无信息搜索实验报告

姓名：_____ 张韫译萱_____ 学号：_____08023214_____

一、实验题目

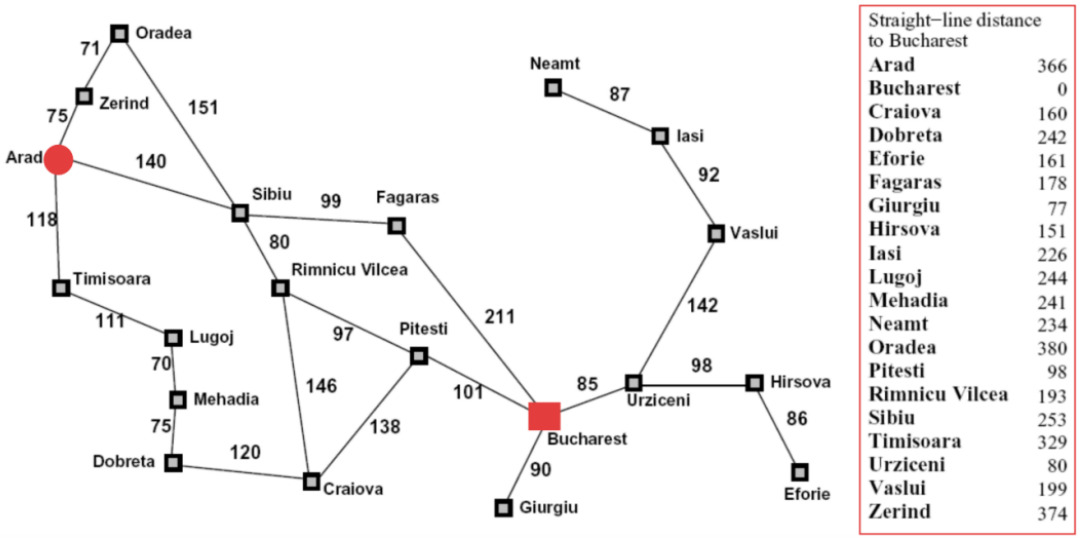

图 1: 罗马尼亚地图

1、实验任务

1. 使用 BFS（广度优先搜索）算法求出从 Arad 到 Bucharest 的路径，给出经过的节点。
2. 使用 DFS（深度优先搜索）算法求出从 Arad 到 Bucharest 的路径，给出经过的节点。
3. 使用 UCS（一致代价搜索）算法求出从 Arad 到 Bucharest 的路径，给出经过的节点。

2、实验要求

1. 选择 C++ 或 Python 实现。
2. 代码以文本形式粘贴在附录相应位置，注释准确，能成功运行。

二、实验结果

1、BFS 算法运行结果

```
开始BFS搜索
当前扩展节点: Arad, 当前路径: Arad,路径代价: 0
  将节点 Sibiu 入队
  将节点 Timisoara 入队
  将节点 Zerind 入队
当前扩展节点: Sibiu, 当前路径: Arad -> Sibiu,路径代价: 140
  将节点 Fagaras 入队
  将节点 Oradea 入队
  将节点 Rimnicu Vilcea 入队
当前扩展节点: Timisoara, 当前路径: Arad -> Timisoara,路径代价: 118
  将节点 Lugoj 入队
当前扩展节点: Zerind, 当前路径: Arad -> Zerind,路径代价: 75
当前扩展节点: Fagaras, 当前路径: Arad -> Sibiu -> Fagaras,路径代价: 239
  将节点 Bucharest 入队
当前扩展节点: Oradea, 当前路径: Arad -> Sibiu -> Oradea,路径代价: 291
当前扩展节点: Rimnicu Vilcea, 当前路径: Arad -> Sibiu -> Rimnicu Vilcea,路径代价: 220
  将节点 Craiova 入队
  将节点 Pitesti 入队
当前扩展节点: Lugoj, 当前路径: Arad -> Timisoara -> Lugoj,路径代价: 229
  将节点 Mehadia 入队
当前扩展节点: Bucharest, 当前路径: Arad -> Sibiu -> Fagaras -> Bucharest,路径代价: 450
OK
路径: Arad -> Sibiu -> Fagaras -> Bucharest
路径代价: 450
被搜索的节点: ['Arad', 'Fagaras', 'Lugoj', 'Oradea', 'Rimnicu Vilcea', 'Sibiu', 'Timisoara', 'Zerind']
```

