

AIM OF THE EXPERIMENT: WRITE A PROGRAM IN PYTHON TO SOLVE ANY ONE AI PROBLEM

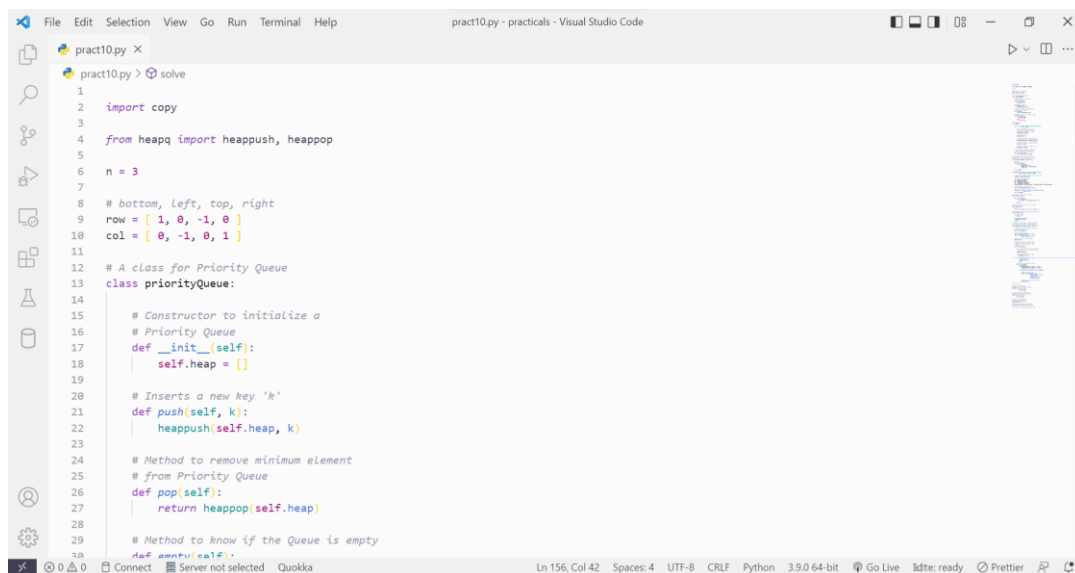
THEORY:

8 Puzzle Problem:

Given a 3x3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.

SOURCE CODE:

Below is the code of 8 Puzzle Problem

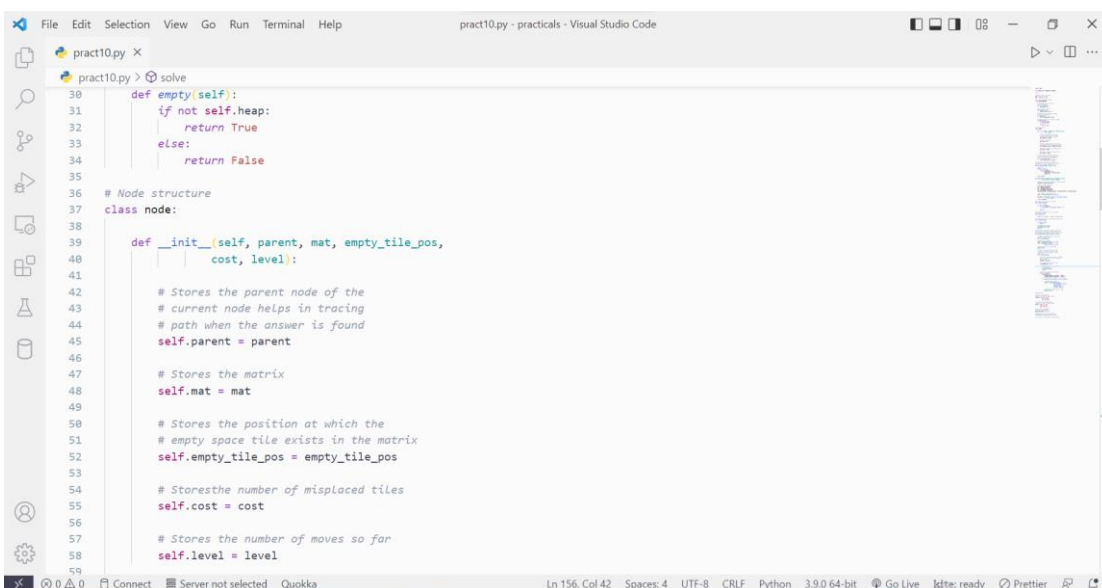


```

pract10.py X
pract10.py > solve
1
2 import copy
3
4 from heapq import heappush, heappop
5
6 n = 3
7
8 # bottom, left, top, right
9 row = [ 1, 0, -1, 0 ]
10 col = [ 0, -1, 0, 1 ]
11
12 # A class for Priority Queue
13 class PriorityQueue:
14
15     # Constructor to initialize a
16     # Priority Queue
17     def __init__(self):
18         self.heap = []
19
20     # Inserts a new key 'k'
21     def push(self, k):
22         heappush(self.heap, k)
23
24     # Method to remove minimum element
25     # from Priority Queue
26     def pop(self):
27         return heappop(self.heap)
28
29     # Method to know if the Queue is empty
30     def empty(self):

