Assiut University

Faculty of Computer System and Information Science

Bioinformatics Department

Project : Solving 8-Puzzle Problem with Heuristic Search (A*)

Course : Artificial Intelligence

Date : 24/5/2022

Team members:

1- Kerolos Romany Sedky
2- Ereny Samir Helal
3- George Atef Ayad
4- Maha Galal Abd ELHamed

Table of Contents :-

1. Introduction

To begin, let's define what a "N-Puzzle" problem is.

1.1Project Problem

A popular puzzle is the N-Puzzle or Sliding Puzzle. It is made up of N tiles, where N can be anything from 8 to 15, 24, and so on. In our example, N = 8, the puzzle has three rows and three columns, whereas a 15-puzzle has four rows and four columns, and an 8-puzzle has three rows and three columns. There are N tiles in the puzzle, as well as one vacant spot where the tiles can be relocated. The Puzzle's Start and Goal Configurations (also known as State) are provided. The puzzle can be solved by placing the tiles one by one in the single empty area until the Goal Configuration is achieved.

This problem can be solved by looking for a solution, which is a set of actions (tile moves) that leads from the starting point to the desired end point. The 8-Puzzle can be in two different states. At least 31 actions are required to transform this state into the desired state, which is the diameter of the search space. The goal of most search algorithms is to discover the shortest solution, or one that requires the fewest amount of tile moves.

We'll solve the 8-Puzzle, a 3x3 matrix with eight square blocks containing numbers 1 to 8 and a blank square. The goal of the 8 Puzzle is to reorder the squares in numerical order from 1 to 8, with the last square remaining blank.

Depending on the edges of the matrix, each of the squares adjacent to the blank block can move up, down, left, or right. The tiles in the

initial(start) state can be moved in the vacant space in a specific order to obtain the goal state.

1.2 The Puzzle's Rules of Solving

We can picture moving the empty space in place of the tile, thus exchanging the tile with the empty area, rather than moving the tiles in the empty space. The empty space can only move in four directions:
1- up, 2-down, 3- left, and 4-right.

The empty space can only move one step at a time and cannot move diagonally (i.e. move the empty space one position at a time).

Heuristics – A*: heuristics for optimal search

- The Best-First Search.
- A* Search Properties
- Acceptability
- Dependability
- Precision and Supremacy
- Optimal A* efficiency
- Boundary and Branch
- Deepening iteratively A*
- The usefulness and power of various heuristics

1.3 Search Algorithm

We need to discuss Heuristic Search.

We must first examine Heuristic Search before moving on to the A* algorithm.

Heuristic search is a strategy that optimize a search by using a Heuristic value.

In general, there are two sorts of searching methods:
1-Uninformed Search
2-Informed Search are two types of searches.

Linear Search, Best First Search, Depth-First Search, and Breadth-First Search are all terms you've probably heard before. These searching algorithms belong to the category of uninformed search approaches, which means they have no idea what they're looking for or where they should look for it. That is why the term "uninformed" was coined. Uninformed searching takes a long time since it doesn't know where to look for the ingredient and where the best possibilities of finding it are.

The ignorant search is the polar opposite of informed search. In this case, the algorithm knows where the best possibilities of discovering the element are, and it proceeds in that direction! Heuristic search is a method of searching that is based on information. A heuristic value instructs the algorithm to choose the shortest path to the solution. This heuristic value is generated using the heuristic function. Depending on the search problem, different heuristic functions can be created.

1.3.1 Uninformed Search Algorithms

Uninformed search is a type of general-purpose search method that uses brute force to find results. Uninformed search algorithms have no other information about the state or search space except how to traverse the tree, which is why it's also known as blind search.