图 2: BFS 算法搜索结果

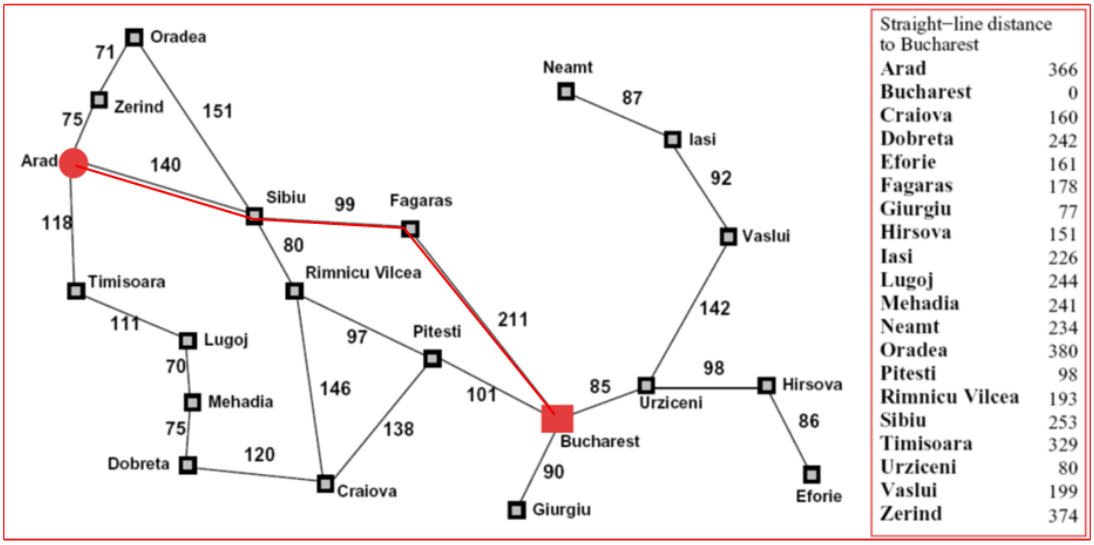

图 3: BFS 算法搜索结果可视化

2、DFS 算法运行结果

```
开始DFS搜索
当前扩展节点：Arad，当前路径：Arad，当前路径代价：0
将节点 Sibiu 加入栈
将节点 Timisoara 加入栈
将节点 Zerind 加入栈
当前扩展节点：Zerind，当前路径：Arad -> Zerind，当前路径代价：75
将节点 Oradea 加入栈
当前扩展节点：Oradea，当前路径：Arad -> Zerind -> Oradea，当前路径代价：146
将节点 Sibiu 加入栈
当前扩展节点：Sibiu，当前路径：Arad -> Zerind -> Oradea -> Sibiu，当前路径代价：297
将节点 Fagaras 加入栈
将节点 Rimnicu Vilcea 加入栈
当前扩展节点：Rimnicu Vilcea，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea，当前路径代价：377
将节点 Craiova 加入栈
将节点 Pitesti 加入栈
当前扩展节点：Pitesti，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti，当前路径代价：474
将节点 Bucharest 加入栈
将节点 Craiova 加入栈
当前扩展节点：Craiova，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Craiova，当前路径代价：612
将节点 Dobreta 加入栈
当前扩展节点：Dobreta，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Craiova -> Dobreta，当前路径代价：732
将节点 Mehadia 加入栈
当前扩展节点：Mehadia，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Craiova -> Dobreta -> Mehadia，当前路径代价：807
将节点 Lugoj 加入栈
当前扩展节点：Lugoj，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Craiova -> Dobreta -> Mehadia -> Lugoj，当前路径代价：877
将节点 Timisoara 加入栈
当前扩展节点：Timisoara，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Craiova -> Dobreta -> Mehadia -> Lugoj -> Timisoara，当前路径代价：
988
当前扩展节点：Bucharest，当前路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest，当前路径代价：575
OK
路径：Arad -> Zerind -> Oradea -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
路径代价：575
被搜索的节点：['Arad', 'Craiova', 'Dobreta', 'Lugoj', 'Mehadia', 'Oradea', 'Pitesti', 'Rimnicu Vilcea', 'Sibiu', 'Timisoara', 'Zerind']
```

