6)AIM:Write the python program for Vacuum Cleaner problem.

ALGORITHM:

1.Initialize: Define the environment (dirty and clean squares) and vacuum's initial position.

2.Generate Actions: Define possible actions (move, clean).

3.Apply Actions: Simulate the result of actions on the environment.

4.Check for Goal: Verify if all squares are clean.

5.Plan Path: Use search algorithms (BFS, DFS) to determine the sequence of actions to clean all squares.

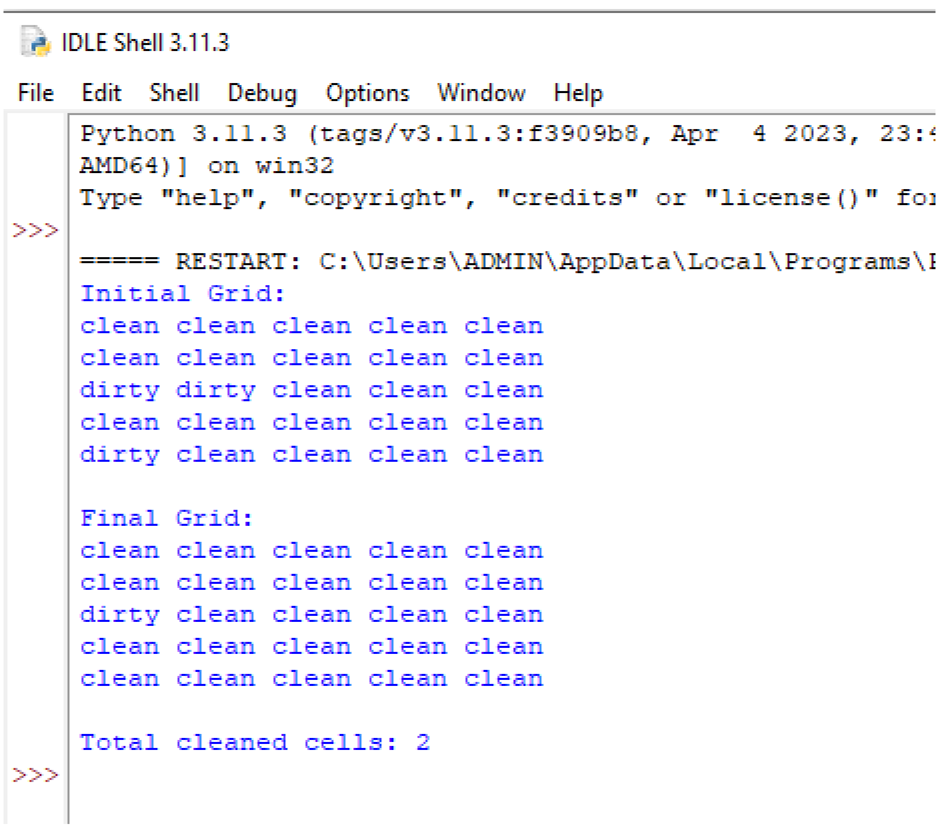
PROGRAM:

```
import random

class VacuumCleaner:

    def __init__(self, grid_size):

        self.grid_size = grid_size

        self.grid = [['dirty' if random.random() < 0.3 else 'clean' for _ in range(grid_size)] for _ in
range(grid_size)]

        self.x = random.randint(0, grid_size - 1)

        self.y = random.randint(0, grid_size - 1)

        self.cleaned_cells = 0

    def print_grid(self):

        for row in self.grid:

            print(' '.join(row))

        print()

    def move(self, direction):

        if direction == 'up' and self.x > 0:

            self.x -= 1

        elif direction == 'down' and self.x < self.grid_size - 1:

            self.x += 1
```

```
    elif direction == 'left' and self.y > 0:
        self.y -= 1
    elif direction == 'right' and self.y < self.grid_size - 1:
        self.y += 1
def clean(self):
    if self.grid[self.x][self.y] == 'dirty':
        self.grid[self.x][self.y] = 'clean'
        self.cleaned_cells += 1
def get_possible_moves(self):
    moves = []
    if self.x > 0:
        moves.append('up')
    if self.x < self.grid_size - 1:
        moves.append('down')
    if self.y > 0:
        moves.append('left')
    if self.y < self.grid_size - 1:
        moves.append('right')
    return moves
def run(self):
    while self.cleaned_cells < sum(row.count('dirty') for row in self.grid):
        self.clean()
        possible_moves = self.get_possible_moves()
        if possible_moves:
            self.move(random.choice(possible_moves))
```

```
def main():
    grid_size = 5
    vacuum = VacuumCleaner(grid_size)
    print("Initial Grid:")
    vacuum.print_grid()
    vacuum.run()
    print("Final Grid:")
    vacuum.print_grid()
    print(f"Total cleaned cells: {vacuum.cleaned_cells}")
if __name__ == "__main__":
    main()
```

OUTPUT



7)AIM: Breadth-First Search (BFS)

ALGORITHM:

1.Initialize: Start from the initial node and add it to a queue.

