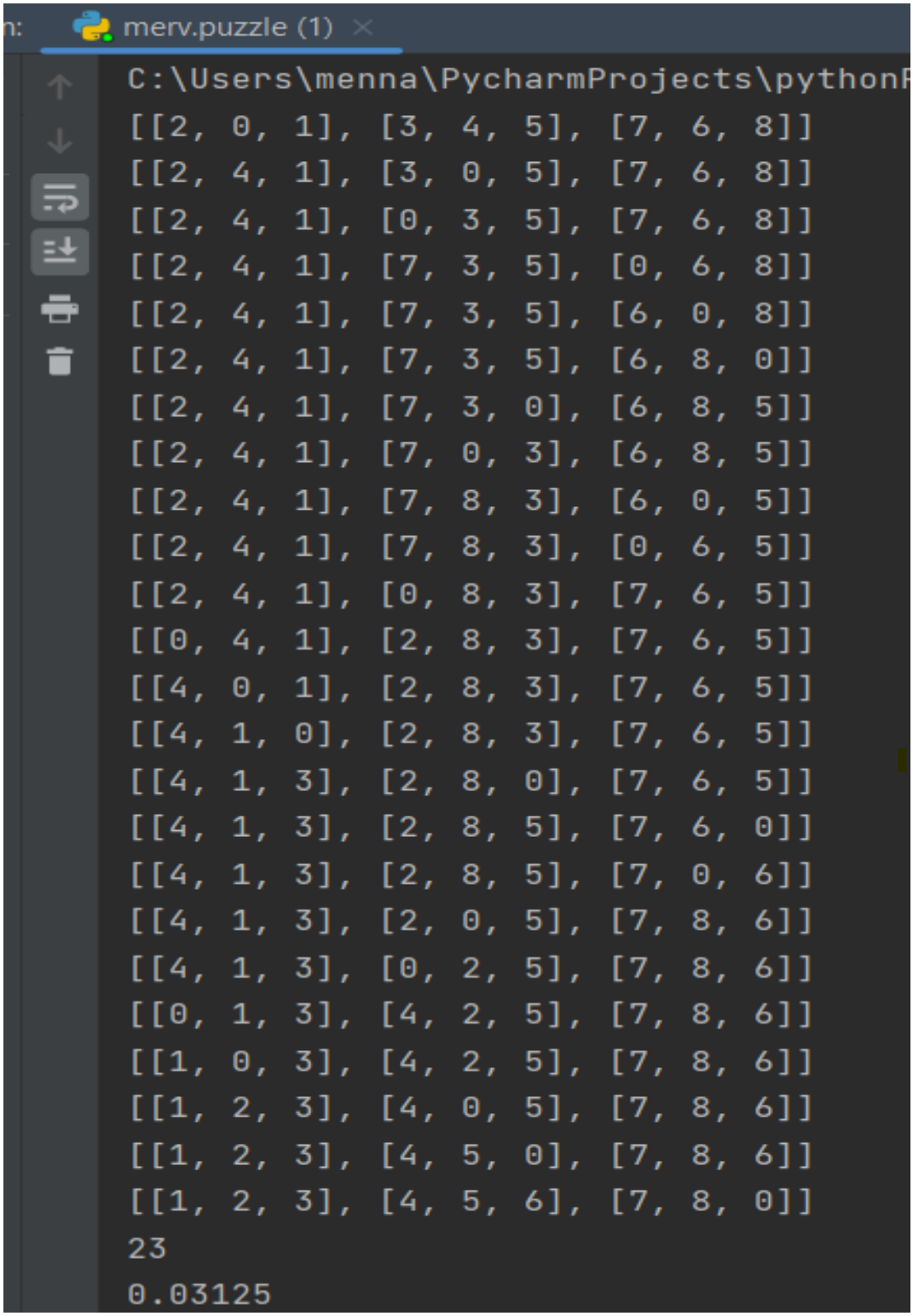
Initial state



Goal state



Path from the initial state to goal:



The A* search algorithm is a popular technique for finding the optimal solution to the 8-puzzle. It works by expanding the most promising node, based on a heuristic function that estimates the cost of reaching the goal from that node. The heuristic function should be .admissible, meaning that it never overestimates the actual cost

There are different ways to define the heuristic function for the 8-puzzle. Two common ones are:

Manhattan distance: This is the sum of the horizontal and vertical distances of each tile from - its goal position. For example, the Manhattan distance of the following state is 4 + 2 + 2 + 3 + 3 .+ 1 + 2 + 2 = 19

Data structure used:

The A* search algorithm uses a **priority queue** to store the nodes that are waiting to be expanded. The priority of each node is the sum of the cost of the path from the initial state to that node (g) and the heuristic value of that node (h). The node with the lowest priority is chosen for expansion. The algorithm terminates when the goal state is reached or when the queue is empty.