```

Fig 10.1 Source Code

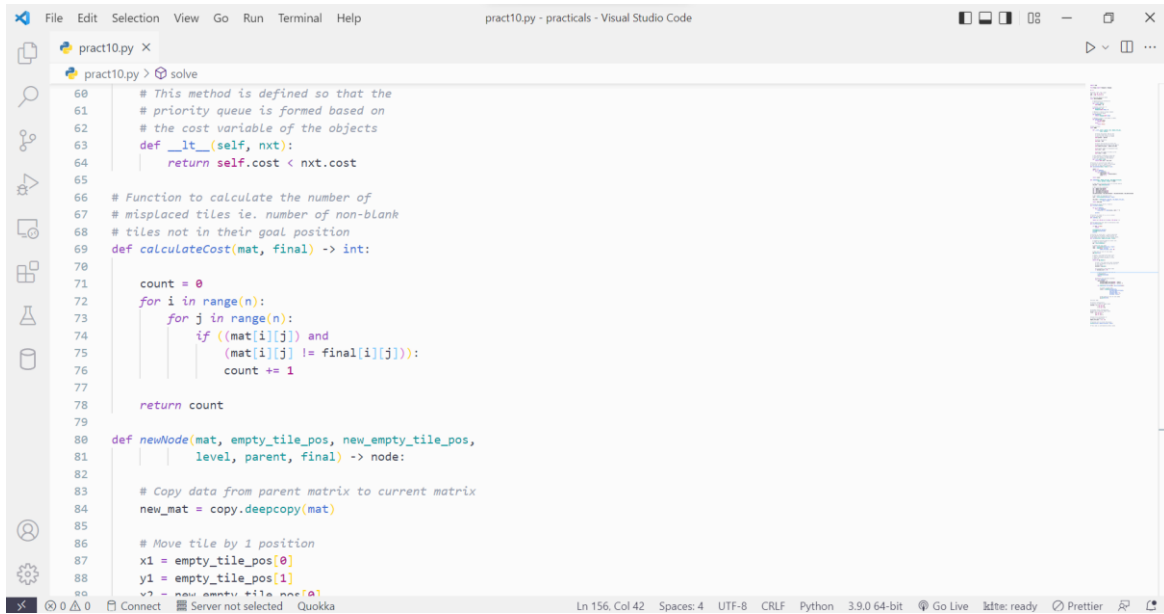


```

pract10.py X
pract10.py > solve
30 def empty(self):
31     if not self.heap:
32         return True
33     else:
34         return False
35
36 # Node structure
37 class node:
38
39     def __init__(self, parent, mat, empty_tile_pos,
40                 cost, level):
41
42         # Stores the parent node of the
43         # current node helps in tracing
44         # path when the answer is found
45         self.parent = parent
46
47         # Stores the matrix
48         self.mat = mat
49
50         # Stores the position at which the
51         # empty space tile exists in the matrix
52         self.empty_tile_pos = empty_tile_pos
53
54         # Stores the number of misplaced tiles
55         self.cost = cost
56
57         # Stores the number of moves so far
58         self.level = level
59

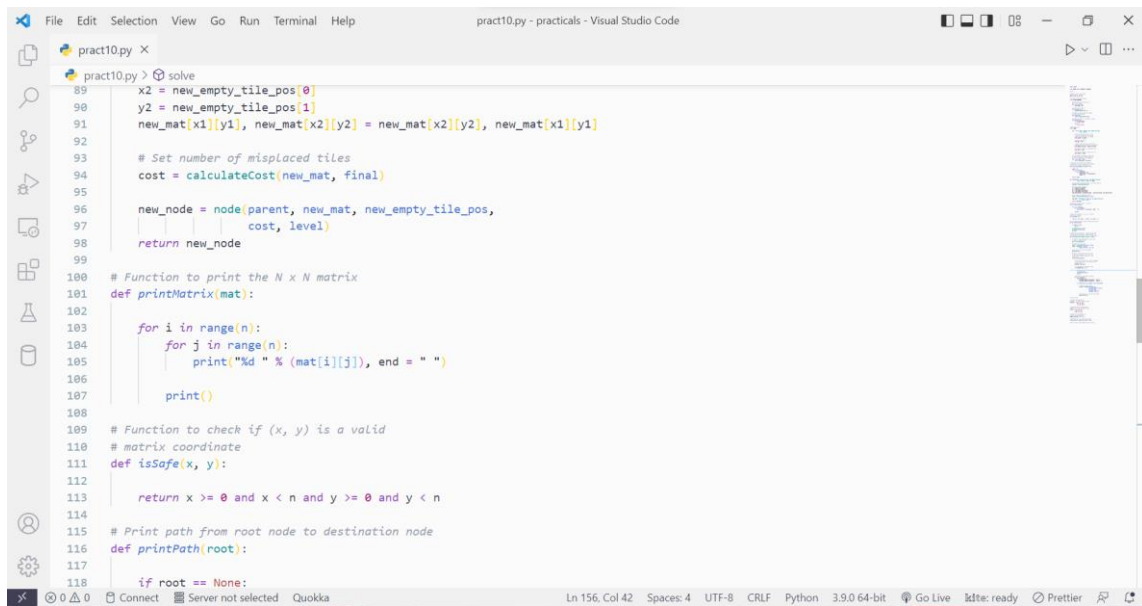
```

