8-12-23

8-Puzzle problem using A* algorithm

- Finds the most cost-effective path to reach the final state from initial state.

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

$g(n) \rightarrow$ depth of node

$h(n) \Rightarrow$ no. of misplaced tiles.

Suppose:

2	8	3
1	6	4
7		5

Initial state

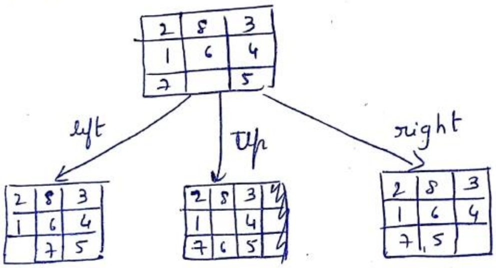
1	2	3
8		4
7	6	5

Final state

g = 0

h = 4

f = 0 + 4.



g = 1
h = 5
f = 1+5=6

g = 1
h = 3.
$\boxed{f = 1+3 = 4}$
(least).

g = 1
h = 5
f = 1+5 = 6.

Code

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \\ 2 & 7 & 8 & 9 \end{array}$$

```
class Node:
    def _init_(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):   # to generate child nodes from the given
                                  node by moving blank space either
                                  in the 4 directions.

        # x,y = self.find(self.data,'_') →#func^n to find position of
                                                blank space
        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 & if the pos. value
        are out of limits, then return None"""
        if x2>=0 and x2 < len(self.data) and y2>=0 &
                                  y2 < len(self.data):
                                                  # checks if new
                                                    position (x2,y2)
                                                    is within the
                                                    bounds of
                                                    puzzle.
            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):
        temp = []
        for i in root:
            t=[]
            for j in i:
                t.append(j)
            temp.append(t)
        return temp
```

Two lists are maintained.

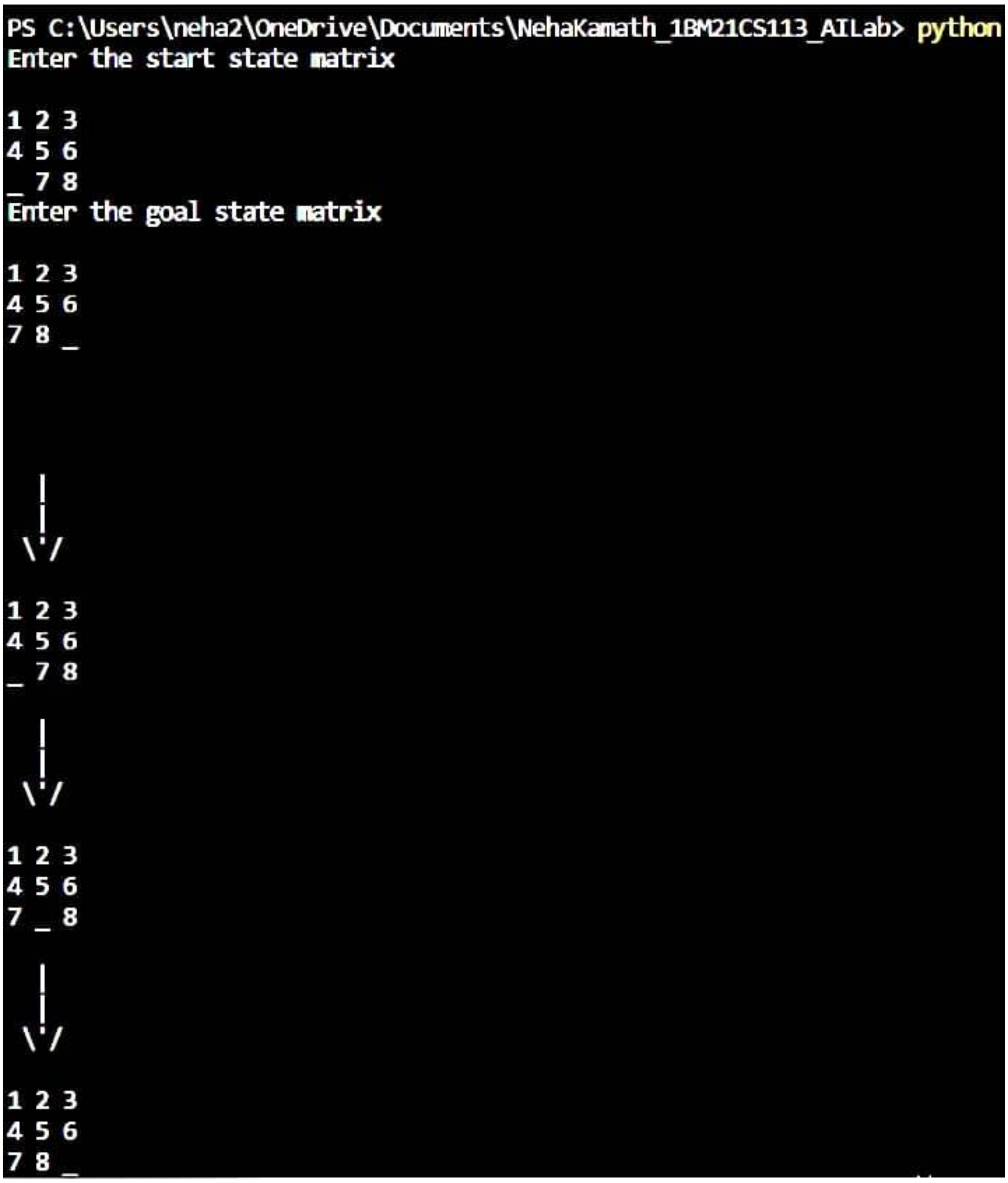
open list → contains all the nodes that are generated & not existing in closed list

closed list – each node ~~explored~~ explored after its neighbouring nodes are discovered

So after expanding a node, it is pushed into the closed ~~stack~~ list and the newly generated states are pushed in open list.

```
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 &
    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 func to calculate heuristic value
        f(x) = h(x) + g(x) """
   return self.h(start, data, goal) + start.level

  def h(self, start, goal):
   """ Calculates the difference b/w 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 & goal puzzle state """
    print("Enter start state matrix :\n")
    start = self.accept()
    print("Enter 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(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 open list based on f value """
     self.open.sort(key = lambda x : x.fval, reverse = False)

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



```
PS C:\Users\neha2\OneDrive\Documents\NehaKamath_1BM21CS113_AILab> python
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _




  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

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