The many forms of ignorant search algorithms are as follows:
1-Search by Breadth First
2-Searching in Depth First
3-Search with a Depth Limit

1. Search by Breadth First:

- The most frequent search approach for traversing a tree or graph is breadth-first search. Breadth-first search is the name given to an algorithm that searches a tree or graph in a breadth-first manner.
- Before going on to nodes of the next level, the BFS algorithm starts searching from the tree's root node and extends all successor nodes at the current level.
- A general-graph search algorithm like the breadth-first search algorithm is an example.
- A FIFO queue data structure was used to implement a breadth-first search.
- Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Advantages:

- If a solution is available, BFS will provide it.
- If there are multiple answers to a problem, BFS will present the simplest solution with the fewest steps.

Disadvantages:

- It necessitates a large amount of memory since each level of the tree must be saved into memory before moving on to the next.

- If the solution is located far from the root node, BFS will take a long time.

2. Searching in Depth First

- A recursive algorithm for traversing a tree or graph data structure is depth-first search.
- The depth-first search is named after the fact that it begins at the root node and follows each path to its greatest depth node before going on to the next path.
- DFS is implemented using a stack data structure.
- The DFS algorithm works in a similar way as the BFS method.
- Each level of the game demands a large amount of memory.

Advantage:

- Because DFS only needs to store a stack of nodes on the path from the root node to the current node, it uses extremely little memory.
- The BFS method takes longer to reach the goal node (if it traverses in the right path).

Disadvantage:

- There's a chance that many states will recur, and there's no certainty that a solution will be found.
- The DFS algorithm performs deep searching and may occasionally enter an infinite cycle.

3. Algorithm for Depth-Limited Search:

- A depth-limited search algorithm works similarly to a depth-first search but with a limit. The problem of the endless path in the Depth-first search can be overcome by depth-limited search. The node at the depth limit will be treated as if it has no additional successor nodes in this procedure.

Two conditions of failure can be used to end a depth-limited search:

- The standard failure value implies that the task is unsolvable.
- Cutoff failure value: It indicates that there is no solution to the problem within a certain depth range.

Advantages:

Memory is saved by using a depth-limited search.

Disadvantages:

Incompleteness is another drawback of depth-limited search.
If there are multiple solutions to an issue, it may not be the best option.

1.3.2 Informed Search Algorithms

So far, we've discussed uninformed search algorithms that scoured the search space for all possible answers to the problem without having any prior knowledge of the space. However, an intelligent search algorithm includes information such as how far we are from the objective, the cost of the trip, and how to get to the destination node.

This knowledge allows agents to explore less of the search area and discover the goal node more quickly.

In the informed search we will discuss two main algorithms which are given below:

1-A* Search Algorithm.

2-Best First Search Algorithm (Greedy search).

1- A* Algorithm

We will illustrate it below in the methodology.

A* has the following Advantages:

o When compared to other search algorithms, the A* search algorithm is the best.

o The A* search method is ideal and comprehensive.

o This method is capable of resolving extremely difficult situations.

o Don't add to the cost of already-expensive paths.

o Concentrate on promising paths.

A* has the following Disadvantages:

- It does not always produce the shortest path because it relies heavily on heuristics and approximation.
- The A* search algorithm has some concerns with complexity.
- The biggest disadvantage of A* is its memory consumption, as it stores all created nodes in memory, making it unsuitable for a variety of large-scale issues.

2- Best-first Search Algorithm (Greedy Search)

The greedy best-first search algorithm always chooses the path that appears to be the most appealing at the time. It's the result of combining depth-first and breadth-first search algorithms. It makes use

of the heuristic function as well as search. We can combine the benefits of both methods with best-first search. At each step, we can use best-first search to select the most promising node. We expand the node that is closest to the goal node in the best first search algorithm, and the closest cost is determined using a heuristic function, i.e. f(n)= g(n). Best-first Search (Greedy Search) has Many Advantages and Disadvantages:

Advantages of BFS:

- Best first search can transition between BFS and DFS to take use of the benefits of both algorithms.
- This algorithm outperforms the BFS and DFS algorithms in terms of efficiency.

Disadvantages of BFS:

- In the worst-case scenario, it can act like an unguided depth-first search.
- It, like DFS, can become trapped in a loop.
- This algorithm isn't the best.