图 4: DFS 算法搜索结果

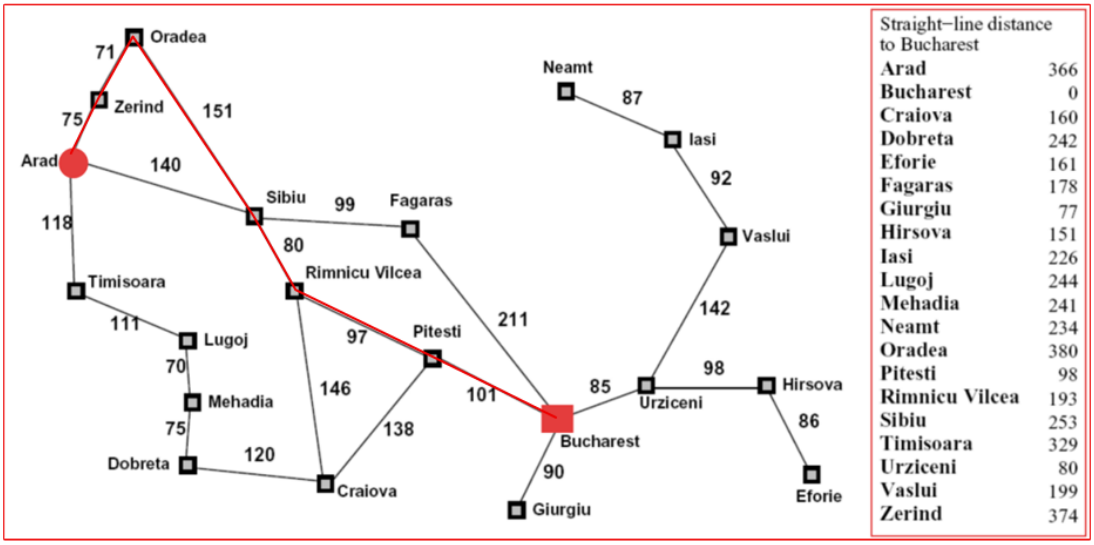

图 5: DFS 算法搜索结果可视化

3、UCS 算法运行结果

```
开始UCS搜索
当前扩展节点: Arad, 累计代价: 0, 当前路径: Arad
  将节点 Sibiu 加入优先队列 (代价: 140)
  将节点 Timisoara 加入优先队列 (代价: 118)
  将节点 Zerind 加入优先队列 (代价: 75)
当前扩展节点: Zerind, 累计代价: 75, 当前路径: Arad -> Zerind
  将节点 Oradea 加入优先队列 (代价: 146)
当前扩展节点: Timisoara, 累计代价: 118, 当前路径: Arad -> Timisoara
  将节点 Lugoj 加入优先队列 (代价: 229)
当前扩展节点: Sibiu, 累计代价: 140, 当前路径: Arad -> Sibiu
  将节点 Fagaras 加入优先队列 (代价: 239)
  将节点 Rimnicu Vilcea 加入优先队列 (代价: 220)
当前扩展节点: Oradea, 累计代价: 146, 当前路径: Arad -> Zerind -> Oradea
当前扩展节点: Rimnicu Vilcea, 累计代价: 220, 当前路径: Arad -> Sibiu -> Rimnicu Vilcea
  将节点 Craiova 加入优先队列 (代价: 366)
  将节点 Pitesti 加入优先队列 (代价: 317)
当前扩展节点: Lugoj, 累计代价: 229, 当前路径: Arad -> Timisoara -> Lugoj
  将节点 Mehadia 加入优先队列 (代价: 299)
当前扩展节点: Fagaras, 累计代价: 239, 当前路径: Arad -> Sibiu -> Fagaras
  将节点 Bucharest 加入优先队列 (代价: 450)
当前扩展节点: Mehadia, 累计代价: 299, 当前路径: Arad -> Timisoara -> Lugoj -> Mehadia
  将节点 Dobreta 加入优先队列 (代价: 374)
当前扩展节点: Pitesti, 累计代价: 317, 当前路径: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti
  将节点 Bucharest 加入优先队列 (代价: 418)
当前扩展节点: Craiova, 累计代价: 366, 当前路径: Arad -> Sibiu -> Rimnicu Vilcea -> Craiova
当前扩展节点: Dobreta, 累计代价: 374, 当前路径: Arad -> Timisoara -> Lugoj -> Mehadia -> Dobreta
当前扩展节点: Bucharest, 累计代价: 418, 当前路径: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
--------------------------------------------------
OK

路径: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
总代价: 418
已探索节点: ['Arad', 'Craiova', 'Dobreta', 'Fagaras', 'Lugoj', 'Mehadia', 'Oradea', 'Pitesti', 'Rimnicu Vilcea', 'Sibiu', 'Timisoara', 'Zerind']
```

