Solve 8-Puzzle Problem With Heuristic Search(A*)

Faculty of Computers & Information

Artificial Intelligence Presentation.

Project Team Member:

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

1-Problem

1- 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 squares into a numerical order of 1 to 8 and last square as blank. Each of the squares adjacent to blank block can move up, down, left or right depending on the edges of the matrix.

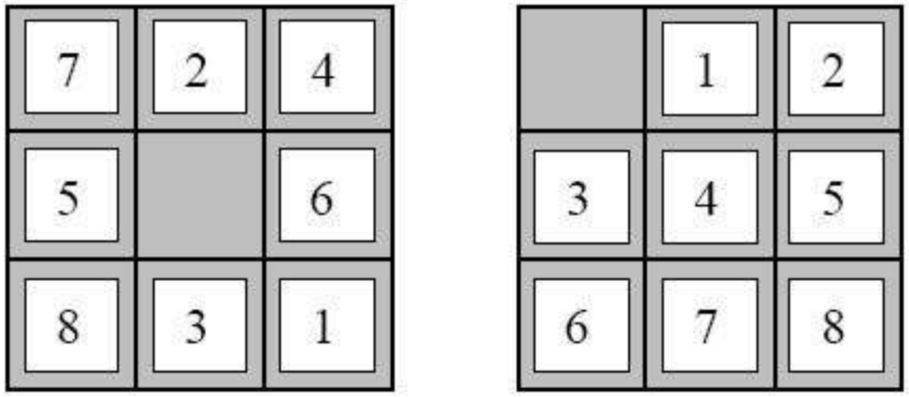
-A* Search is a Computer Algorithm that is Widely used in path Finding and Graph Traversal, the Process of Plotting an Efficiently traversable path between multiple points; called Nodes. Noted for it's performance and accuracy , it enjoys widespread use.

The key feature of the A* algorithm is that it keeps a track of each Visited node which helps in ignoring the nodes that are already visited; saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored ,and it chooses the most optimal node from this list, Thus saving time .not exploring unnecessary or less optimal nodes...

Main idea of A*:

– avoid expanding paths that are already expensive .

– focus on paths that show promise .



Start State Goal State

A*

-In this Problem we need to Implement a solution to 8-puzzle, by using A*

Methodology of Solving A* search

-Keep Track of Visited States. The Heuristic for A* will be Hamming priority function ; Which can be defined as The number of Blocks in the Wrong Position, plus the number of moves made so far to get to the Search node. Intuitively, a search node with a Small Number of Blocks in the Wrong Position is close to the Goal, And we prefer a search node that has been Reached using a Small Number of Moves. The solution should Print All Sequence of moves

that you apply to Reach to a Goal State.

-We are also calculate g(n) which is a measured of step cost for each move made from Current State to Next State; initially it's set to Zero. For each of the Heuristic we have implemented

$f(n)=g(n)+h(n)$;

Where g(n) is Step Cost ,and h(n) is the Heuristic Function used. Each of the States is Explored using Priority Queue, Which Stores the Position and f(n) value as key Value Pair.

Then using Merge-Sort Technique Priority Queue is Sorted ,and the next node to be explored is selected.
based on the least f(n) value

-In this project, we have used the optimality of A* by not Allowing any node Generation of Previously Traversed Nodes. We have used Two sets they are Closed set and Open set to Implement
Functionality of A* algorithm. Closed set stores all the previously expanded nodes and open set (priority queue in source code) which stores all of the non-duplicate nodes and sorts them according to f(n) value.
The reason for this project to be unique is the functionality it offers to user to enter initial state and goal state, thus offering the generalized solution for any initial state against any goal state.
The program comes to an end if the A* algorithm has not found an optimal path within a runtime limit of two minutes.

- The A* algorithm's main characteristic
is that it keeps 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 has 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..

