Date
Page

# 8-puzzle

Puzzle state = [0, 1, 2]
[3, 4, 5]
[6, 7, 8]

def manhattan(state, final)

1. Define the goal state in a 3x3 matrix
2. Function to find blank space

```
for i in range
  for j in range
    if state[i][j] == 0
      return [i][j]
```

Once blank tile is found we move one of the four directions i.e up, down, left, Right.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 5 | 7 | | 4 | 5 | 7 |
| 8 | 0 | 6 | → | 8 | 6 | 0 |
| 3 | 1 | 2 | | 3 | 1 | 2 |

this state is added to stack. The blank space is moved in other directions i.e up, down, left, right.

Date __/__/__
Page _____

Converting each state into a node after it is marked as visited. the node is popped and it becomes the current state.
Each valid neighbour is pushed into the stack

```
neighbour = get_neighbour(current)

for neighbour in neighbours
    if neighbour is not visited
    stack.append(neighbour)
```

stack = [ ]
Node(up)
Node(down)
Node('left')
Node(Right)
[ ]

node (right) is popped
and it is explored

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

→

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

1 2 3
4 0 6
→ 2 5 8

→

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

Added to stack & LIFO is checked

Date / /
Page

```
class Node
    def __init__(self, state, parent=None, move=None, depth=0)
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth

def goal_state(state)
    return state == [[1,2,3],
                     [4,5,6],
                     [7,8,0]]

def find_blank_tile(state)
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return (i, j)

def neighbours(node)
    state = node.state
    row, col = find_blank_tile(state)
    neighbours = []

    moves = {'up': (row-1, col),
             'down': (row+1, col),
             'left': (row, col-1),
             'right': (row, col+1)
    }
```

Date / /
Page

class Node:
def __init__(self, state, parent, move, depth):
self.state = state
self.parent = parent
self.move = move
self.depth = depth

for move, (new_row, new_col) in moves.items():
new_state = [row[:] for row in state]
new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[row][col]
neighbours.append(Node(new_state, node, move, node.depth + 1))
return neighbours

def dfs_limit(start_state, depth_limit):
stack = [Node(start_state)]
visited = set()

while stack:
current_node = stack.pop()

if is_goal(current_node.state):
return reconstruct_path(current_node)

visited.add(tuple(map(tuple, current_node.state)))

if current_node.depth < depth_limit:
neighbours = get_neighbours(current_node)
for neighbour in neighbours:
if tuple(map(tuple, neighbour.state)) not in visited:
stack.append(neighbour)
return None

def reconstruct_path(node):
path = []
while node.parent is not None:
path.append(node.move)
node = node.parent
return path[::-1]

Date
Page

```
initial_state = [ [1, 2, 3]
                  [4, 0, 6]
                  [7, 5, 8] ]
depth_limit = 10
solution = dfs_limit(initial_state, depth_limit)
```

Output

```
Solution : ['right', 'down', 'left', 'up', 'right', 'down',
            'left', 'up', 'right', 'down']
```