图 6: UCS 算法搜索结果

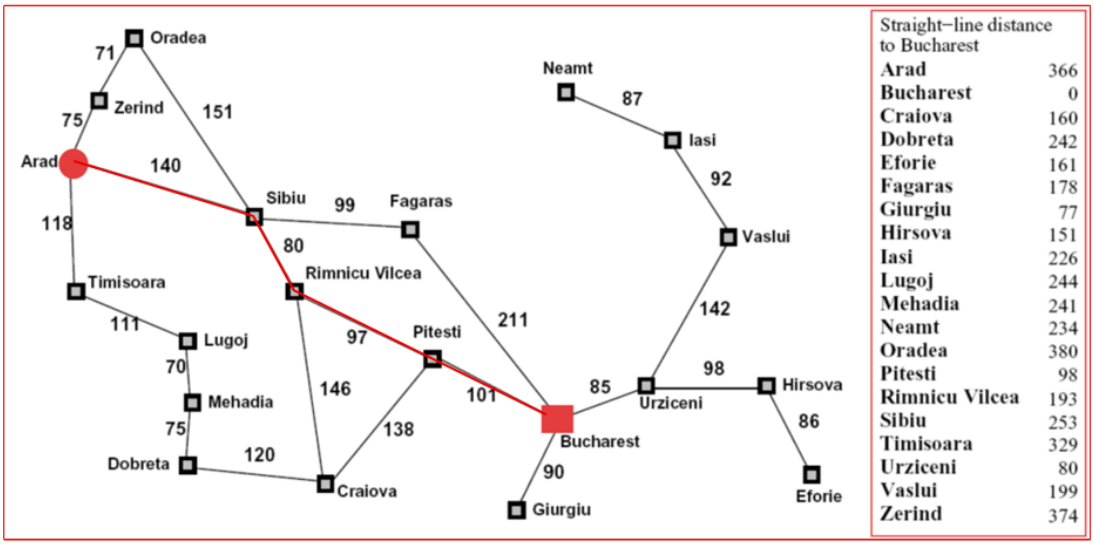

图 7: UCS 算法搜索结果可视化

三、实验分析

通过对各算法的对比，可以总结出各算法的特点以及它们的适用场景。

BFS 采用 FIFO 结构，确保按层次顺序（题目中为节点的首字母顺序）扩展节点，因此在图搜索和树搜索时都能够找到一条路径。本次实验中，BFS 找到路径的总代价为 450，并非最优。DFS 使用 LIFO 结构，优先从队尾扩展节点（优先深入探索），在该地图中搜索的节点数较多，且并未找到最优解。UCS 使用优先队列按路径代价排序，保证找到代价最小的路径，本次搜索的节点数最多，但找到的路径代价为 418，总代价最小。

理论分析下，基于树搜索的 BFS 时间复杂度和空间复杂度均为 $O(b^d)$，取决于目标节点的深度；基于树搜索的 DFS 时间复杂度和空间复杂度均为 $O(b^m)$，取决于搜索树的最大深度；基于树搜索的 UCS 时间复杂度和空间复杂度均为 $O(b^{(1+\frac{C*}{\epsilon})})$，取决于路径代价成本和最短边代价成本的比值。从已探索节点数可以看出，BFS 探索了 8 个节点，DFS 探索了 10 个节点，而 UCS 探索了 12 个节点。在代码中分别计算三种算法的运行时间，得到 BFS 算法的运行时间约为 0.0024s，DFS 算法的平均运行时间约为 0.0029s，UCS 算法的平均运行时间约为 0.003s。基于具体情境，上述结果能够支持理论分析。