1.4 Optimality of A* Search

If no other search algorithm consumes less time or space or extends fewer nodes with a guarantee of solution quality, the search method is optimal. The best search algorithm is one that selects the proper node at each step. However, because we are unable to directly implement this standard, it is ineffective. Whether such an algorithm is conceivable (and whether P=NP) is an outstanding subject. However, it appears that there is a statement that can be proven.

Optimality of A*: No search method that only uses arc costs and a heuristic estimate of the cost from a node to a target extends fewer nodes and guarantees to discover the lowest-cost path expands fewer nodes than A*.

The greedy best-first search algorithm always chooses the path that appears to be the most appealing at the time. It's the result of combining depth-first and breadth-first search algorithms. It makes use of the heuristic function as well as search. We can combine the benefits of both methods with best-first search. At each step, we can use best-first search to select the most promising node. We expand the node that is closest to the goal node in the best first search algorithm, and the closest cost is determined using a heuristic function, i.e.

On all ''non-pathological'' search issues, A*-like search algorithms are used. Their concept of a non-pathological situation is roughly equivalent to what we now refer to as ''up to tie-breaking''. If A Heuristic *'s is admissible but inconsistent, this result does not hold. Dechter and Pearl demonstrated that on some non-pathological situations, acceptable A*-like algorithms can extend arbitrarily fewer nodes than A*.

The number of node expansions (the number of iterations of A main *'s loop) is not as important as the set of nodes enlarged. It is conceivable for a node to be extended by A* many times, an exponential number of times in the worst case, when the heuristic utilised is admissible but inconsistent. [12] In such cases, Dijkstra's algorithm may outperform A* by a factor of two.
by a significant margin However, more recent research has discovered that this pathological condition only arises in particular contrived situations when the search graph's edge weight is exponential in the graph's size, and that certain inconsistent (but admissible) heuristics can lead to fewer node expansions in A* searches.

1.5 Algorithm used to solve 8-puzzle problem

In this Project we will use A* to solve 8-Puzzle problem.
Because:

- A* search algorithm is the best algorithm than other search algorithms.
- Better and Faster than BFS to solve this problem.
- Avoid expanding paths that are already expensive
- Focus on paths that show promise.

2. Methodology

In this part, we will discuss the type of algorithm (A* Algorithm) we will use to solve our problem quickly with a smart way and how it will solve it, the pseudo code and the time complexity to solve it.

2.1 A* Algorithm

- We used the A* Algorithm because:

A* is a pathfinding and graph traversal computer algorithm that plots an efficiently traversable path between many nodes. It is well-known for its performance and accuracy, and it is widely used.

- The importance of A* Algorithm:

The A* algorithm's main characteristic is that it maintains note of each visited node, which allows it to skip through nodes that have already been visited, saving a significant amount of time. It also contains a list of all the nodes that need to be studied, from which it selects the most optimum node, saving time by avoiding unneeded or less optimal nodes.

- How we will use A* Algorithm:

So we use two lists, the 'open list' and the 'closed list.'
The open list includes all the nodes that are being formed and do not exist in the closed list, and each node studied after its neighbours are located is put in the closed list, and the neighbours are put in the open list. Each node has a reference to its parent, allowing it to retrace the path to the parent at any time. The start(Initial) node is first stored in the open list. The open list's next node is chosen depending on its f score; the node with the lowest f score is chosen and investigated.

