Week - 03

Date 8/10/2024
Page

8 Puzzle problems using DFS :-

Step 1 : initialize the problem.
* Define the initial state of Puzzle (3x3 grid)
* Define Goal state
* Randomly arranged list of range [1-8]
* Define 1 empty block / space as '0' called tiles.

Step 2 : Define the DFS Function
Defining the functions of DFS needs
* inputs :- current state, goal state, path, visited
* current_state is Equal to goal state:
return path taken to reach goal
* Add current state to visited set.

(3) Generating possible moves :-
find position of blank (or) space
and 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.
recursily call DFS function with
next_state, goal state, updated path
* if a solution is found (return
value is not None) :
return that solution

(5) Backtrack :-
If no solution are found after
exploring all moves from current_state

Date __/__/__
Page _____

remove current state from visited set
and back track
indicate no solution in this path

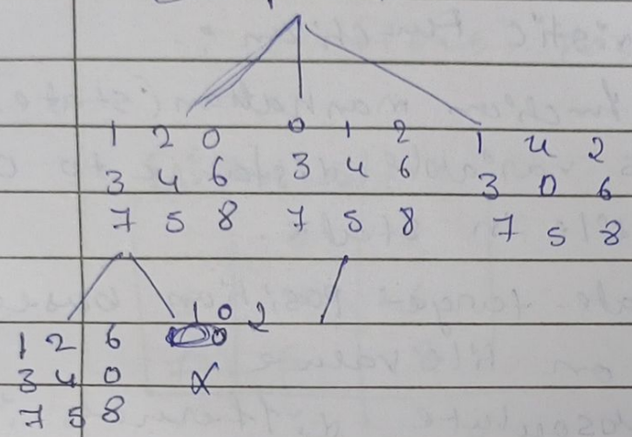
(6) Start construct

initial state

1	0	2
3	4	6
7	5	8

Our Goal

1	2	3
4	5	6
7	8	0



Date __/__/__
Page _____

8-Puzzle Solving using Manhattan Distance

(1) initialize the problem

* Define the initial state of Puzzle (3x3 Grid)
* Define goal state
* Randomly arranged list of range (0-8)
* Define 1 Empty block as '0' called tile

(2) Defining Heuristic Function :-

create a function manhattan(state) :
initializes variable distance to 0
for each tile in state.
calculate target position based on tile value.
Add absoulute differences in row and colum indices to distance and return distance

* 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 the heuristic h_cost using manhattan_distance (next_state).
calculate total cost f_cost = g_cost + h_cost
if next state not visited.
Add (f_cost, next_state, updated_path)
to Priority Queue

Date __/__/__
Page _____

④ Process of priority queue.

while Priority queue not empty
POP state lowest total cost (f(n))
mark it as visited
call the A* function recursively with state & updated costs

⑤ Backtrack.

if no solution found Return to previous position.

initial state

1	0	2
3	4	6
7	5	8

our Goal

1	2	3
4	5	6
7	8	0

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

Priority queue. → POP state with lowest cost.
return Path
make as visited.

Proceed.

Date __/__/__
Page _____

DFS :-

```
class PuzzleState:
    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 = []
        directions = [(-1,0),(1,0),(0,-1),(0,1)]

        for dr, dc in directions:
            new_row, new_col = row + dr, 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 = [r[:] for r 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)])
```

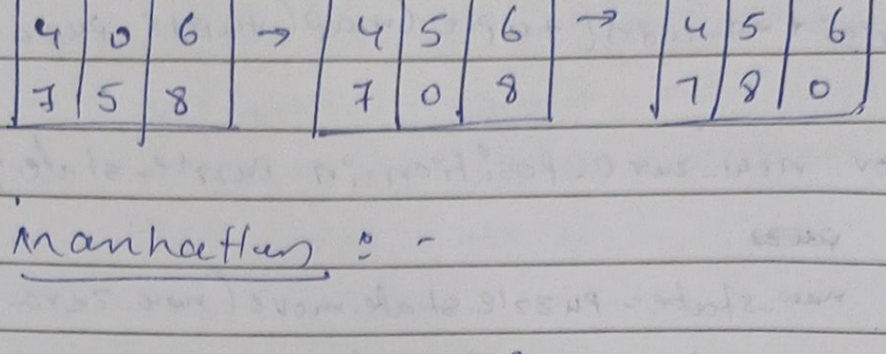
Date __/__/__
Page _____

```
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)
            if result is not None:
                return result
    return None

def solve_8_puzzle(initial_board):
    zero_position = next((i,j) for
    for i, row in enumerate(initial_board):
        for j, val in enumerate(row):
            if(val == 0):
                initial_state = PuzzleState(
                    initial_board, zero_positi
                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! moves:", solution)
    else
        print("No solution exist.")
```

Date __/__/__
Page _____

Sample output

Solution found :



Manhattan :-

```
from collection import deque
goal_state = [[1,2,3],[4,5,6],[7,8,'_']]
moves = [(-1,0),(1,0),(0,-1),(0,1)]

def manhattan_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-gal_j)
        return distance
def get_neighbors(state):
    neighbor = []
    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[i][j], new_state[new_i]
    [new_j] = new_state[new_i][new_j],
    new_state[i][j]
    neighbour.append(new_state)
return neighbour

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 neighbour get_neighbour(current_state):
queue.append([neighbour, path + [neighbour]])
return none.
initial_state = [[1, 1, 3], [3, 2, 6], [5, 8, '-']]
path = dfs(initial_state)
if path:
    Print("solution is found:"):
        for state in path:
            for row in state:
                Print(row);
                print()
else:
    print("No solution is found");
```

Date __/__/__
Page

Solution

Solution is found.