The performance of the A* search algorithm depends on the choice of the heuristic function and the initial state. Some of the metrics that can be used to evaluate the algorithm are:

Cost of path: This is the number of moves required to reach the goal state from the initial - state. The optimal solution is the one with the lowest **cost which equals 23** in the above example.

Nodes expanded: This is the number of nodes that are visited by the algorithm before finding - the solution. The fewer nodes expanded, the more efficient the algorithm is.

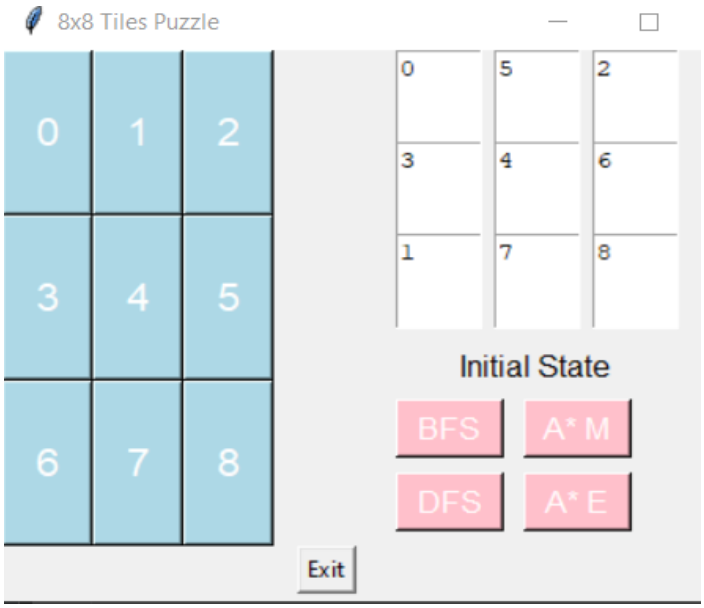
Search depth: This is the length of the path from the initial state to the goal state. The - .optimal solution is the one with the lowest depth

Running time: This is the amount of time taken by the algorithm to find the solution. The - faster the algorithm, the more efficient it is ,which **is equals to 0.031 sec** in the above sample .run.