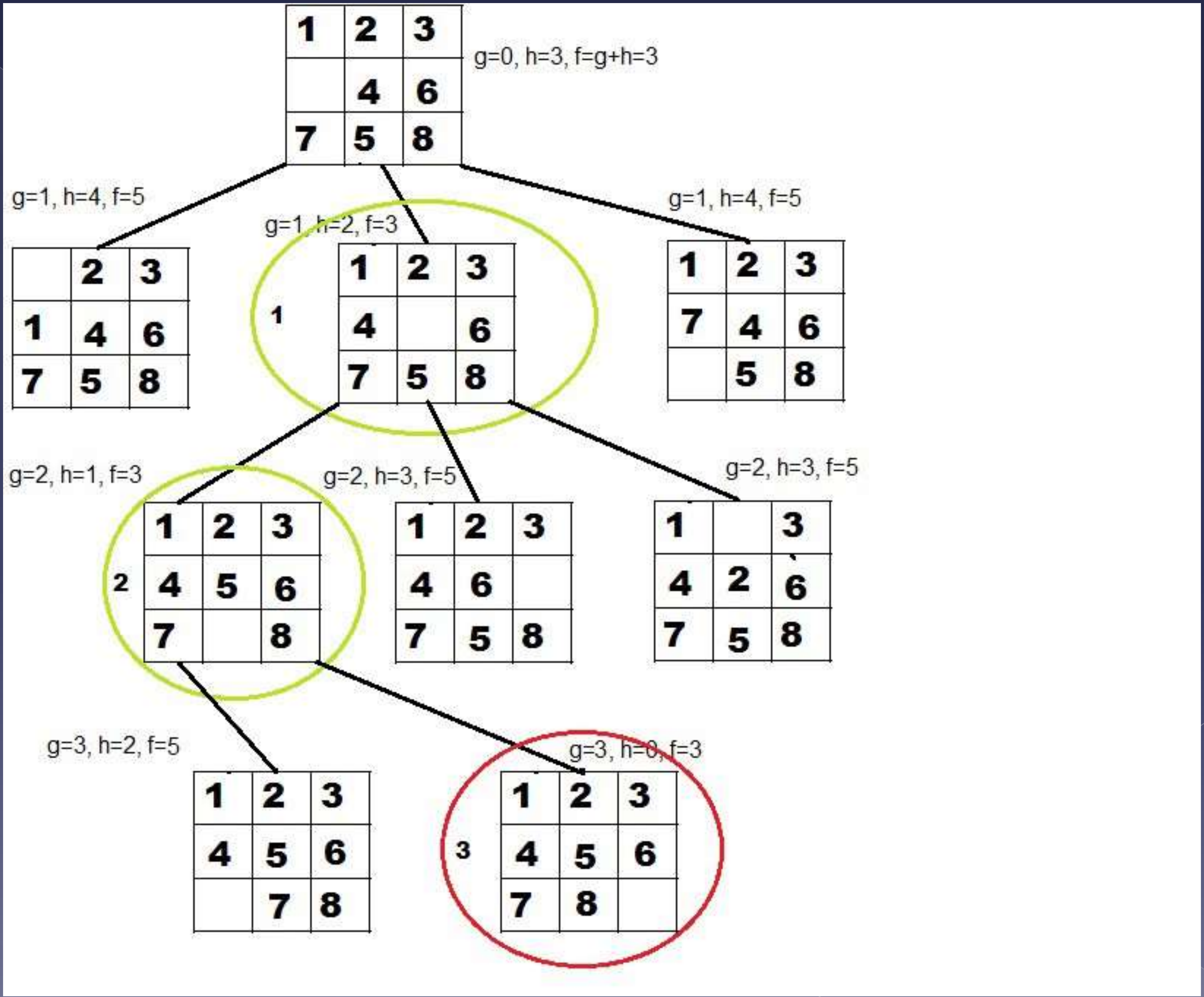
A* has the following Advantages :

- When compared to other search algorithms, the A* search algorithm is the best.
- The A* search method is ideal and comprehensive.
- This method is capable of resolving extremely difficult situations.
- Don't add to the cost of already-expensive paths.
- Concentrate on promising paths

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*.
- 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.

Methodology of Solving A* search



Programming Language used

We used Python Programming Language to solve our Problem.

-At the beginning of code we have to

Define all variable we use and the node

With data and calculate fvalue .

- Function of def generate_child :

enetrate child nodes from the given

node by moving the blank space
either in the four directions {up ,down ,left , right}

- **val_list** contains position values for moving

the blank space in either of
the 4 directions [up ,down ,left , right]

 respectively.

