* Lab-03 :-

DATE: 08/10/24 PAGE: 11

* solve 8 puzzle problems using DFS and Manhattan distance

→ The 8 puzzle problem consisting of a 3x3 grid with tiles which are numbered from 1 to 8 and one empty space. the goal is to rearrange the tiles to reach a specified end state.

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

→ end state.

* algorithm :-

initialize :

* create a stack to store the states to be explored. (stack data structure used by DFS)
* Add initial state to the stack
* keep track of states that have been explored as visited set

while the stack is Not empty :-

- pop the top state from
- check if the current state is the goal state.
- return - if yes
- Mark the current state as visited
- Generate all moves to reach end state.

* calculate the manhattan distance
prioritize moves using Manhattan distance
* Repeat the same thing until you get a end state

* Return result : If the goal is reached give it as output or else the puzzle is unsolved.

DATE: PAGE:

* Manhattan distance :-
* Initialization
  - start with the initial state of the puzzle.
  - Define the goal state of the puzzle.
  - check the current state of the puzzle.
  - cost of reaching that state
  - path of moves that led to that state

* calculate the Manhattan distance.
  - compute the manhattan distance blw the crnt position & the target position of that tile.
  - calcuate the sum of distances of all tiles.
* Begin search :- check if this state matches the goal state.
  if yes, retur the path that led to this state or else continue exploring neighboring states
* Explore the neighboring states.
  For each valid move.
  * compute the total cost for reaching the new state (path length + manhattan distance)
  if the state has been visited before mark it as visted otherwise continue to explore
* Repeat till you get end state.
* Return the solution path.
  if the goal is reached give it as output
  Otherwise the puzzle is unsolved.

initial state.

1 2 3
4 0 5
6 7 8

1. up : Moves the empty tile up

1 0 3
4 2 5
6 7 8

* Down : Moves the empty tile down
1 2 3
4 7 5
6 0 8

* Left : Moves the empty tile to the left
1 2 3
0 4 5
6 7 8

* DFS:

1. Initialize:
   - create a stack and push the initial state onto it
   - create a set to track visited states.
2. while the stack is not empty:
   a. Pop the top state from the stack.
   b. If this state is the goal return the solution path.
   c. If the state is not visited
      - Mark it as visited.
      - push all possible new states (by moving the empty tile on to the stack)

* If the goal state is not reached return "No solution found"

* Initial state:
1 2 3
4 0 5
6 7 8

DATE: PAGE:

* up : swap with 2, resulting in
  1 0 3
  4 2 5
  6 7 8
* Down : swap with 7, resulting in
  1 2 3
  4 7 5
  6 0 8
* Left : swap with 4, resulting in
  1 2 3
  0 4 5
  6 7 8

code : DFS

```
class Node:
def __init__(self, state, parent = None, move = None
    self.state = state
    self.parent = parent
    self.move = move.
def print_puzzle(state):
        for i in range(0, 9, 3):
            print(state[i:i+3])
def get_neighbors(node):
    neighbors = []
    state = node.state
    blank_index = state.index(0)
    MOVE = {
        'up': -3
        'down': 3
```

```
'left' : -1
'right' : 1
}

For move, pos_change in moves.items():
    new_blank_index = blank_index + pos_change
    if move == 'left' and blank_index % 3 == 0:
        continue
    if move == 'right' and blank_index % 3 == 2:
        continue
    if new_blank_index < 0 or new_blank_index >= 9:
        continue
    new_state = state[:]
    new_state[blank_index], new_state[new_blank_index]
    = new_state[new_blank_index], new_state[blank_index]
```

8/10/24

* code:

```
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i = (state[i][j] - 1) // 3
                goal_j = (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):
```

## Lab 03:

**8-Puzzel Game**

Code:-

Using Depth First Search (DFS)

```
class SlidingPuzzle:
    def __init__(self, board, empty_pos, path=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.path = path

    def is_solved(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_moves(self):
        x, y = self.empty_pos
        possible_moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny], new_board[x * 3 +
y]
                possible_moves.append((new_board, (nx, ny)))
        return possible_moves

def depth_first_search(initial_puzzle):
    stack, visited = [initial_puzzle], set()
    while stack:
        current_puzzle = stack.pop()
        if current_puzzle.is_solved():
            return current_puzzle.path
```

```
        visited.add(tuple(current_puzzle.board))
        for new_board, new_empty_pos in current_puzzle.get_moves():
            new_state = SlidingPuzzle(new_board, new_empty_pos, current_puzzle.path + [new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None

def display_board(board):
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()

def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_puzzle = SlidingPuzzle(initial_board, (empty_pos // 3, empty_pos % 3))

    print("Initial state:")
    display_board(initial_board)

    solution = depth_first_search(initial_puzzle)

    if solution:
        print("Solution found:")
        for step in solution:
            display_board(step)
    else:
        print("No solution found.")

if __name__ == "__main__":
```

main()

Output:-

```
Initial state:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

Solution found:
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Code:-

Using Manhattan Distance

```
class SlidingPuzzleSolver:
    def __init__(self, initial_state):
        self.initial_state = initial_state
        self.goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def manhattan_distance(self, state):
        distance = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != 0:
                    goal_i = (state[i][j] - 1) // 3
                    goal_j = (state[i][j] - 1) % 3
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance

    def get_neighbors(self, state):
```

```
        i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
        moves = [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]
        return [self.swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

    def swap(self, state, i1, j1, i2, j2):
        new_state = [row[:] for row in state]
        new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
        return new_state

    def dfs_with_manhattan(self, state, visited=set()):
        if state == self.goal_state:
            return [state]
        visited.add(str(state))
        neighbors = sorted(self.get_neighbors(state), key=lambda x: self.manhattan_distance(x))
        for neighbor in neighbors:
            if str(neighbor) not in visited:
                path = self.dfs_with_manhattan(neighbor, visited)
                if path:
                    return [state] + path
        return None

    def solve(self):
        solution = self.dfs_with_manhattan(self.initial_state)
        return solution


initial_state = [[int(x) for x in input(f"Enter row {i + 1}: ").split()] for i in range(3)]
solver = SlidingPuzzleSolver(initial_state)
solution = solver.solve()

if solution:
```

```
    print("Solution found:")
    for state in solution:
        print(*state, sep='\n', end='\n\n')
else:
    print("No solution found.")
```

Output:-

```
>>
===== RESTART: C:/Users/User/AppData/Local/Programs/I
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

[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]
```