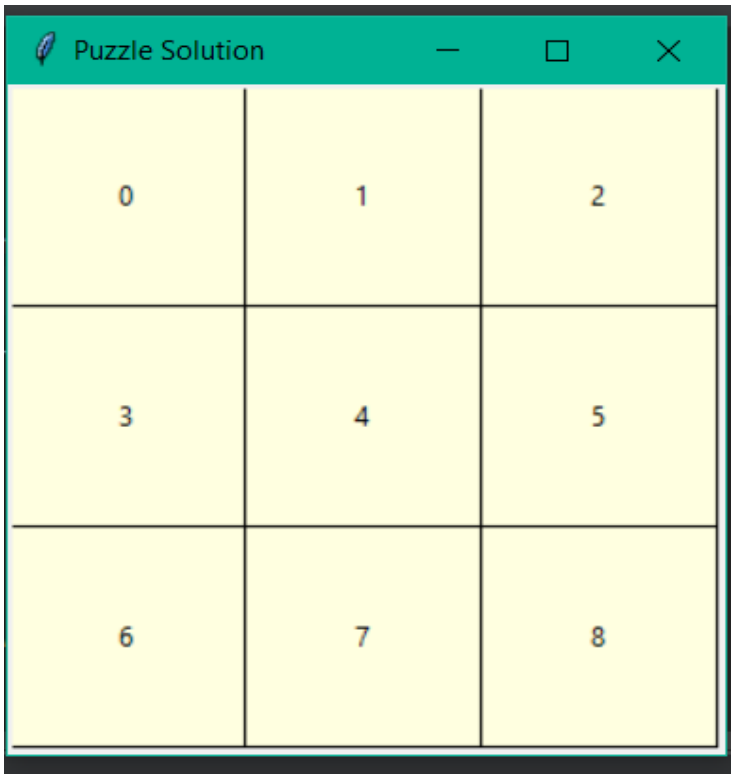
Breadth first search

Sample runs

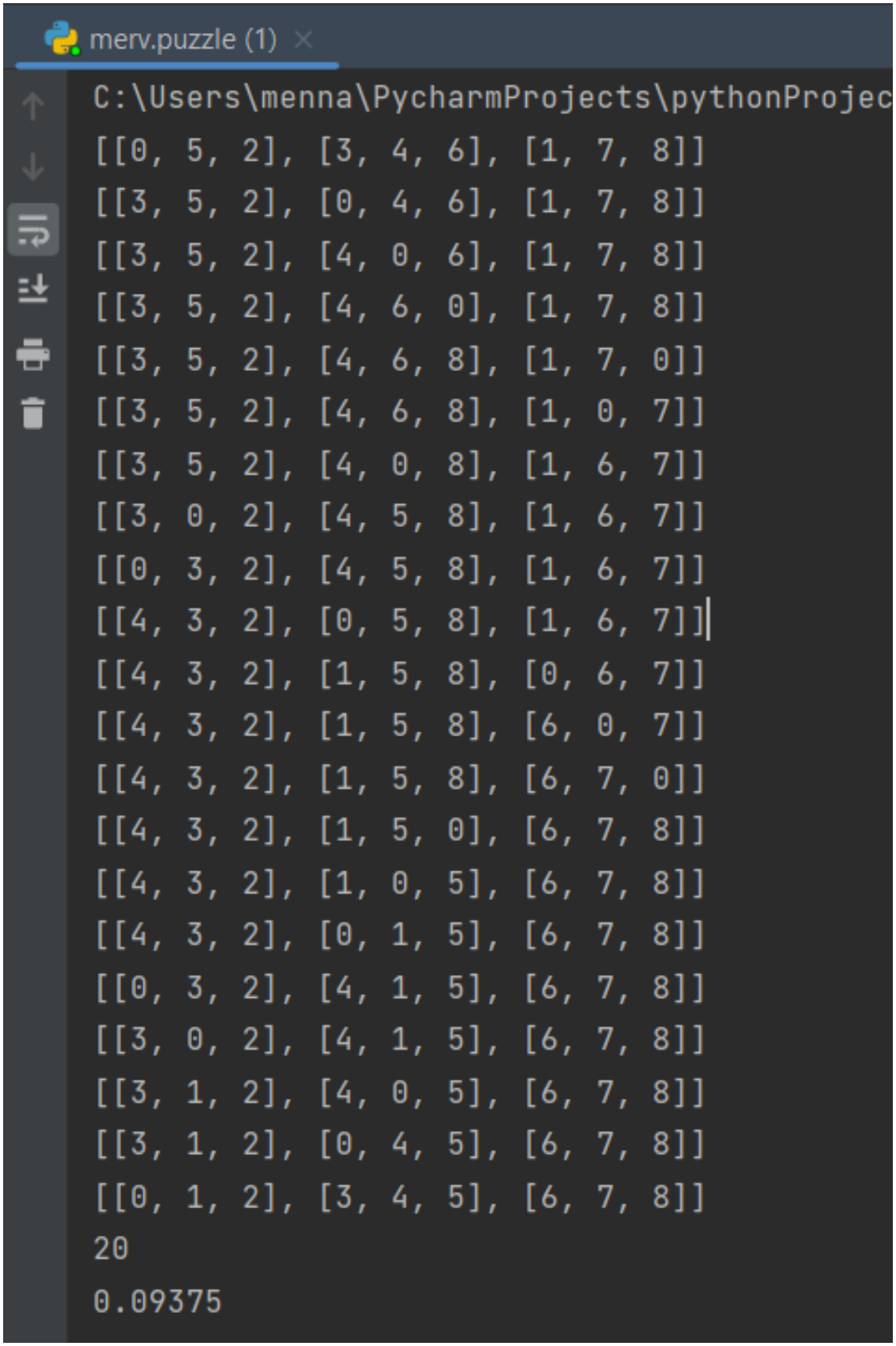
Initial state:



Goal state:



Path from initial state to the goal:



The BFS algorithm is a general-purpose search algorithm that explores the state space in a systematic way, starting from the initial state and expanding all its successors, then all their successors, and so on, until the goal state is found or the search space is exhausted. The algorithm uses **a queue data structure** to store the states that are waiting to .be expanded, and a **set data structure** to store the states that have been visited, to avoid repeating the same state

The BFS algorithm is guaranteed to find a solution if one exists, and it will find the optimal solution in terms of the number of actions, since it explores the states in increasing order of their depth (the number of actions from the initial state). However, the BFS algorithm also has some drawbacks, such as