- Function def shuffle:

Move the blank space in the given direction

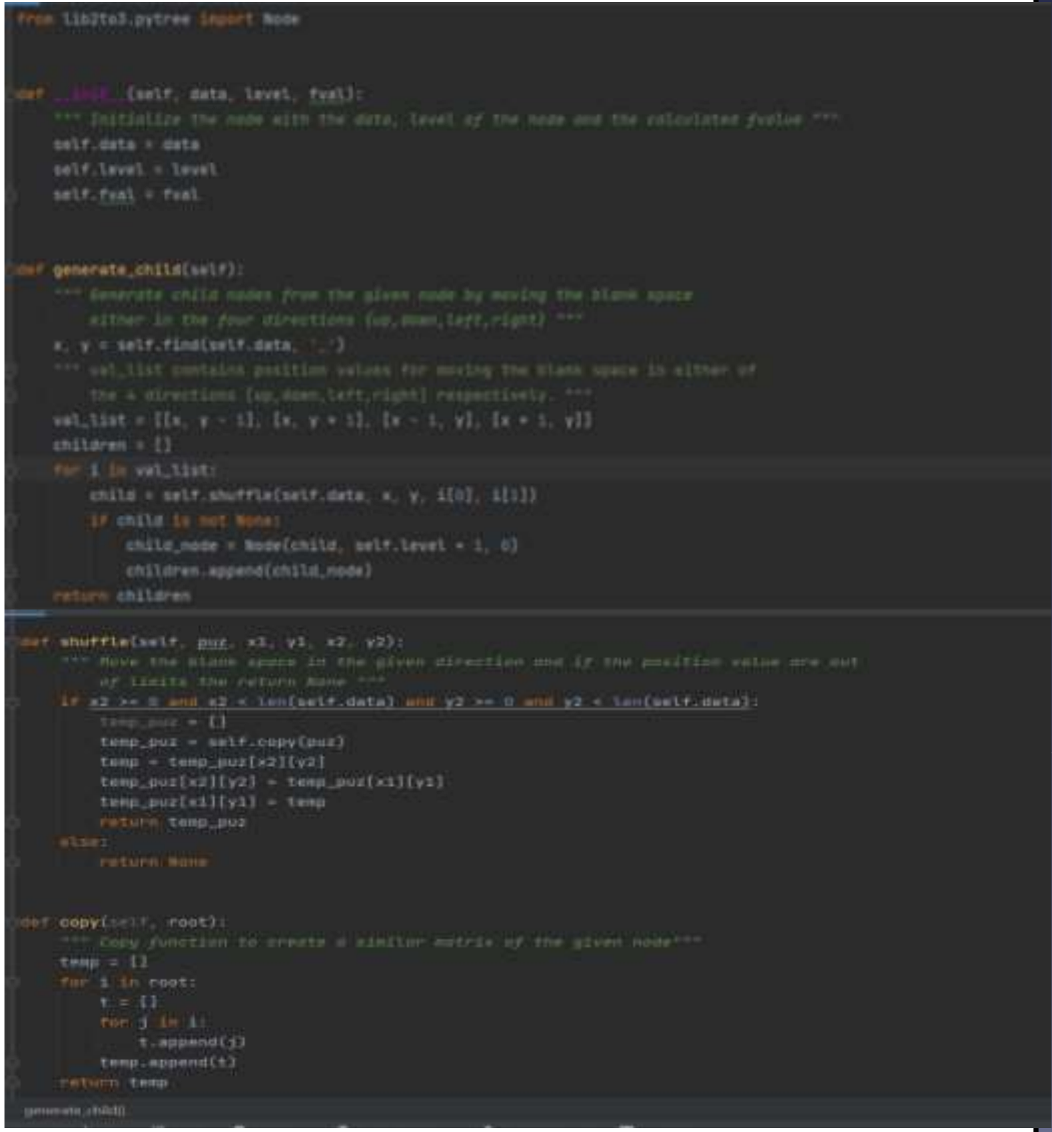
and if the position value are out

of limits the return None.

-Function def copy :

Copy function to create a similar matrix of

the given node.



```
from lib2to3.pytree import 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
```

Programming Language used

We used Python Programming Language to solve our Problem.

-First we define class Puzzle:

Which we can specified puzzle size ,

And open and close list to be empty.

-if puzzle isn’t empty it will be accepted.

-Function f() :is a Heuristic function to

 calculate heuristic value :

F(x)=h(x)+g(x).

-Function h() :calculate difference

Between given puzzle.

