Date 18/10/2024
Page

Lab - 2

☆ 8 Puzzle game:

→ BFS Algorithm

Loop
    if fringe is empty return failure
    Node ← remove - first (fringe)
    if Node is a goal
        then return the path from initial
            state to final state
    else Generate all successors of Node.
        And add generated node to the back
        of fringe
End Loop

→ DFS Algorithm

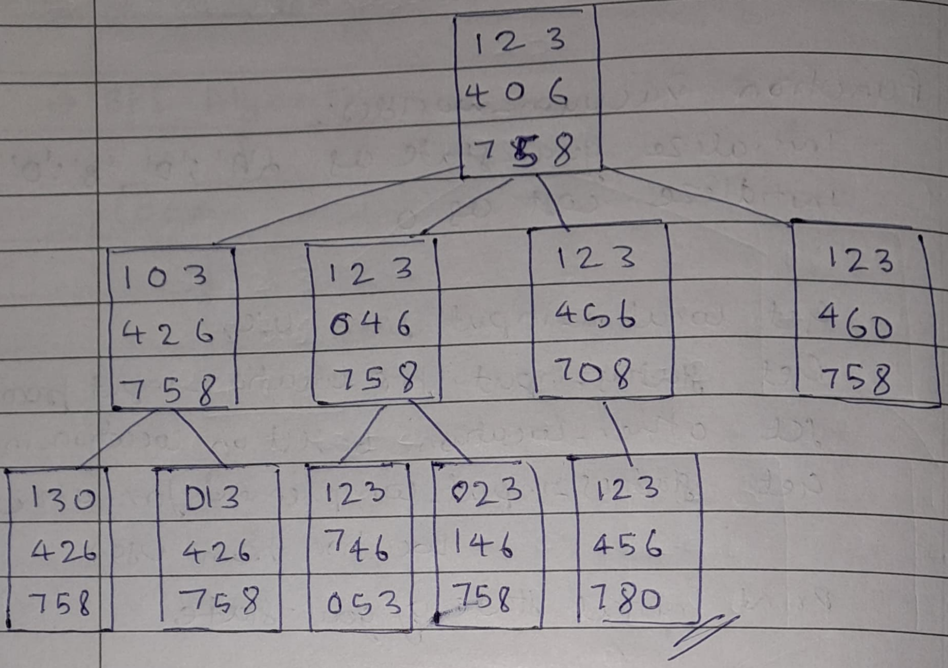
Loop
    if fringe is empty return failure
    Node ← remove - first (fringe)
    if Node is a goal
        then return the path from initial
            state to final state
    else Generate all successors of node.
        And add generated node to the
        front of fringe
End Loop.

* Vacuum cleaner.

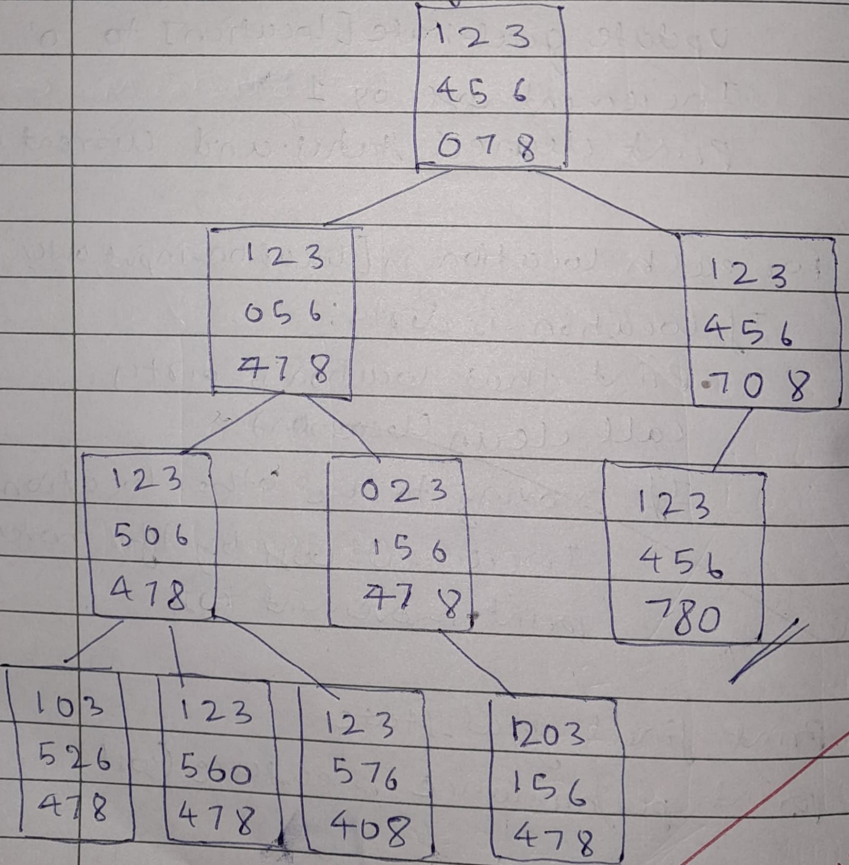
```
Function vacuum-world():
    Initialize goal-state as {'A':'0', 'B':'0'}
    Initidlize cost as 0

    Get location-input from user
    Get status-input for location-input from user
    Set other-location based on location_input
    Get status-input_complement for other-
                    location from user
    Print initial state of goal-state

    Function clean(location):
        Update goal-state [location] to '0'
        Increment cost by 1
        print cleaned status and current cost

    For each location in [location-input, other-location]
        If location is dirty:
            Print that location is dirty
            call clean (location)
        If moving to the other location:
            Increment cost by 1 for movement
            print movement cost

    Print final goal-state
    print performance measure (cost)

call vacuum-world()
```

