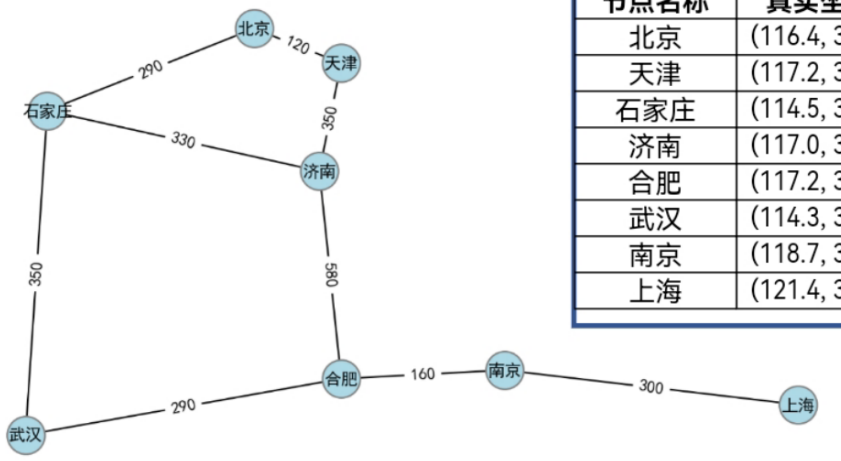
# 有信息搜索实验报告

姓名：_____ 张韫译萱_____　　学号：_____08023214_____

## 一、　实验题目




| 节点名称 | 真实坐标 |
|---|---|
| 北京 | (116.4, 39.9) |
| 天津 | (117.2, 39.1) |
| 石家庄 | (114.5, 38.0) |
| 济南 | (117.0, 36.6) |
| 合肥 | (117.2, 31.8) |
| 武汉 | (114.3, 30.5) |
| 南京 | (118.7, 32.0) |
| 上海 | (121.4, 31.2) |

图 1: 城市路网图及节点坐标

### 1　实验任务

1. 使用贪婪算法（Greedy Search）求出从石家庄到上海的路径，给出经过的节点，并计算出步骤成本。

2. 使用 $A*$ 算法求出从石家庄到上海的路径，给出经过的节点，并计算出步骤成本。

### 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。

3. 记录完整的搜索过程，包括节点扩展顺序、评估函数值等。

# 二、 实验结果

## 1 贪婪算法运行结果

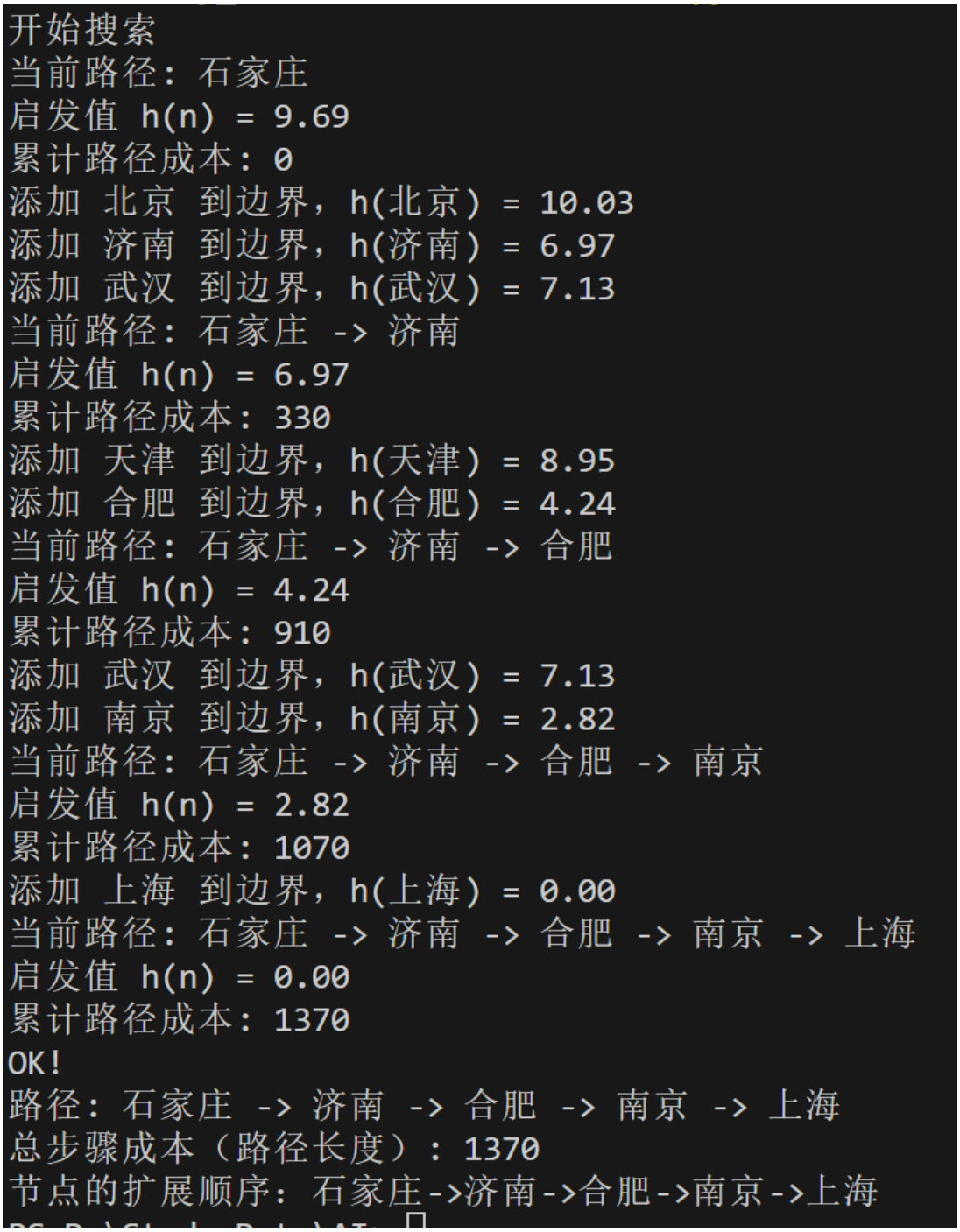
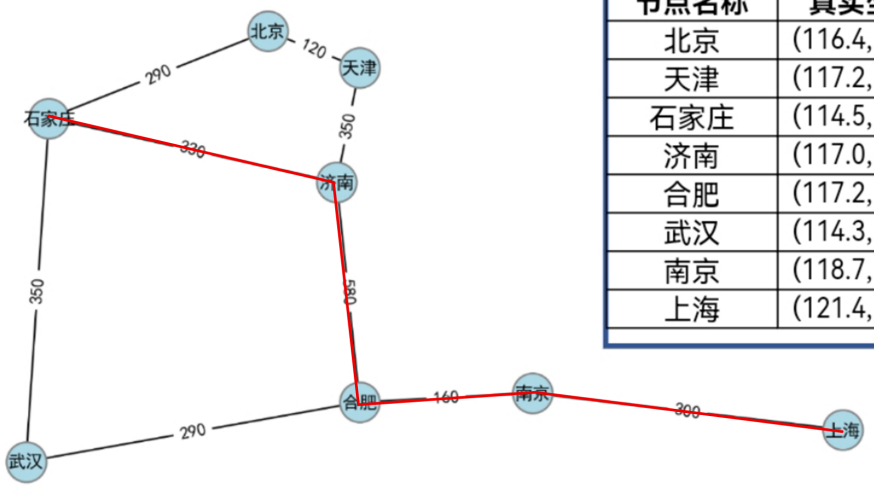