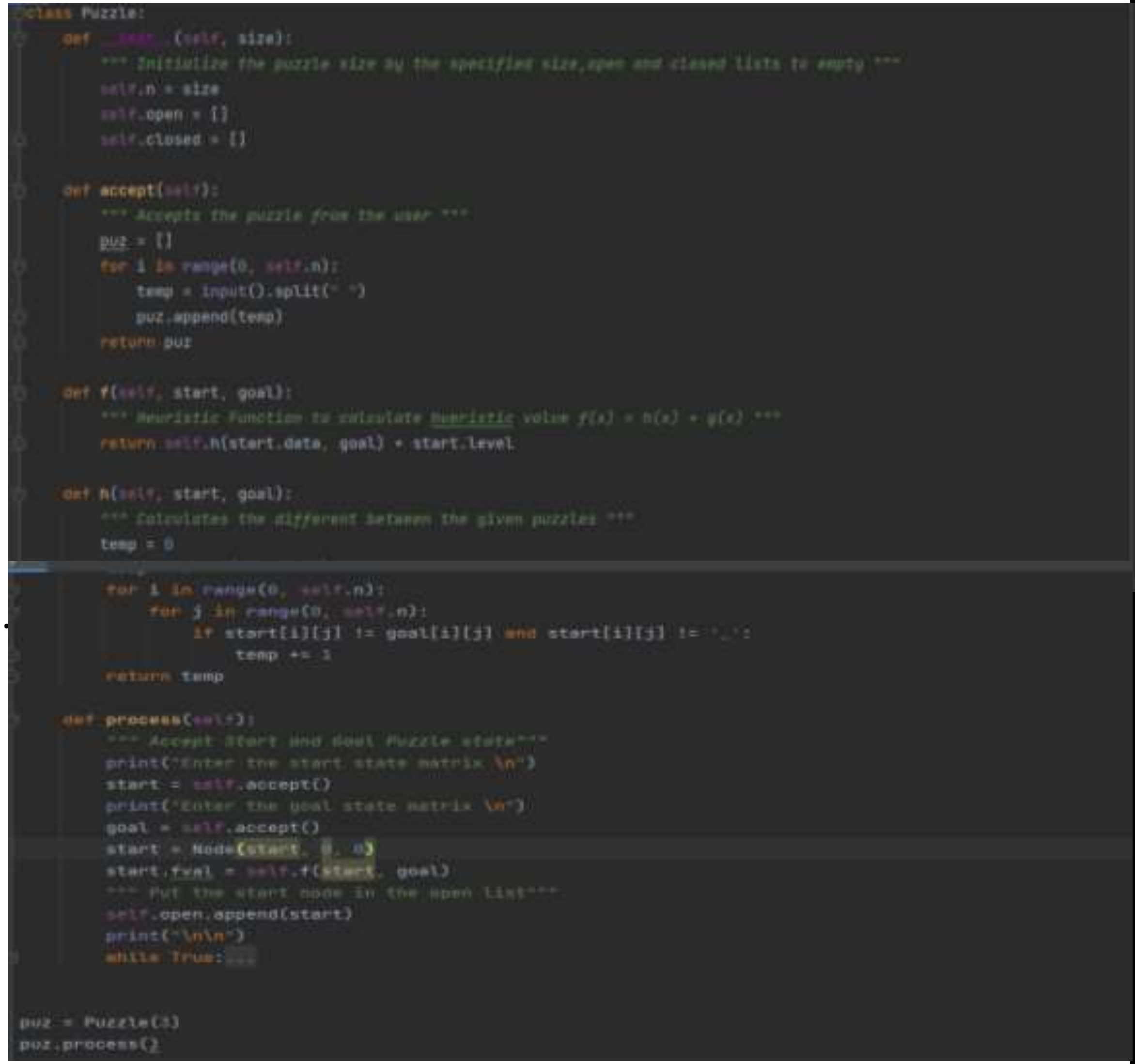
-And Process Function :

        to accept Start and Goal puzzle state.

