08-10-24

Algorithm for 8 Puzzle using dfs

- Considering 3×3 {2D} grid
  initialize the problem
- Define the initial state of puzzle (3×3 grid)
- Define goal state
- Randomly arranged list of range [1-8]
- Define 1 empty blank / space as '0', called titles

Step 2:- Define the DFS Function
Defining the function of DFS heady
input:- current state, goal state, path, visited.
- current state is equal to goal state:
  return path taken to reach goal
- Add current state to visited state

3) Generating possible moves:-
find position of blank (0) space & generate all possible new states by sliding adjacent tiles.

4) Iterate through possible moves
for each next-state generated:
if next-state has not been visited
recursively call DFS function with next_state, goal state, updated path
- if a solution is found (return value is not None):
  return that solution

Date / /
Page

5) Backtrack:
If no solution are found after exploring all boxes from current state, remove current state from visited set & back track indicate no solution in this path.

Initial state

2	3	5
7	1	4
0	8	6

Our goal

1	2	3
4	5	6
7	8	0

Solving 8 puzzle using Manhattan Distance

① Initialize the problem
- Define the initial state of puzzle (3×3 grid)
- Define goal state
- Randomly arrange the list of range (1 to 8)
- Define 1 Empty block as '0' called tile.

② Defining Heuristic Function:
Create a Function manhatten (state):
Initialize variable distance to 0
for each title set in state
calculate target position based on tile value.
Add absolute difference in row & column indices to distance and return distance

Date __/__/__
Page _____

2) input :- Current state, goal state, g-cost visited, path
if current state is equal to goal state
Return path.

3) Generating Possible moves

* Calculate g-cost as g.cost + 1
Calculate heuristic h-cost using manhatten-distance (next-state)
calculate total cost f-cost = g-cost + h-cost.
if next state not visited
Add (f-cost, next-state, updated state) to priority queue

4) Process of priority queue
while priority queue not empty
pop state lowest total cost (f(n))
mark it as visited
call the function recursively with state & updated costs

5) Backtrack
if no solution occurs return to previous position

Initial state

2	0	1
5	4	6
7	8	3

1	2	3
4	5	6
7	8	0

Date / /
Page

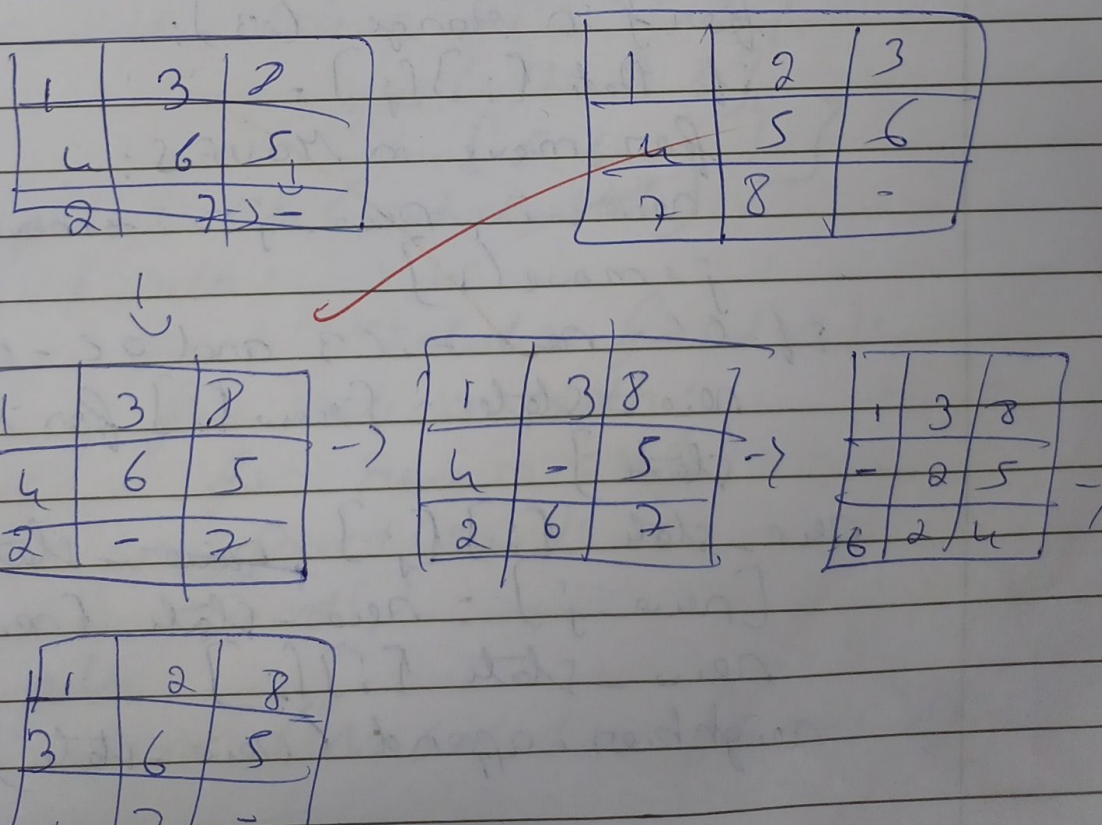
1	0	0	Priority → Pop state from queue with lowest cost, return path, mark as visited
2	0	1	
4	0	1	
3	1	0	
5	1	1	
6	0	0	
7	0	0	
8	0	1	