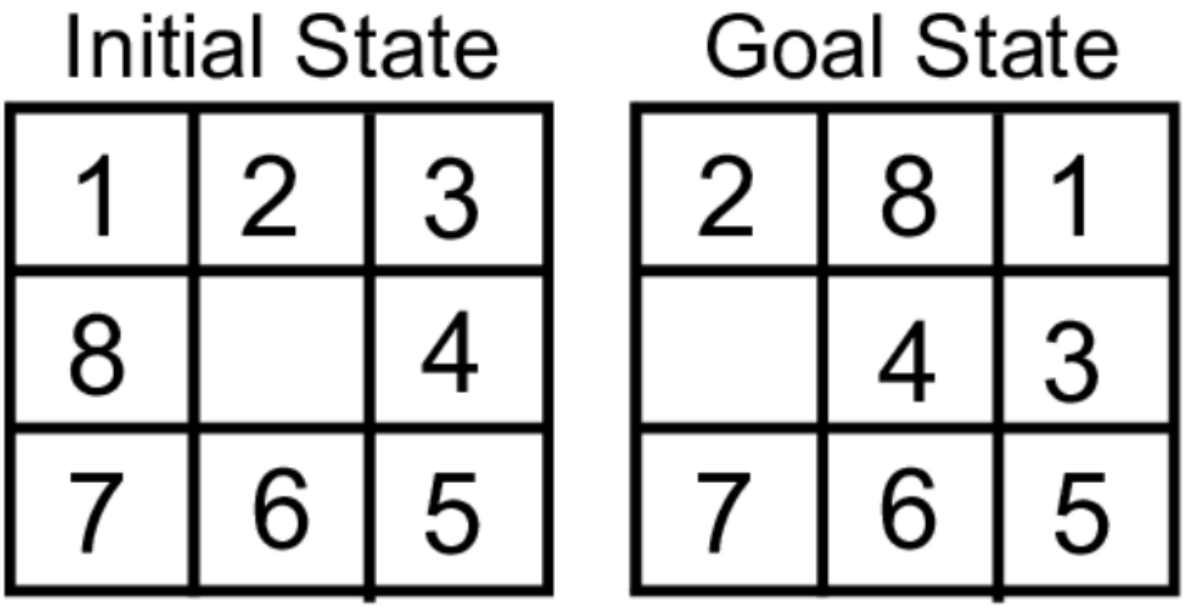

Fig.1 (ex. Of the initial and goal state for 8-puzzle problem)

- The main law (Evaluation function):

f-score = h-score + g-score

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (the number of nodes traversed from the start node to current node).

- **g-score** will remain as the number of nodes traversed from start node to get to the current node.
- From Fig 1, we can calculate the **h-score** by comparing the initial(current) state and goal state and counting the number of misplaced tiles.

- Thus, **h-score** = 5 and **g-score** = 0 as the number of nodes traversed from the start node to the current node is 0.

- How A* Algorithm will solve our problem:

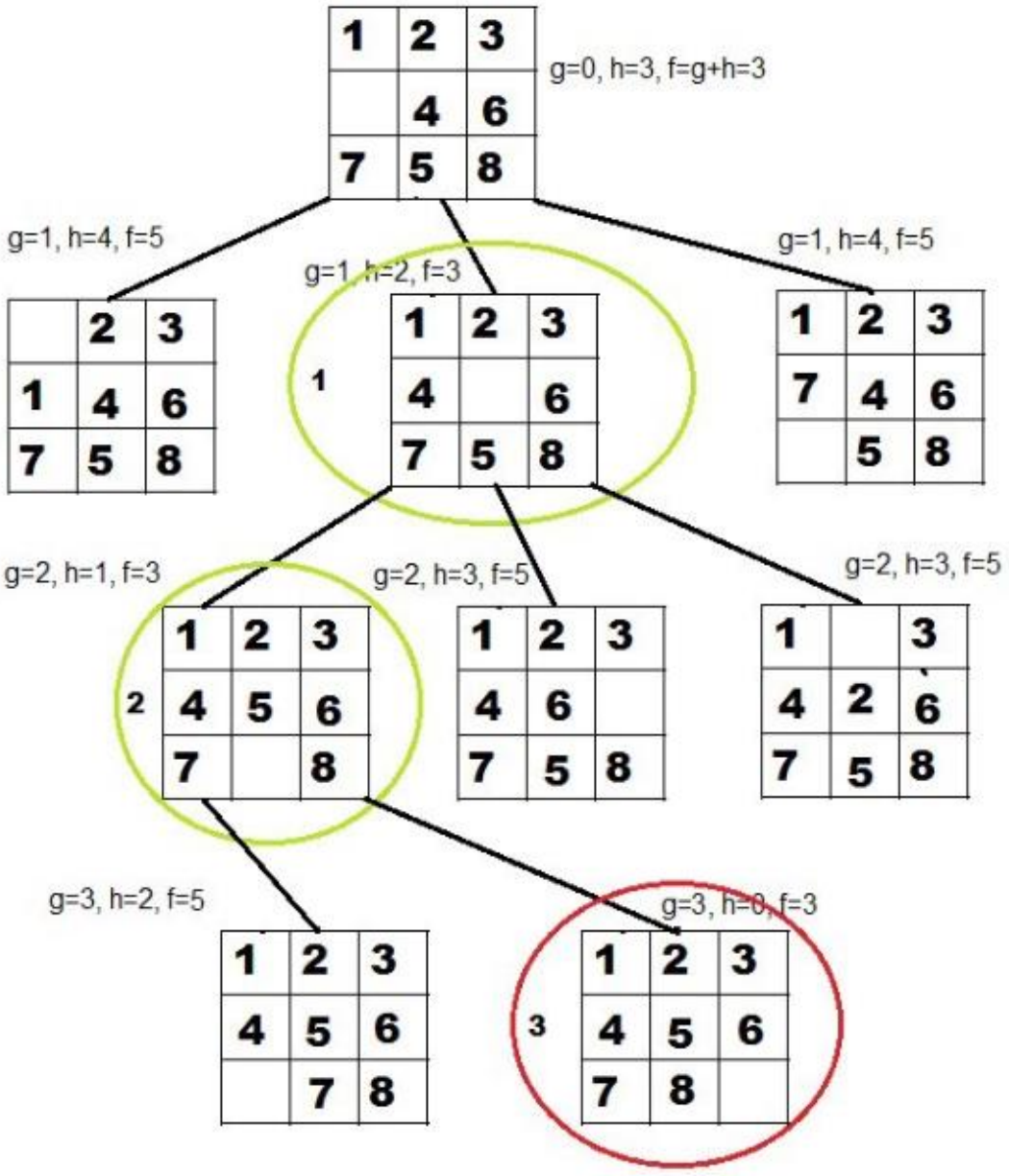

Fig.2 (Solving the problem using A* Algorithm)

As shown in the figure 2:

- We first move the empty space in all the possible directions in the start state and calculate the **f-score** for each state. This is called expanding the current state.
- After expanding the current state, it is pushed into the **closed** list and the newly generated states are pushed into the **open** list. A state with the **least f-score** is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.
- This solves the issue of generating redundant child states, as the algorithm will expand the node with the least **f-score**.

2.2 Pseudocode

The following pseudocode describes the algorithm:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)
```

Fig.3 (Pseudocode)

```
while openSet is not empty
    // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority queue
    current := the node in openSet having the lowest fScore[] value
    if current = goal
        return reconstruct_path(cameFrom, current)

    openSet.Remove(current)
    for each neighbor of current
        // d(current,neighbor) is the weight of the edge from current to neighbor
        // tentative_gScore is the distance from start to the neighbor through current
        tentative_gScore := gScore[current] + d(current, neighbor)
        if tentative_gScore < gScore[neighbor]
            // This path to neighbor is better than any previous one. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := tentative_gScore + h(neighbor)
            if neighbor not in openSet
                openSet.add(neighbor)