Fig 10.2 Source Code



```
pract10.py > solve
60 # This method is defined so that the
61 # priority queue is formed based on
62 # the cost variable of the objects
63 def __lt__(self, nxt):
64     return self.cost < nxt.cost
65
66 # Function to calculate the number of
67 # misplaced tiles ie. number of non-blank
68 # tiles not in their goal position
69 def calculateCost(mat, final) -> int:
70
71     count = 0
72     for i in range(n):
73         for j in range(n):
74             if ((mat[i][j] and
75                 (mat[i][j] != final[i][j]))):
76                 count += 1
77
78     return count
79
80 def newNode(mat, empty_tile_pos, new_empty_tile_pos,
81             level, parent, final) -> node:
82
83     # Copy data from parent matrix to current matrix
84     new_mat = copy.deepcopy(mat)
85
86     # Move tile by 1 position
87     x1 = empty_tile_pos[0]
88     y1 = empty_tile_pos[1]
89     x2 = new_empty_tile_pos[0]
90     y2 = new_empty_tile_pos[1]
91     new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
92
93     # Set number of misplaced tiles
94     cost = calculateCost(new_mat, final)
95
96     new_node = node(parent, new_mat, new_empty_tile_pos,
97                    cost, level)
98     return new_node
99
100 # Function to print the N x N matrix
101 def printMatrix(mat):
102
103     for i in range(n):
104         for j in range(n):
105             print("%d " % (mat[i][j]), end = " ")
106
107     print()
108
109 # Function to check if (x, y) is a valid
110 # matrix coordinate
111 def isSafe(x, y):
112
113     return x >= 0 and x < n and y >= 0 and y < n
114
115 # Print path from root node to destination node
116 def printPath(root):
117
118     if root == None:
```