And this function responsible for print

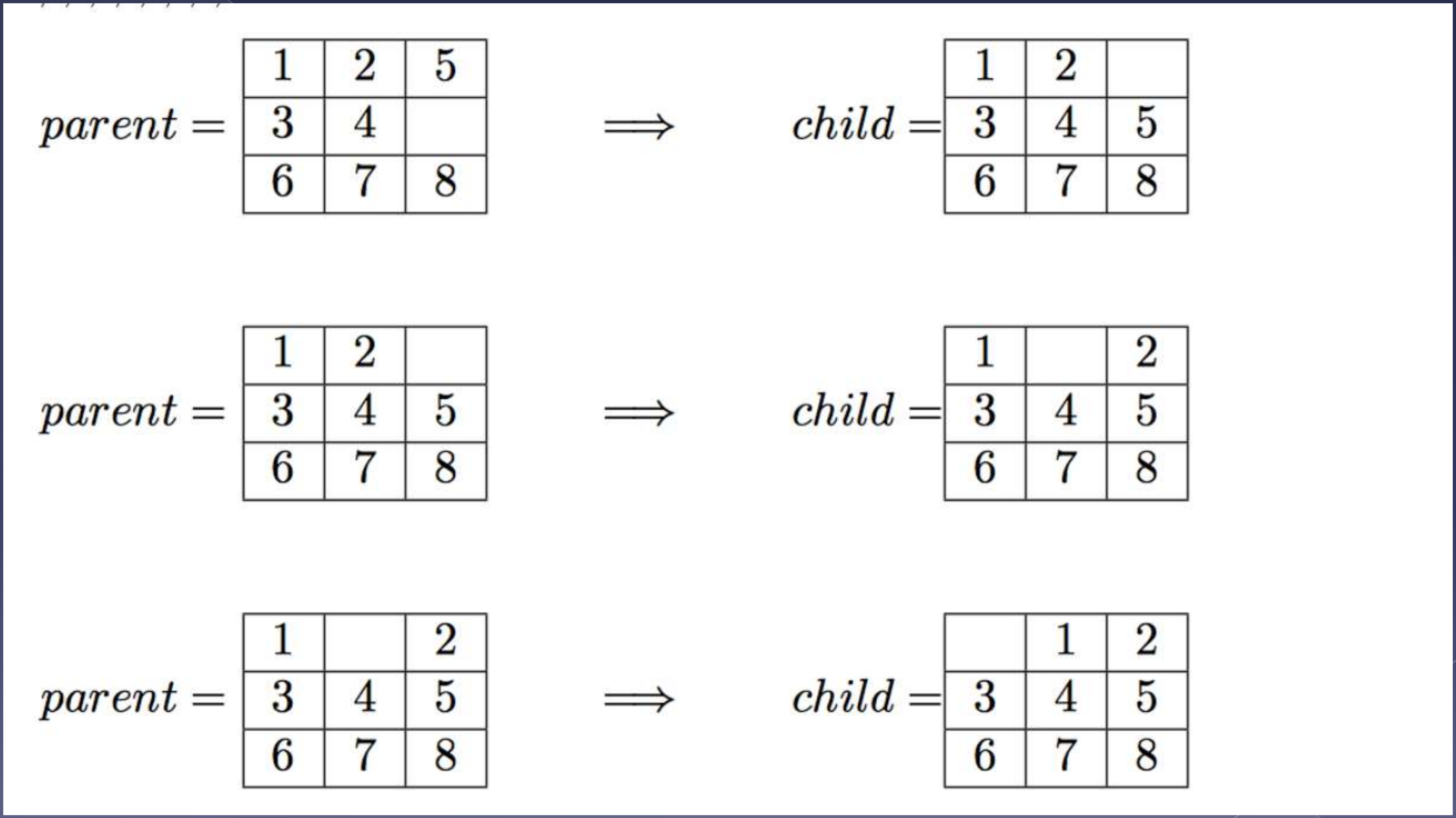
In the run screen “what the user have to

Enter in each step.”



```
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:...


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

The steps of run code



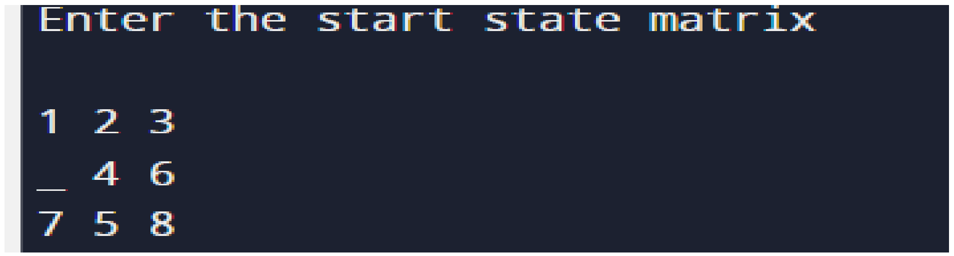
$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline\end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array}$$

-First step when run. -

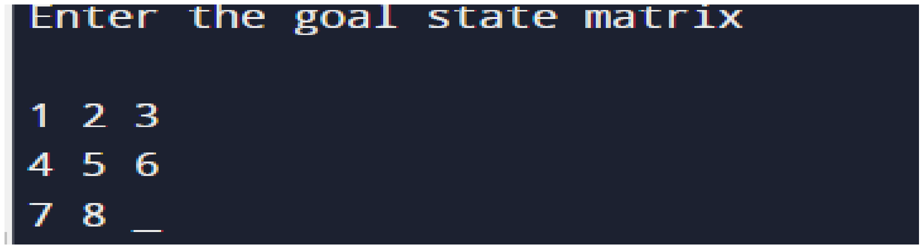
-The code when run will show this message to -
let user to enter his matrix 3*3 as input . -