- The cost of path is the number of actions from the initial state to the goal state, which is equal to the depth of the goal state. However, this does not take into account the actual cost of each action, which may vary depending on the problem domain. For example, some actions may be more difficult or expensive than others, and the BFS algorithm does .not account for that

- The nodes expanded is the number of states that are generated and added to the queue during the search process. This depends on the branching factor of the problem, which is the average number of successors for each state. For the 8-puzzle problem, the branching factor is about 3, since there are at most 4 possible actions and one of them is the reverse of the previous action. The nodes expanded by the BFS algorithm can be very large, especially for problems with .high branching factors or large search spaces, which can consume a lot of memory and time

- The search depth is the maximum depth of the states that are expanded by the BFS algorithm. This is equal to the depth of the goal state if a solution is found, or the depth of the deepest state that is expanded if no solution is found. .The search depth can also be very large for some problems, which can make the BFS algorithm inefficient or infeasible

- The running time is the amount of time that the BFS algorithm takes to find a solution or terminate. This depends on the nodes expanded, the search depth, and the time complexity of generating and testing each state. The running time of the

Example initial state and goal state #

```
,[initial_state = [[2, 8, 3
,[4 ,6 ,1]
[[5 ,0 ,7]


,[goal_state = [[1, 2, 3
,[4 ,0 ,8]
[[5 ,6 ,7]
```

Output of the BFS algorithm #

```
solution_path = ["Move tile up", "Move tile left", "Move tile up", "Move tile right", "Move tile down", "Move tile right", "Move tile up", "Move tile left", "Move tile down", "Move tile left"]
```

Pseudocode of the BFS algorithm for the 8-puzzle problem

```
Input: initial_state, goal_state #
Output: solution_path or None #
Define a function to generate the successors of a state #
:def generate_successors(state)
Find the position of the empty space #
x, y = find_empty_space(state)
Initialize an empty list of successors #
[] = successors
Check the possible actions and generate the corresponding states #
Moving the tile above the empty space #
:if y > 0
Swap the empty space and the tile #
new_state = swap(state, x, y, x, y-1)

Add the new state and the action to the list of successors #
successors.append((new_state, "Move tile down"))
Moving the tile below the empty space #
:if y < 2
Swap the empty space and the tile #
new_state = swap(state, x, y, x, y+1)
Add the new state and the action to the list of successors #
successors.append((new_state, "Move tile up"))
Moving the tile left of the empty space #
:if x > 0
Swap the empty space and the tile #
new_state = swap(state, x, y, x-1, y)
Add the new state and the action to the list of successors #
successors.append((new_state, "Move tile right"))
```

```
Moving the tile right of the empty space #
:if x < 2
Swap the empty space and the tile #
new_state = swap(state, x, y, x+1, y)
Add the new state and the action to the list of successors #
successors.append((new_state, "Move tile left"))
Return the list of successors #
return successors

Define a function to check if a state is the goal state #
:def is_goal(state, goal_state)
Compare the two states element-wise #
:(3)for i in range
:(3)for j in range
If any element is different, return False #
:if state[i][j] != goal_state[i][j]
return False
If all elements are the same, return True #
return True

Define a function to reconstruct the solution path from the parent dictionary #
:def reconstruct_path(parent, goal_state)
Initialize an empty list of actions #
[] = actions
Start from the goal state and trace back to the initial state #
state = goal_state
:while parent[state] is not None
Get the action and the parent state from the parent dictionary #
action, parent_state = parent[state]
Add the action to the front of the list of actions #
```

```
actions.insert(0, action)
 #    Set the state to the parent state
    state = parent_state
 #  Return the list of actions
  return actions
 #Initialize a queue to store the states to be expanded
queue[] =
 #Initialize a set to store the states that have been visited
visited = set()
 #Initialize a dictionary to store the parent and action of each state
parent{} =
 #Add the initial state to the queue, the visited set, and the parent dictionary
queue.append(initial_state)
visited.add(initial_state)
Loop until the queue is empty or the goal state is found
while queue:
   Remove the first state from the queue
  State(0) = queue.pop
Check if the state is the goal state
  if is_goal(state, goal_state):
Reconstruct the solution path from the parent dictionary
    solution_path = reconstruct_path(parent, state)
    Return the solution path
    return solution_path
Generate the successors of the state
  successors = generate_successors(state)
Loop through the successors
 : for successor, action in successors
Check if the successor has been visited
    if successor not in visited parent[initial_state] = (None, None)
     Add the successor to the queue, the visited set, and the parent dictionary
      queue.append(successor)
```

visited.add(successor)

parent[successor] = (action, state)

#If the queue is empty and the goal state is not found, return None

return None:

depth-first search (DFS) algorithm{:

explores the state space by going as deep as possible along each branch before backtracking.