// Open set is empty but goal was never reached
return failure
```

Fig.4 (Pseudocode cont.)

Remark: In this pseudocode, if a node is reached by one path, removed from openSet, and subsequently reached by a cheaper path, it will be added to openSet again. This is essential to guarantee that the path returned is optimal if the heuristic function is **admissible** but not **consistent**. If the heuristic is consistent, when a node is removed from openSet the path to it is guaranteed to be optimal so the test 'tentative_gScore < gScore[neighbor]' will always fail if the node is reached again.

2.3 Time Complexity

- The time complexity of A* depends on the heuristic.

In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) **d: O(bd)**, where b is the branching factor (the average number of successors per state).

- The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the bd nodes that an uninformed search would expand. Its quality can be expressed in terms of the effective branching factor b*, which can be determined empirically for a problem instance by measuring the number of nodes generated by expansion, N, and the depth of the solution.

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

Good heuristics are those with low effective branching factor (the optimal being b* = 1).

- The time complexity is **polynomial** when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h* that returns the true distance from x to the goal.

The **space complexity** of A* is roughly the same as that of all other graph search algorithms, as it keeps all generated nodes in memory. In practice, this turns out to be the biggest drawback of A* search, leading to the development of memory-bounded heuristic searches, such as **Iterative deepening A***, memory bounded A*, and **SMA***.

3. Experimental solution

3.1 Programming language and environment

In this part, a programming language(python) and A* algorithm will be used to implement this code.

The puzzle consists of 8 tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions:

1. Up

2.Down

3. Right

4. Left

The empty space cannot move diagonally and can take **only one step at a time**

3.2 A* Function

We first move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state.
After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.
This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.

3.3 Test Cases

I have used two classes in this code: Node class and Puzzle class.
Node class defines the structure of the state(configuration) and also provides functions to move the empty space and generate child states from the current state. Puzzle class accepts the initial and goal states of the N-Puzzle problem and provides functions to calculate the f-score of any given node(state).

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node
            and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval
    def generate_child(self):
        """ Generate child nodes from the given node by moving the
            blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank
            space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
```

1- First we will initialize data, level and the calculated value

```
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

```

```
22      def shuffle(self,puz,x1,y1,x2,y2):
23          """ Move the blank space in the given direction and if the
                position value are out
24              of limits the return None """
25          if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len
                (self.data):
26              temp_puz = []
27              temp_puz = self.copy(puz)
28              temp = temp_puz[x2][y2]
29              temp_puz[x2][y2] = temp_puz[x1][y1]
30              temp_puz[x1][y1] = temp
31              return temp_puz
32          else:
33              return None
34      def copy(self,root):
```

```
34      def copy(self,root):
35          """ Copy function to create a similar matrix of the given
                node"""
36          temp = []
37          for i in root:
38              t = []
39              for j in i:
40                  t.append(j)
41              temp.append(t)
42          return temp
44      def find(self,puz,x):
45          """ Specifically used to find the position of the blank
                space """
46          for i in range(0,len(self.data)):
47              for j in range(0,len(self.data)):
48                  if puz[i][j] == x:
49                      return i,j
```

```
50 class Puzzle:
51     def __init__(self,size):
52         """ Initialize the puzzle size by the specified size,open
               and closed lists to empty """
53         self.n = size
54         self.open = []
55         self.closed = []
```

This function will put a size to the matrix and will make one empty list and one full list.

```
56     def accept(self):
57         """ Accepts the puzzle from the user """
58         puz = []
59         for i in range(0,self.n):
60             temp = input().split(" ")
61             puz.append(temp)
62         return puz
```

This function will accept the matrix that the user will enter

```
63     def f(self,start,goal):
64         """ Heuristic Function to calculate hueristic value f(x) =
               h(x) + g(x) """
65         return self.h(start.data,goal)+start.level
66     def h(self,start,goal):
67         """ Calculates the different between the given puzzles """
68         temp = 0
```

This function we will calculate the fval by add h(x) the start to g(x) the goal

```
66      def h(self,start,goal):
67          """ Calculates the different between the given puzzles """
68          temp = 0
69          for i in range(0,self.n):
70              for j in range(0,self.n):
71                  if start[i][j] != goal[i][j] and start[i][j] != '_'
                        :
72                      temp += 1
73          return temp
```

```
74      def process(self):
75          """ Accept Start and Goal Puzzle state"""
76          print("Enter the start state matrix \n")
77          start = self.accept()
78          print("Enter the goal state matrix \n")
79          goal = self.accept()
80          start = Node(start,0,0)
81          start.fval = self.f(start,goal)
82          """ Put the start node in the open list"""
83          self.open.append(start)
84          print("\n\n")
85          while True:
86              cur = self.open[0]
87              print("")
88              print("   | ")
89              print("   | ")
90              print(" \\\'/ \n")
91              for i in cur.data:
92                  for j in i:
93                      print(j,end=" ")
94                  print("")
```