基于图搜索的三种算法均具有完备性，但在最优性方面，只有 UCS 能够保证找到代价最小的路径。BFS 仅在路径成本是深度的非递减函数时才是最优的，DFS 则完全不保证最优性。实验结果充分验证了理论分析的正确性。

总体来说，BFS 适用于寻找最少步数（经过节点数最少）的路径问题，如迷宫路径求解等。DFS 适用于路径存在性判断、拓扑排序等不要求最优解的场景，且在深度有限的情况下效率较高。UCS 适用于边权不同的图中寻找最优路径，如路径规划、网络路由等实际应用场景，是 Dijkstra 算法的前身。

四、实验总结

通过本次无信息搜索算法实验，我深入理解了 BFS、DFS 和 UCS 三种无信息搜索算法的原理与实现细节。实验过程中，我使用 Python 实现了基于图搜索的三种算法，通过维护已探索集合有效避免了循环和重复访问问题（尤其需要注意的是，节点不再重复扩展的条件是该节点已经被搜索过（出栈），判断条件设定应正确）。

实验结果表明，不同的搜索策略会导致截然不同的搜索过程和结果。BFS 的层次扩展策略适合寻找最少步数的解，DFS 的深度优先策略在某些情况下效率较高但不保证最优性，而 UCS 通过代价排序保证了解的最优性但需要更多的计算开销。这三种算法各有优势，在实际应用中应根据问题特性选择合适的算法。

附录

1. BFS 算法代码（Python）

```python
from collections import deque

graph = {
    'Arad': [ ('Sibiu', 140), ('Timisoara', 118),('Zerind', 75)],
    'Zerind': [('Arad', 75), ('Oradea', 71)],
    'Oradea': [('Sibiu', 151),('Zerind', 71)],
    'Sibiu': [('Arad', 140), ('Fagaras', 99), ('Oradea', 151), ('Rimnicu Vilcea', 80)],
    'Timisoara': [('Arad', 118), ('Lugoj', 111)],
    'Lugoj': [('Mehadia', 70),('Timisoara', 111)],
    'Mehadia': [('Dobreta', 75),('Lugoj', 70)],
    'Dobreta': [('Craiova', 120),('Mehadia', 75)],
    'Craiova': [('Dobreta', 120), ('Pitesti', 138), ('Rimnicu Vilcea', 146)],
    'Rimnicu Vilcea': [('Craiova', 146), ('Pitesti', 97), ('Sibiu', 80)],
    'Fagaras': [('Bucharest', 211),('Sibiu', 99)],
    'Pitesti': [('Bucharest', 101),('Craiova', 138),('Rimnicu Vilcea', 97)],
    'Bucharest': [('Fagaras', 211), ('Giurgiu', 90), ('Pitesti', 101), ('Urziceni', 85)],
    'Giurgiu': [('Bucharest', 90)],
    'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],
    'Hirsova': [('Eforie', 86),('Urziceni', 98)],
    'Eforie': [('Hirsova', 86)],
    'Vaslui': [('Iasi', 92),('Urziceni', 142)],
    'Iasi': [('Neamt', 87), ('Vaslui', 92)],
    'Neamt': [('Iasi', 87)]
}


def bfs_search(graph, start, goal):#FIFO
    frontier = deque()
    frontier.append((start, [start],0))
    # 已探索集合
    explored = set()
    print("开始BFS搜索")
    while frontier:
        current_node, path, distance1 = frontier.popleft()#从队头取出节点

        print(f"当前扩展节点: {current_node}, 当前路径: {' -> '.join(path)},路径代价: {
    distance1}")

        if current_node == goal:#找到目标
            print(f"OK")
            return path, explored,distance1

        # 将当前节点加入已探索集合
```

```
        if current_node not in explored:
            explored.add(current_node)

            # 扩展当前节点的所有邻居
            for neighbor, distance in graph.get(current_node, []): #解包
                # 如果邻居未被探索且不在边缘中
                if neighbor not in explored:
                    # 检查邻居是否已在队列中
                    in_frontier = any(node == neighbor for node, _ ,__ in frontier)
                    if not in_frontier:
                        # 将邻居加入队列
                        new_path = path + [neighbor]
                        Distance = distance + distance1
                        frontier.append((neighbor, new_path,Distance))
                        print(f"  将节点 {neighbor} 入队")

    return None, explored,distance1


if __name__ == "__main__":
    start_city = 'Arad'
    goal_city = 'Bucharest'

    path, explored ,distance = bfs_search(graph, start_city, goal_city)

    if path:
        print(f"路径: {' -> '.join(path)}")
        print(f"路径代价: {distance}")
        print(f"被搜索的节点: {sorted(explored)}")
    else:
        print(f"没有找到路径")
```