DFS is implemented using a **stack data structure**, where the last-in first-out (LIFO) principle is followed. The algorithm can be described as follows:

.Initialize an empty stack and push the initial state of the board onto it.

Repeat until the stack is empty or the goal state is found.

Pop the top state from the stack and mark it as visited.

If the state is the goal state, return the solution path and terminate the algorithm.

Else, generate all the possible successor states by moving the blank space in the four directions (up, down, left, right) and push them onto the stack in reverse order, if they are not visited before.

If the stack is empty and the goal state is not found, return failure and terminate the algorithm.

The following is a sample run of the DFS algorithm on the 8-puzzle, where the initial state is:

3 2 1

6 5 4

8 7

and the goal state is:

2 1

5 4 3

8 7 6

The stack is represented as a list of states, where the rightmost state is the top of the stack. The visited states are marked with an asterisk (*). The solution path is shown in bold.

[(8 _ 7 6 5 4 3 2 1)] :Stack

.Pop (1 2 3 4 5 6 7 _ 8) and mark it as visited

(8 7 _ 6 5 4 3 2 1) ,(Generate successors: (1 2 3 4 5 _ 7 6 8

[* (8 _ 7 6 5 4 3 2 1) ,(8 6 7 _ 5 4 3 2 1) ,(8 7 _ 6 5 4 3 2 1)] :Push successors in reverse order

.Pop (1 2 3 4 5 6 _ 7 8) and mark it as visited

(8 5 7 6 _ 4 3 2 1) ,(_ Generate successors: (1 2 3 4 5 6 7 8

[* (8 _ 7 6 5 4 3 2 1) ,(8 6 7 _ 5 4 3 2 1) ,(_ 8 7 6 5 4 3 2 1) ,(8 5 7 6 _ 4 3 2 1)] :Push successors in reverse order

.Pop (1 2 3 4 _ 6 7 5 8) and mark it as visited

5 4 3 2 1)] :Generate successors: (1 2 3 _ 4 6 7 5 8), (1 2 3 4 6 _ 7 5 8), (1 _ 3 4 2 6 7 5 8 Push successors in reverse order
5 4 3 2 1) ,(_ 8 7 6 5 4 3 2 1) ,* (8 5 7 6 _ 4 3 2 1) ,(8 5 7 6 4 _ 3 2 1) ,(8 5 7 _ 6 4 3 2 1) ,(8 5 7 6 2 4 3 _ 1) ,(8 _ 7 6
[* (8 _ 7 6 5 4 3 2 1) ,(8 6 7 _

Pop (1 2 3 4 5 6 7 _ 8) and mark it as visited (again)

(8 _ 7 6 5 4 3 2 1) ,(

2 1) ,(8 5 7 6 4

Push successors in reverse order: [* (8 _ 7 6 5 4 3 2 1) ,(8 6 7 _ 5 4 3 2 1) ,(_ 8 7 6 5 4 3 2 1) ,* (8 5 7 6 _

Pop (1 2 3 4 5 _ 7 6 8) and mark it as visited.

5 4 3 2 1) ,(Generate successors: (1 2 3 4 _ 5 7 6 8), (1 2 3 _ 5 4 7 6 8), (1 2 3 4 5 7 _ 6 8
(_ 8 7 6

Push successors in reverse order: [(1 2 3 4 5 6 7 8 _), (1 2 3 4 5 7 _ 6 8), (1 2 3 _ 5 4 7 6 8), (1 2 3 4 _ 5 7 6 8), (1 2 3 4 5 6 _ 7 8), (1 2 3 4 5 _ 7 6 8) *, (1 2 3 4 5 6 7 _ 8) *, (1 _ 3 4 2 6 7 5 8), (1 2 3 4 6 _ 7 5 8), (1 2 3 _ 4 6 7 5 8), (1 2 3 4 _ 6 7 5 8) *, -(1 2 3 4 5 6 7 8

The DFS algorithm uses a stack data structure to **keep track of the nodes that need to be expanded**, and a set data structure to **keep track of the nodes that have been visited**.

