AI Laboratory Programs (For D,E,F Sections)

1. DFS and BFS

```
from collections import deque

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

def bfs(start):
    visited = []
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            queue.extend(graph[node])
    return visited

def dfs(start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(neighbor, visited)
    return visited

print("BFS:", bfs('A'))
print("DFS:", dfs('A'))
```

2. Travelling Salesman Problem

```
from itertools import permutations

def travelling_salesman(graph, start):
    nodes = list(graph.keys())
    nodes.remove(start)
    min_path = float('inf')
    best_route = []
    for perm in permutations(nodes):
        current_cost = 0
        k = start
        for j in perm:
            current_cost += graph[k][j]
            k = j
        current_cost += graph[k][start]
        if current_cost < min_path:
            min_path = current_cost
            best_route = perm
    return min_path, [start] + list(best_route) + [start]

graph = {
    'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}
}

print(travelling_salesman(graph, 'A'))
```

3. Simulated Annealing Algorithm

```
import math, random

def objective_function(x):
    return x**2  # minimize x²

def simulated_annealing():
    temp = 10000
    cooling_rate = 0.98
    current_x = random.uniform(-10, 10)
    best_x = current_x

    while temp > 1:
        new_x = current_x + random.uniform(-1, 1)
        cost_diff = objective_function(new_x) - objective_function(current_x)

        if cost_diff < 0 or random.random() < math.exp(-cost_diff / temp):
            current_x = new_x
            if objective_function(current_x) < objective_function(best_x):
                best_x = current_x

        temp *= cooling_rate

    return best_x, objective_function(best_x)

print("Best solution:", simulated_annealing())
```

4. 8 Puzzle Problem

```
from collections import deque

def bfs(start, goal):
    queue = deque([(start, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if state == goal:
            return path + [state]
        visited.add(tuple(state))
        index = state.index(0)
        moves = []
        if index not in [0, 1, 2]: moves.append(-3)
        if index not in [6, 7, 8]: moves.append(3)
        if index not in [0, 3, 6]: moves.append(-1)
        if index not in [2, 5, 8]: moves.append(1)

        for move in moves:
            new_state = state[:]
            new_state[index], new_state[index + move] = new_state[index + move], new_state[index]
            if tuple(new_state) not in visited:
                queue.append((new_state, path + [state]))

start = [1, 2, 3, 4, 0, 5, 6, 7, 8]
goal = [1, 2, 3, 4, 5, 0, 6, 7, 8]
result = bfs(start, goal)
for step in result:
    print(step)
```

5. Tower of Hanoi

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, target, source)

tower_of_hanoi(3, 'A', 'C', 'B')
```

6. A* Algorithm

```
from queue import PriorityQueue

def a_star(graph, start, goal, h):
    open_list = PriorityQueue()
    open_list.put((0, start))
    g = {start: 0}
    came_from = {}

    while not open_list.empty():
        _, current = open_list.get()
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor, cost in graph[current]:
            temp_g = g[current] + cost
            if neighbor not in g or temp_g < g[neighbor]:
                g[neighbor] = temp_g
                f = temp_g + h[neighbor]
                open_list.put((f, neighbor))
                came_from[neighbor] = current

graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 3), ('E', 1)],
    'C': [('F', 5)],
    'D': [],
    'E': [('F', 2)],
    'F': []
}

h = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 1, 'F': 0}
print("Path:", a_star(graph, 'A', 'F', h))
```

7. Hill Climbing Algorithm

```
import random

def hill_climb():
    def fitness(x): return -(x - 3)**2 + 9  # peak at x=3
    current = random.uniform(-10, 10)
    step_size = 0.1
    for _ in range(1000):
        neighbor = current + random.uniform(-step_size, step_size)
        if fitness(neighbor) > fitness(current):
            current = neighbor
    return current, fitness(current)

print("Best solution:", hill_climb())
```

8. Monkey Banana Problem

```
def monkey_banana():
    states = ['at_door', 'at_window', 'on_box', 'has_banana']
    monkey = {'position': 'door', 'on_box': False, 'has_banana': False}

    print("Initial:", monkey)

    print("Monkey moves from door to window")
    monkey['position'] = 'window'
    print("Monkey pushes box under banana")
    print("Monkey climbs box")
    monkey['on_box'] = True

    print("Monkey grabs banana")
    monkey['has_banana'] = True

    print("Final:", monkey)

monkey_banana()
```