2. DFS 算法代码（Python）

```python
from collections import deque

graph = {
    'Arad': [ ('Sibiu', 140), ('Timisoara', 118),('Zerind', 75)],
    'Zerind': [('Arad', 75), ('Oradea', 71)],
    'Oradea': [('Sibiu', 151),('Zerind', 71)],
    'Sibiu': [('Arad', 140), ('Fagaras', 99), ('Oradea', 151), ('Rimnicu Vilcea', 80)],
    'Timisoara': [('Arad', 118), ('Lugoj', 111)],
    'Lugoj': [('Mehadia', 70),('Timisoara', 111)],
    'Mehadia': [('Dobreta', 75),('Lugoj', 70)],
    'Dobreta': [('Craiova', 120),('Mehadia', 75)],
    'Craiova': [('Dobreta', 120), ('Pitesti', 138), ('Rimnicu Vilcea', 146)],
    'Rimnicu Vilcea': [('Craiova', 146), ('Pitesti', 97), ('Sibiu', 80)],
    'Fagaras': [('Bucharest', 211),('Sibiu', 99)],
    'Pitesti': [('Bucharest', 101),('Craiova', 138),('Rimnicu Vilcea', 97)],
    'Bucharest': [('Fagaras', 211), ('Giurgiu', 90), ('Pitesti', 101), ('Urziceni', 85)],
    'Giurgiu': [('Bucharest', 90)],
    'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],
    'Hirsova': [('Eforie', 86),('Urziceni', 98)],
    'Eforie': [('Hirsova', 86)],
    'Vaslui': [('Iasi', 92),('Urziceni', 142)],
    'Iasi': [('Neamt', 87), ('Vaslui', 92)],
    'Neamt': [('Iasi', 87)]
}

def dfs_search(graph, start, goal):#LIFO
    frontier = deque()
    frontier.append((start, [start], 0))
    explored = set()

    print(f"开始DFS搜索")

    while frontier:
        current_node, path ,distance1 = frontier.pop() # 从队尾取出节点

        print(f"当前扩展节点: {current_node}, 当前路径: {' -> '.join(path)}, 当前路径代价: {distance1}")

        if current_node == goal:
            print(f"OK")
            return path, explored,distance1

        # 将当前节点加入已探索集合
        if current_node not in explored:
            explored.add(current_node)

```

```
            neighbors = graph.get(current_node, [])
            for neighbor, distance in (neighbors):#扩展邻居
                # 如果邻居未被探索且不在边缘中
                if neighbor not in explored:
                    # 检查邻居是否已在栈中
                    in_frontier = any(node == neighbor for node, _ ,__ in frontier)
                    if not in_frontier:
                        # 将邻居加入栈
                        Distance = distance + distance1
                        new_path = path + [neighbor]
                        frontier.append((neighbor, new_path,Distance))
                        print(f"将节点 {neighbor} 加入栈")

    return None, explored, distance1


if __name__ == "__main__":
    start_city = 'Arad'
    goal_city = 'Bucharest'

    path, explored, distance = dfs_search(graph, start_city, goal_city)

    if path:
        print(f"路径: {' -> '.join(path)}")
        print(f"路径代价: {distance}")
        print(f"被搜索的节点: {sorted(explored)}")
    else:
        print(f"\n从 {start_city} 到 {goal_city} 没有找到路径")
```