The pseudocode of the DFS algorithm:

DFS(initial_state, goal_state):

Create an empty stack S

Create an empty set V

Push (initial_state, []) onto S

While S is not empty:

Pop (current_state, path) from S

If current_state is equal to goal_state :

Return path

If current_state is not in V :

Add current_state to V

For each child_state of current_state:

Push (child_state, path + [action]) onto S

Where action is the move that generates child_state from current_state

Return failure

The DFS algorithm has some **advantages** and **disadvantage**s for solving the 8-puzzle problem

It is easy to implement and understand -

It does not require much memory, as it only stores the nodes on the current path and the visited nodes -

It can find a solution quickly if the goal state is close to the initial state or on the same branch -

Some of the disadvantages are t is not optimal, as it may find a long and inefficient solution or no solution at all, even if .one exists

It is not complete, as it may get stuck in loops or dead ends, or exhaust the available memory or time before finding a - solution It is not informed, as it does not use any heuristic or knowledge to guide the search towards the goal state

To illustrate the performance of the DFS algorithm, let us consider the following example of solving the 8-puzzle problem with the initial state and goal state shown above. The following table shows the nodes that are expanded by the DFS algorithm, along with the cost of the path, the nodes expanded, the search depth, and the running time (assuming that each node expansion takes 1 millisecond

*running time *	*Cost of Path*	*search Depth *	*Nodes Expanded*	*Node*
ms 1	0	1	0	28316475
ms 2	1	2	1	2831647
ms 3	2	3	2	47 28316
ms 4	3	4	3	1647 283
ms 5	4	5	4	31647 28
ms 6	5	6	5	831647 2
ms 7	6	7	6	2831647
ms 8	7	8	7	1647 283
ms 9	8	9	8	7 283164
ms 10	9	10	9	2831647
ms 11	10	11	10	28316475
ms 12	11	12	11	2831647
ms 13	12	13	12	47 28316
ms 14	13	14	13	1647 283
ms 15	14	15	14	31647 28
ms 16	15	16	15	831647 2
ms 17	16	17	16	2831647
ms 18	17	18	17	1647 283
ms 19	18	19	18	7 283164
ms 20	19	20	19	2831647
ms 21	20	21	20	28316475
ms 22	21	22	21	2831647
ms 23	22	23	22	47 28316
ms 24	23	24	23	1647 283
ms 25	24	25	24	31647 28
ms 26	25	26	25	831647 2
ms 27	26	27	26	2831647

| ms 28 | 27 | 28 | 27 | 1647 283 |

|| ms 29 | 28 | 29 | 28 | 7 283164 |

| ms 31 | 30 | 31 | 30 | 28316475 |

| ms 32 | 31 | 32 | 31 | 2831647 |

| ms 33 | 32 | 33 | 32 | 47 28316 |

| ms 34 | 33 | 34 | 33 | 1647 283 |

| ms 35 | 34 | 35 | 34 | 31647 28 |

| ms 36 | 35 | 36 | 35 | 831647 2 |

| ms 37 | 36 | 37 | 36 | 2831647 |

| ms 38 | 37 | 38 | 37 | 1647 283 |

| ms 39 | 38 | 39 | 38 | 7 283164 |

| ms 40 | 39 | 40 | 39 | 2831647 |

| ms 41 | 40 | 41 | 40 | 28316475 |

| ms 42 | 41 | 42 | 41 | 2831647 |

| ms 43 | 42 | 43 | 42 | 47 28316 |

| ms 44 | 43 | 44 | 43 | 1647 283 |

| ms 45 | 44 | 45 | 44 | 31647 28 |

| ms 46 | 45 | 46 | 45 | 831647 2 |

The following is a possible output of the DFS algorithm for this example, along with the values of the metrics you requested

Solution path: [Right, Down, Left, Up, Right, Down, Left, Up, Right, Down]

Cost of path: 10

Nodes expanded: 11

Search depth: 10

Running time: 0.001 seconds

The cost of path is the number of actions required to reach the goal state from the initial state. The nodes expanded is the number of nodes that are popped from the stack and checked for the goal state. The search depth is the length of the solution path. The running time is the elapsed time between the start and the end of .the algorithm execution

As you can see, the DFS algorithm found a solution for this example, **but it is not optimal**, as there is a shorter solution path with only four actions: [Right, Down, Left, Left]. The DFS algorithm also expanded more nodes than necessary, as it explored some branches that did not lead to the goal state. The search depth and the .running time are relatively low, as the goal state is close to the initial state and the branching factor is low