Date __/__/__
Page ____

→ State space tree for BFS (8 puzzle game)



→ State space tree for DFS (8 puzzle game)



18/10/24

vaccand8puzzle

November 9, 2024

[3]:
```
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

def vacuum_world():
    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum (A or B): ").strip().
↪upper()  # User input for vacuum location
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for␣
↪Dirty): ").strip()  # Status of the current location
    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input(f"Enter status of {other_location} (0 for␣
↪Clean, 1 for Dirty): ").strip()  # Status of the other room

    print("Initial Location Condition: " + str(goal_state))

    # Helper function to clean a location
    def clean(location):
        nonlocal cost
        goal_state[location] = '0'
        cost += 1  # Cost for sucking dirt
        print(f"Location {location} has been Cleaned. Cost: {cost}")

    # Main logic
    if location_input == 'A':
        print("Vacuum is placed in Location A.")
        if status_input == '1':
            print("Location A is Dirty.")
            clean('A')
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print(f"COST for moving RIGHT: {cost}")
                clean('B')
            else:
```

```
                print("Location B is already clean.")
        else:
            print("Location A is already clean.")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print(f"COST for moving RIGHT: {cost}")
                clean('B')
            else:
                print("Location B is already clean.")

    else:  # Vacuum is placed in Location B
        print("Vacuum is placed in Location B.")
        if status_input == '1':
            print("Location B is Dirty.")
            clean('B')
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving left to Location A.")
                cost += 1  # Cost for moving left
                print(f"COST for moving LEFT: {cost}")
                clean('A')
            else:
                print("Location A is already clean.")
        else:
            print("Location B is already clean.")
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving left to Location A.")
                cost += 1  # Cost for moving left
                print(f"COST for moving LEFT: {cost}")
                clean('A')
            else:
                print("Location A is already clean.")

    # Done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

# Output
vacuum_world()
```

```
Name:Sudarshan Komar
USN:1BM22CS291
Enter Location of Vacuum (A or B): B
```

```
Enter status of B (0 for Clean, 1 for Dirty): 1
Enter status of A (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location B.
Location B is Dirty.
Location B has been Cleaned. Cost: 1
Location A is Dirty.
Moving left to Location A.
COST for moving LEFT: 2
Location A has been Cleaned. Cost: 3
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

[4]:

```
# 8 puzzle problem using BFS technique
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

from collections import deque

def solve_8puzzle_bfs(initial_state):
    """
    Solves the 8-puzzle using Breadth-First Search.

    Args:
        initial_state: A list of lists representing the initial state of the␣
↪puzzle.

    Returns:
        A list of lists representing the solution path, or None if no solution␣
↪is found.
    """

    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        if row > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row -␣
↪1][col], new_state[row][col]
```

```
            neighbors.append(new_state)
        if row < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col], new_state[row][col]
            neighbors.append(new_state)
        if col > 0:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1], new_state[row][col]
            neighbors.append(new_state)
        if col < 2:
            new_state = [row[:] for row in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1], new_state[row][col]
            neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path + [current_state]

        visited.add(tuple(map(tuple, current_state)))
        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_8puzzle_bfs(initial_state)

if solution:
    print("Solution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

```
Name:Sudarshan Komar
USN:1BM22CS291
Solution found:
[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]
```

[5]:

```python
# 8 puzzle problem using DFS technique
print("Name:Sudarshan Komar","USN:1BM22CS291",sep="\n")

from collections import deque

def solve_8puzzle_dfs(initial_state):
    """
    Solves the 8-puzzle using Depth-First Search.

    Args:
        initial_state: A list of lists representing the initial state of the␣
    ↪puzzle.

    Returns:
        A list of lists representing the solution path, or None if no solution␣
    ↪is found.
    """

    def find_blank(state):
        """Finds the row and column of the blank tile."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dr, dc in directions:
```

```
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [r[:] for r in state]
                new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[row][col]
                neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()
        state_tuple = tuple(map(tuple, current_state))  # Convert to tuple for set
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        if current_state == goal_state:
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            stack.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)

if solution:
    print("Solution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

```
Name:Sudarshan Komar
USN:1BM22CS291
Solution found:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
```

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

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