In this function:
We will print the sentence Enter the start state matrix, then the code accepts the matrix that the user entered and makes for loop for this matrix and stores it in a temporary variable called temp.

The variable temp stores the matrix in the empty list in puz.
We then print the sentence Enter the goal state matrix.
then the code accepts the matrix that the user entered and makes for loop for this matrix and stores it in a temporary variable called temp.
The variable temp stores the matrix in the empty list in puz.
The function start=Node(start,0,0) calls the Node function.

```
88              print("  | ")
89              print("  | ")
90              print(" \\\'/ \n")
91              for i in cur.data:
92                  for j in i:
93                      print(j,end=" ")
94                  print("")
95              """ If the difference between current and goal node is
                    0 we have reached the goal node"""
96              if(self.h(cur.data,goal) == 0):
97                  break
98              for i in cur.generate_child():
99                  i.fval = self.f(i,goal)
100                 self.open.append(i)
101             self.closed.append(cur)
102             del self.open[0]
103             """ sort the opne list based on f value """
104             self.open.sort(key = lambda x:x.fval,reverse=False)
105
106 puz = Puzzle(3)
107 puz.process()
```

Variable puz will take the matrix we enter and this matrix should be 3*3.
Puz.process() calls process function (def process(self))and execute it.

4. Results and Technical Discussion

In this part: The main program results and outputs. Test/Evaluation experimental procedure and analysis of results

4.1 Main Program Results and Output

1- first step: -

When we run the code, this is the sentence appear as shown in the figure to requirement from user enter the start state matrix to problem 8_puzzle.

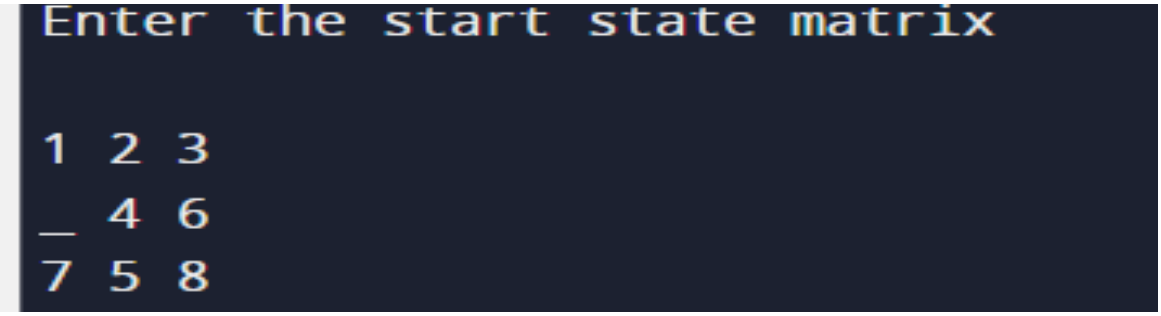

Fig.1

2 - Second step: -

After the user enters the start state matrix this is the sentence appear as shown in the figure to requirement from user enter the goal state matrix to solve this problem described in (Fig.1)