Code:-

```
def DFS_8 Puzzle (initialize_state, goal_state)
    stack = [(initial_state, 0)]
    visited = set()
    while stack:
        current_state, depth = stack.pop()
    if current_state == goal state
        return goal state achieved

def find_blank tile (state):
        return state.index(0)
def swap (state, i, j)
        new_state = state[:]
    new state[i], new state[j] = merge
    return new state;

def get_neighbours (state)
        neighbours = []
        index = find_blank_tile (state)
        row, col, index;
```

```
moves = ((-1,0), (1,0), (0,1), (0,-1))
in moves:
new_row, new_col = row + move[0], col + move[1]
if 0 <= new_row < 3 and 0 <= new_col
new_state = (state, index, new_state)
neighbours = append (new_state)
if current state == goal_state
    return path + (current state)
if (tuple neighbour) not in visited:
    stack.append ((neighbour, path + [current])
if solution != solution:
    print_puzzle (state)
else
    print (solution)
```

Output:



Manhatten

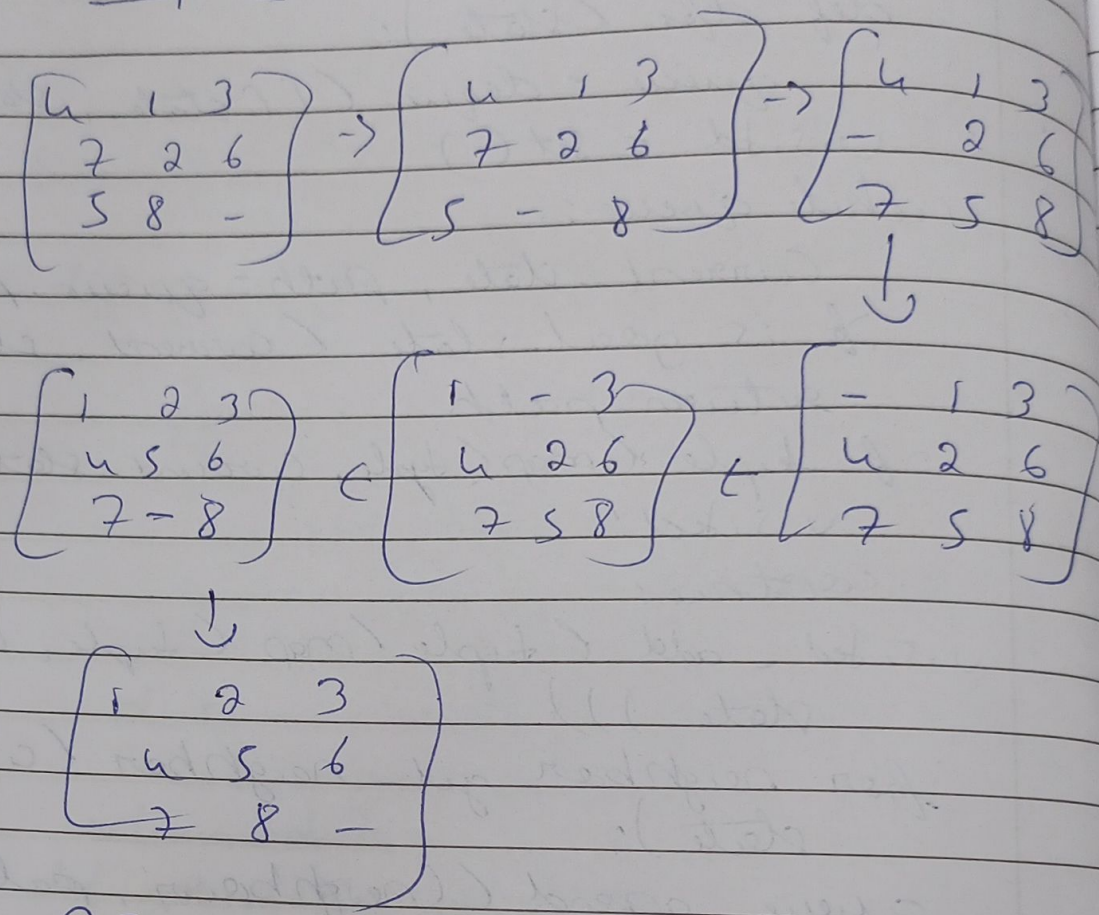
```
from collections import deque
GOAL_STATE = [[1, 2, 3], [4, 5, 6],
              [7, 8, '-']]
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def manhatten_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != '-':
                goal_i, goal_j = divmod(state
                [i][j] - 1, 3)
                distance += abs(i - goal_i) + abs
                (j - goal_j)
    return distance
def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == '-':
                for move in MOVES:
                    new_i, new_j = i + move[0],
                    j + move[1]
                    if 0 <= new_i < 3 and 0 <= new_j < 3:
                        new_state = [row[:] for row in
                        state]
                        new_state[i][j], new_state[new_i]
                        [new_j] = new_state[new_i][new_j],
                        new_state[i][j]
                        neighbors.append(new_state)
```