2.Explore Nodes: Dequeue a node and explore its neighbors.

3.Check for Goal: If a neighbor is the goal, return the path.

4.Queue Neighbors: Add unvisited neighbors to the queue.

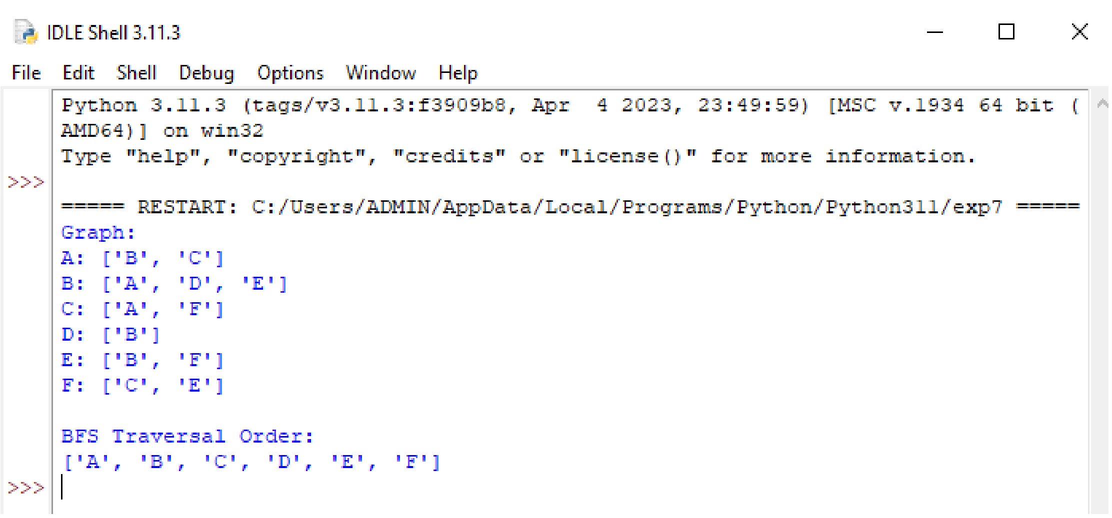
5.Repeat: Continue until the queue is empty or the goal is found.

PROGRAM:

```
from collections import deque

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])
    traversal_order = []
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            traversal_order.append(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
    return traversal_order
def main():
    graph = {
        'A': ['B', 'C'],
```

```
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }
    start_node = 'A'
    print("Graph:")
    for node, neighbors in graph.items():
        print(f"{node}: {neighbors}")
    print("\nBFS Traversal Order:")
    traversal_order = bfs(graph, start_node)
    print(traversal_order)
if __name__ == "__main__":
    main()
```

output



```
IDLE Shell 3.11.3
File Edit Shell Debug Options Window Help
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python311/exp7 =====
Graph:
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A', 'F']
D: ['B']
E: ['B', 'F']
F: ['C', 'E']

BFS Traversal Order:
['A', 'B', 'C', 'D', 'E', 'F']
>>> |
```

8)AIM: Depth-First Search (DFS)

ALGORITHM:

1.Initialize: Start from the initial node and add it to a stack.

2.Explore Nodes: Pop a node from the stack and explore its neighbors.

3.Check for Goal: If a neighbor is the goal, return the path.

4.Stack Neighbors: Add unvisited neighbors to the stack.

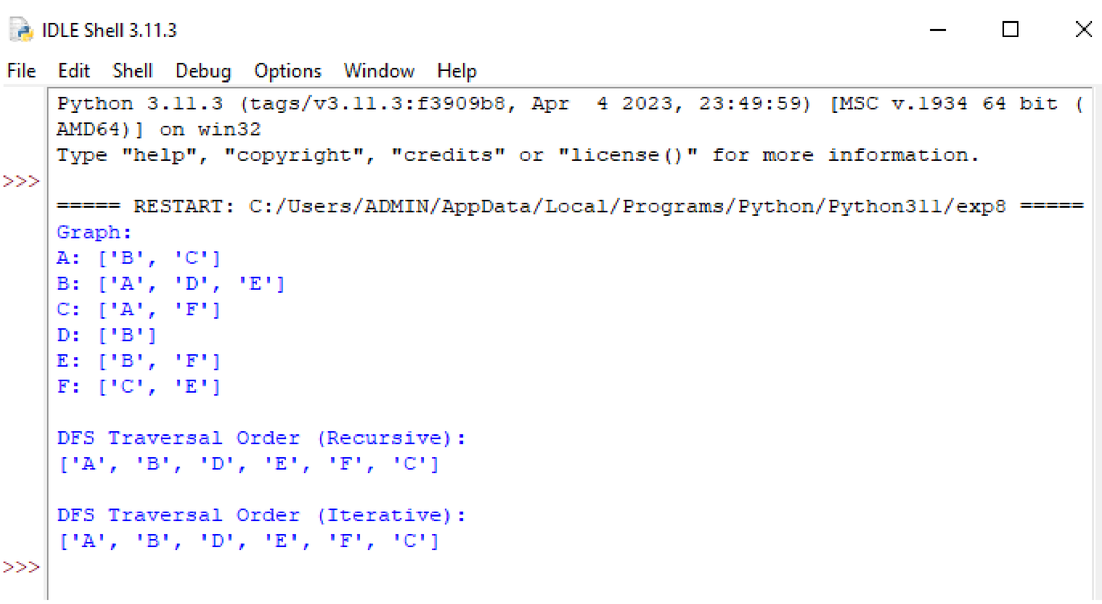
5.Repeat: Continue until the stack is empty or the goal is found.