Fig 10.3 Source Code



```
pract10.py > solve
89     x2 = new_empty_tile_pos[0]
90     y2 = new_empty_tile_pos[1]
91     new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
92
93     # Set number of misplaced tiles
94     cost = calculateCost(new_mat, final)
95
96     new_node = node(parent, new_mat, new_empty_tile_pos,
97                    cost, level)
98     return new_node
99
100 # Function to print the N x N matrix
101 def printMatrix(mat):
102
103     for i in range(n):
104         for j in range(n):
105             print("%d " % (mat[i][j]), end = " ")
106
107     print()
108
109 # Function to check if (x, y) is a valid
110 # matrix coordinate
111 def isSafe(x, y):
112
113     return x >= 0 and x < n and y >= 0 and y < n
114
115 # Print path from root node to destination node
116 def printPath(root):
117
118     if root == None:
```

Fig 10.4 Source Code

```
pract10.py > solve
118     if root == None:
119         return
120
121     printPath(root.parent)
122     printMatrix(root.mat)
123     print()
124
125     # Function to solve N*N - 1 puzzle algorithm
126     # using Branch and Bound. empty_tile_pos is
127     # the blank tile position in the initial state.
128     def solve(initial, empty_tile_pos, final):
129
130         # Create a priority queue to store live
131         # nodes of search tree
132         pq = PriorityQueue()
133
134         # Create the root node
135         cost = calculateCost(initial, final)
136         root = node(None, initial,
137                     empty_tile_pos, cost, 0)
138
139         # Add root to List of Live nodes
140         pq.push(root)
141
142         # Finds a Live node with Least cost,
143         # add its children to List of Live
144         # nodes and finally deletes it from
145         # the List.
146         while not pq.empty():
147
```

Fig 10.5 Source Code

```
pract10.py > solve
148     # Find a Live node with Least estimated
149     # cost and delete it from the List of
150     # Live nodes
151     minimum = pq.pop()
152
153     # If minimum is the answer node
154     if minimum.cost == 0:
155
156         # Print the path from root to
157         # destination;
158         printPath(minimum)
159         return
160
161     # Generate all possible children
162     for i in range(n):
163         new_tile_pos = [
164             minimum.empty_tile_pos[0] + row[i],
165             minimum.empty_tile_pos[1] + col[i], ]
166
167         if isSafe(new_tile_pos[0], new_tile_pos[1]):
168
169             # Create a child node
170             child = newNode(minimum.mat,
171                             minimum.empty_tile_pos,
172                             new_tile_pos,
173                             minimum.level + 1,
174                             minimum, final,)
175
176             # Add child to List of Live nodes
```

Fig 10.6 Source Code

```
pract10.py > solve
175
176     # Add child to List of Live nodes
177     pq.push(child)
178
179 # Driver Code
180
181 # Initial configuration
182 # Value 0 is used for empty space
183 initial = [ [ 1, 2, 3 ],
184             [ 5, 6, 0 ],
185             [ 7, 8, 4 ] ]
186
187 # Solvable Final configuration
188 # Value 0 is used for empty space
189 final = [ [ 1, 2, 3 ],
190           [ 5, 8, 6 ],
191           [ 0, 7, 4 ] ]
192
193 # Blank tile coordinates in
194 # initial configuration
195 empty_tile_pos = [ 1, 2 ]
196
197 # Function call to solve the puzzle
198 solve(initial, empty_tile_pos, final)
199
200 # This code is contributed by Kevin Joshi
201
```

Fig 10.7 Source Code

OUTPUT:

```
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

D:\B.Tech\6th semester\AI lab\practicals>python -u "d:\B.Tech\6th semester\AI lab\practicals\pract10.py"
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

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

1 2 3
5 8 6
0 7 4

D:\B.Tech\6th semester\AI lab\practicals>
```

Fig 10.8 Output