3. UCS 算法代码（Python）

```python
import heapq

graph = {
    'Arad': [ ('Sibiu', 140), ('Timisoara', 118),('Zerind', 75)],
    'Zerind': [('Arad', 75), ('Oradea', 71)],
    'Oradea': [('Sibiu', 151),('Zerind', 71)],
    'Sibiu': [('Arad', 140), ('Fagaras', 99), ('Oradea', 151), ('Rimnicu Vilcea', 80)],
    'Timisoara': [('Arad', 118), ('Lugoj', 111)],
    'Lugoj': [('Mehadia', 70),('Timisoara', 111)],
    'Mehadia': [('Dobreta', 75),('Lugoj', 70)],
    'Dobreta': [('Craiova', 120),('Mehadia', 75)],
    'Craiova': [('Dobreta', 120), ('Pitesti', 138), ('Rimnicu Vilcea', 146)],
    'Rimnicu Vilcea': [('Craiova', 146), ('Pitesti', 97), ('Sibiu', 80)],
    'Fagaras': [('Bucharest', 211),('Sibiu', 99)],
    'Pitesti': [('Bucharest', 101),('Craiova', 138),('Rimnicu Vilcea', 97)],
    'Bucharest': [('Fagaras', 211), ('Giurgiu', 90), ('Pitesti', 101), ('Urziceni', 85)],
    'Giurgiu': [('Bucharest', 90)],
    'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],
    'Hirsova': [('Eforie', 86),('Urziceni', 98)],
    'Eforie': [('Hirsova', 86)],
    'Vaslui': [('Iasi', 92),('Urziceni', 142)],
    'Iasi': [('Neamt', 87), ('Vaslui', 92)],
    'Neamt': [('Iasi', 87)]
}


def ucs_search(graph, start, goal):

    frontier = []
    counter = 0  # 用于打破代价相同时的平局
    heapq.heappush(frontier, (0, counter, start, [start]))
    counter += 1

    # 已探索集合
    explored = set()

    # 记录到达每个节点的最小代价
    best_cost = {start: 0}

    while frontier:
        # 从优先队列中取出代价最小的节点
        current_cost, _, current_node, path = heapq.heappop(frontier)

        print(f"当前扩展节点: {current_node}, 累计代价: {current_cost}, 当前路径: {' -> '.
    join(path)}")

```

```
        # 目标测试（当节点从队列中弹出时进行）
        if current_node == goal:
            print("-" * 50)
            print(f"OK")
            return path, current_cost, explored

        # 将当前节点加入已探索集合
        if current_node not in explored:
            explored.add(current_node)

            # 扩展当前节点的所有邻居
            for neighbor, distance in graph.get(current_node, []):
                new_cost = current_cost + distance
                new_path = path + [neighbor]

                # 如果邻居未被探索
                if neighbor not in explored:
                    # 如果找到更优路径或首次遇到该节点
                    if neighbor not in best_cost or new_cost < best_cost[neighbor]:
                        best_cost[neighbor] = new_cost
                        heapq.heappush(frontier, (new_cost, counter, neighbor, new_path))
                        counter += 1
                        print(f"  将节点 {neighbor} 加入优先队列 (代价: {new_cost})")

    print("未找到路径! ")
    return None, float('inf'), explored


if __name__ == "__main__":
    start_city = 'Arad'
    goal_city = 'Bucharest'

    print("开始UCS搜索")

    path, total_cost, explored = ucs_search(graph, start_city, goal_city)

    if path:
        print(f"\n路径: {' -> '.join(path)}")
        #print(f"路径长度: {len(path)} 个节点")
        print(f"总代价: {total_cost}")
        #print(f"已探索节点数: {len(explored)}")
        print(f"已探索节点: {sorted(explored)}")

        # # 计算路径详细代价
        # print(f"\n路径代价详情:")
        # path_cost = 0
        # for i in range(len(path) - 1):
        #     city1 = path[i]
```

```
94          #     city2 = path[i + 1]
95          #     # 查找边的代价
96          #     for neighbor, cost in graph[city1]:
97          #         if neighbor == city2:
98          #             path_cost += cost
99          #             print(f"  {city1} -> {city2}: {cost}")
100         #             break
101         # print(f"总代价: {path_cost}")
102     else:
103         print(f"\n从 {start_city} 到 {goal_city} 没有找到路径")
```