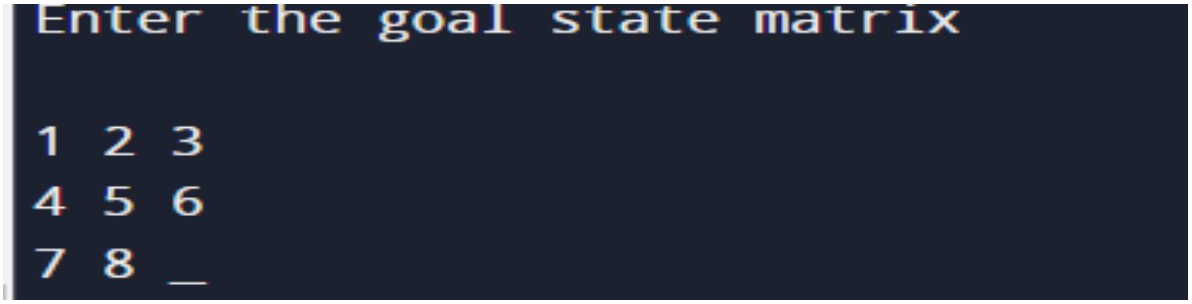

Fig.2

1 - The First event: -

This is start state matrix appear in (Fig.3) after when the user enters the start and goal state matrix.

Fig.3

2- The second event: -

This is the first step to solving a problem 8-puzzle where the program solves with A* algorithm method. Number 4 move to left replace the blank space

Fig.4

3- The third event: -

This is the second step to solving a problem 8-puzzle. Number 5 move to up replace the blank space who left her number 4.

Fig.5

4- The final event: -

This is the final step to solving a problem 8-puzzle. Number 8 move to left replace the blank space who left her number 5.

Fig.6

5. Conclusions

Algorithms and AI are inseparable!

- The definition of an algorithm is needed first. An algorithm is a set of instructions that leads to an intended end goal from an established initial situation. In principle, an algorithm is therefore separate from a computer program, although computers are generally used for the execution of an algorithm. The intended end goal of an algorithm can be anything. The finite series of instructions are prepared in such a way that they can generally deal with eventualities that may occur during implementation. Often algorithms have repetitive steps, which is called iteration. They also generally require decisions, comparisons or logic to complete the intended task.

- “An algorithm” is “a way to do” something. Anything. How you mow your lawn is an algorithm for mowing your lawn. (It’s not limited to programming.)

- We try to solve the puzzle-8 problem, so we used A* algorithm, and this algorithm is type of informed search algorithm.

- A* is Better and Faster than BFS to solve this problem.

- Also, A* algorithm has many advantages:
  1- When compared to other search algorithms, the A* search algorithm is the best.
  2- The A* search method is ideal and comprehensive.
  3- This method can resolve extremely difficult situations.
  4- Don't add to the cost of already-expensive paths.
  5-Concentrate on promising paths.

- Also, we used pseudocode to describe this algorithm.

- Some uses of A*:

find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

6. References

https://www.linkedin.com/pulse/solving-8-puzzle-using-algorithm-python-ajinkya-sonawane/

https://www.javatpoint.com/ai-informed-search-algorithms

https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-aima-24

7. Appendix

```
class Node:
   def _init_(self,data,level,fval):
      """ Initialize the node with the data, level of the node and the
calculated fvalue """
      self.data = data
      self.level = level
      self.fval = fval
   def generate_child(self):
      """ Generate child nodes from the given node by moving the blank
space
         either in the four directions {up,down,left,right} """
      x,y = self.find(self.data,'_')
      """ val_list contains position values for moving the blank space in
either of
         the 4 directions [up,down,left,right] respectively. """
      val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
      children = []
      for i in val_list:
         child = self.shuffle(self.data,x,y,i[0],i[1])
         if child is not None:
            child_node = Node(child,self.level+1,0)
            children.append(child_node)
      return children

   def shuffle(self,puz,x1,y1,x2,y2):
      """ Move the blank space in the given direction and if the position
value are out
         of limits the return None """
      if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
```

```
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None
    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j
class Puzzle:
    def _init_(self,size):
        """ Initialize the puzzle size by the specified size,open and closed
lists to empty """
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
        """ Accepts the puzzle from the user """
```

```
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x)
"""
        return self.h(start.data,goal)+start.level
    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp
    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()
        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
```

```
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have
reached the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]
            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)
puz.process()
```