Date __/__/__
Page _____

```
return neighbors
def dfs(state):
    queue = deque([[state, state]])
visited = set()
while queue:
    current_state, path = queue.popleft()
if is_goal_state(current_state):
    return path
if tuple(map(tuple, current_state)) in
    visited:
    continue
visited.add(tuple(map(tuple, current_
    state)))
for neighbor get_neighbor(current_
    state):
    queue.append((neighbour, path +
    [neighbour]))
return none
initial_state = [[4, 1, 3],
                 [3, 2, 6],
                 [5, 8, :]]
path = dfs(initial_state)
if path:
    print("Solution found:");
    for state in path:
        for row in state:
            print(row);
        print()
else:
    print("No solution found");
```

Date
Page

Output



DFS:-

```
class puzzle state:
def __init__(self, board, zero_position,
    moves=[]):
    self.board = board
    self.zero_position = zero_position
    self.moves = moves
def is_goal(self):
    return self.board == [[1,2,3],[4,5,6],
        [7,8,0]]
def get_possible_moves(self):
    row, col = self.zero_position
    possible_moves = []
    direction = [(-1,0),(1,0),(0,-1),
        (0,1)]
```

```
for dx, dc in direction :
    new_row, new_col = row + dx,
    col + dc
    if 0 <= new_row < 3 and 0 <= new_col < 3:
    possible_moves.append((new_row,
    new_col))
    return possible_moves
def move(self, new_zero_position):
    new_row, new_col = new_zero_position
    row, col = self.zero_position
    new_board = [x[:] for x in self.
    board]
    new_board[row][col], new_board
    [new_row]
    [new_col] = new_board[new_row]
    [new_col],
    new_board[row][col]
    return PuzzleState(new_board,
    new_zero_position, self.moves +
    [(new_row, new_col)])
def dfs(puzzle_state, visited):
    if puzzle_state.is_goal():
        return puzzle_state.moves
    visited.add(tuple(map(tuple,
    puzzle_state.board)))
for new_zero_position in puzzle_state.
get_possible_moves():
    new_state = puzzle_state.move(new_
    zero_position)
    if tuple(map(tuple, new_state.
    board)) 
not in visited:
    result = dfs(new_state, visited))
```

Date __/__/__
Page

```
    if result is not None:
        return result
return None
def solve_8_puzzle(initial_board):
    zero_position = next((i, j)
    for i, row in enumerate(initial
    board)
    for j, val in enumerate(row)
    if (val == 0)
    initial_state = PuzzleState(
    initial_board, zero position)
    visited = set()
    return dfs(initial_state, visited)
if __name__ == "__main__":
initial_board = [[1, 2, 3], [4, 5, 6],
                 [7, 8, 0]]
Solution = solve_8_puzzle(initial_board)
if solution is not None:
    Print("Solution found:")
else
    Print("No sol" Exist.")
```

Output

1	2	3
4	0	6
7	5	8

→

1	2	3
4	5	6
7	0	8

→

1	2	3
4	5	6
7	8	0