PROGRAM:

```
def dfs_recursive(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    return [node] + [n for neighbor in graph[node] if neighbor not in visited for n in dfs_recursive(graph,
neighbor, visited)]
def dfs_iterative(graph, start_node):
    visited, stack, order = set(), [start_node], []
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            order.append(node)
            stack.extend(reversed(graph[node]))
    return order
def main():
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
```

```
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }
    start_node = 'A'
    print("Graph:")
    for node, neighbors in graph.items():
        print(f"{node}: {neighbors}")
    print("\nDFS Traversal Order (Recursive):")
    print(dfs_recursive(graph, start_node))
    print("\nDFS Traversal Order (Iterative):")
    print(dfs_iterative(graph, start_node))
if __name__ == "__main__":
    main()
```

output



```
IDLE Shell 3.11.3
File  Edit  Shell  Debug  Options  Window  Help
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python311/exp8 =====
Graph:
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A', 'F']
D: ['B']
E: ['B', 'F']
F: ['C', 'E']

DFS Traversal Order (Recursive):
['A', 'B', 'D', 'E', 'F', 'C']

DFS Traversal Order (Iterative):
['A', 'B', 'D', 'E', 'F', 'C']
>>>
```

9)AIM: Travelling Salesman Problem (TSP)

ALGORITHM:

1.Initialize: Define cities and distances between them.

2.Generate Permutations: Create all possible routes.

3.Calculate Costs: Compute the total distance for each route.

4.Find Minimum: Identify the route with the minimum total distance.

5.Return Solution: Return the shortest route and its cost.

PROGRAM:

```
from itertools import permutations
def calculate_path_cost(distance_matrix, path):
    cost = 0
    for i in range(len(path) - 1):
        cost += distance_matrix[path[i]][path[i + 1]]
    cost += distance_matrix[path[-1]][path[0]]
    return cost
def tsp_bruteforce(distance_matrix):
    n = len(distance_matrix)
    cities = list(range(n))
    min_cost = float('inf')
    best_path = []
    for perm in permutations(cities):
        current_cost = calculate_path_cost(distance_matrix, perm)
        if current_cost < min_cost:
            min_cost = current_cost
            best_path = perm
    return min_cost, best_path
def main():
    distance_matrix = [
        [0, 10, 15, 20],
```

```
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
    min_cost, best_path = tsp_bruteforce(distance_matrix)
    print("Optimal Path:", best_path)
    print("Minimum Cost:", min_cost)
if __name__ == "__main__":
    main()
```

OUTPUT:

```
===== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python311/exp9
Optimal Path: (0, 1, 3, 2)
Minimum Cost: 80
```

10)AIM:A* Algorithm

ALGORITHM:

1.Initialize: Start from the initial state, define the goal, and create open and closed lists.

2.Generate Successors: Create possible moves from the current state.

3.Evaluate Cost: Compute the cost (g) and heuristic (h) for each successor.

4.Select Node: Choose the node with the lowest f = g + h from the open list.

5.Check for Goal: If the node is the goal, return the path.

PROGRAM:

```
import heapq
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def astar(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    open_set = [(0 + heuristic(start, goal), 0, start)]
```

```
    came_from = {}
    cost_so_far = {start: 0}
    while open_set:
        _, current_cost, current = heapq.heappop(open_set)
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path[::-1] + [goal]
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:
                new_cost = current_cost + 1
                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + heuristic(neighbor, goal)
                    heapq.heappush(open_set, (priority, new_cost, neighbor))
                    came_from[neighbor] = current
    return []
def print_grid(grid, path):
    path_set = set(path)
    for i, row in enumerate(grid):
        for j, cell in enumerate(row):
            if (i, j) == path[0]: print('S', end=' ')
            elif (i, j) == path[-1]: print('G', end=' ')
            elif (i, j) in path_set: print('.', end=' ')
            else: print('#' if cell == 1 else ' ', end=' ')
```

```
        print()
def main():
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0]
    ]
    start, goal = (0, 0), (4, 4)
    path = astar(grid, start, goal)
    if path:
        print("Path found:")
        print_grid(grid, path)
    else:
        print("No path found.")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
===== RESTART: C:/Users/ADMIN/AppData/Local/Programs/Python/Python311/exp10 ====
Path found:
  S . . .
  # # # .
      # .
  #     .
        G
>>> |
```