```
Enter the start state matrix

1 2 3
_ 4 6
7 5 8
```

-Second step.

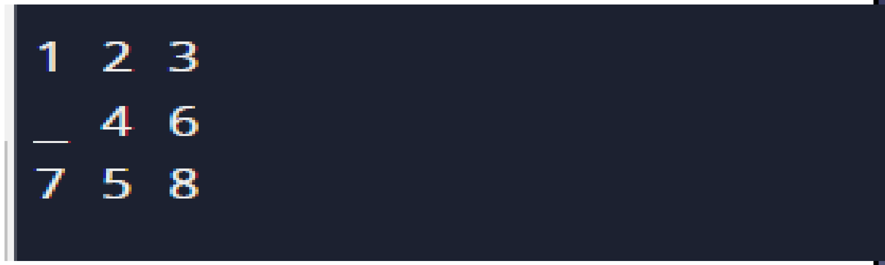
After the user enters the start state matrix this is the sentence appear as shown in the figure to requirement from user enter the goal state matrix to solve this problem.



```
Enter the goal state matrix

1 2 3
4 5 6
7 8 _
```

-The First event:

This is start state matrix
after when the user enters the start and goal state matrix.

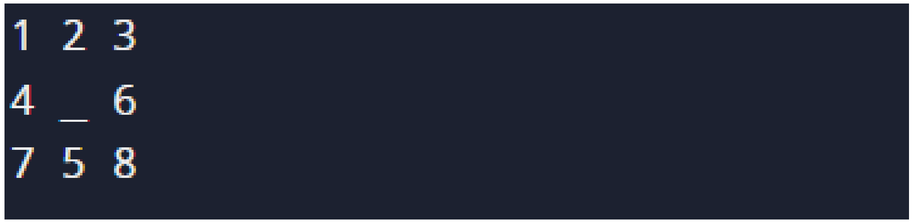


```
1 2 3
_ 4 6
7 5 8
```

.

- The second event: -

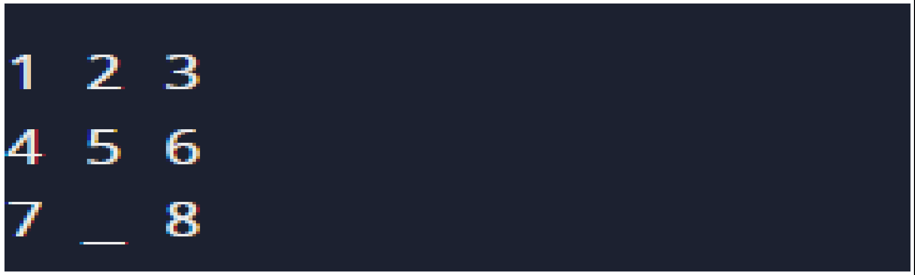
This is the first step to solving a problem 8-puzzle where the program solves with A* algorithm method. Number 4 move to left replace the blank space.



```
1 2 3
4 _ 6
7 5 8
```

-The third event: -

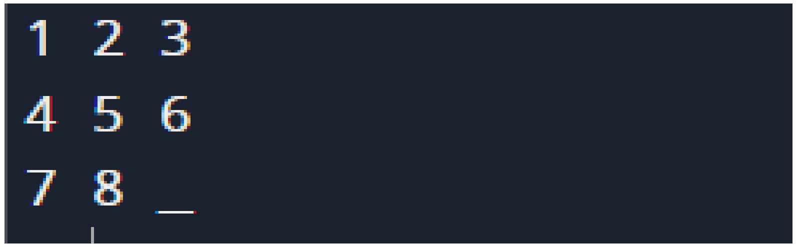
This is the second step to solving a problem 8-puzzle. Number 5 move to up replace the blank space who left her number 4.



```
1 2 3
4 5 6
7 _ 8
```

-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.



```
1 2 3
4 5 6
7 8 _
```

Thank You