图 2: 贪婪算法运行结果



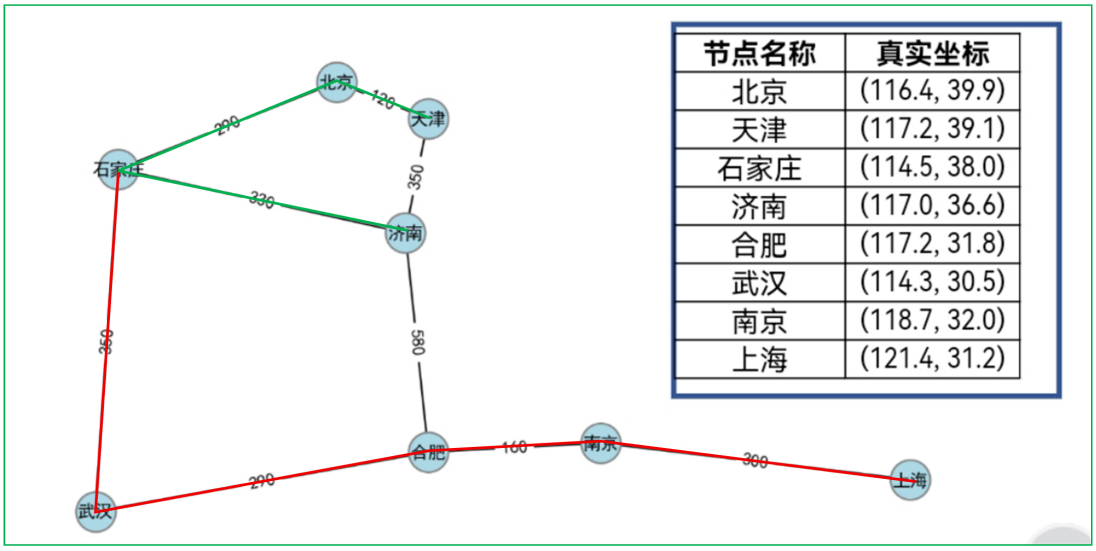

| 节点名称 | 真实坐标 |
|---|---|
| 北京 | (116.4, 39.9) |
| 天津 | (117.2, 39.1) |
| 石家庄 | (114.5, 38.0) |
| 济南 | (117.0, 36.6) |
| 合肥 | (117.2, 31.8) |
| 武汉 | (114.3, 30.5) |
| 南京 | (118.7, 32.0) |
| 上海 | (121.4, 31.2) |

图 3: 贪婪算法搜索过程可视化（红色为路径）

## 2 A* 算法运行结果

```
扩展石家庄,当前路径: 石家庄
添加 北京,f(北京) = 300.03
添加 济南,f(济南) = 336.97
添加 武汉,f(武汉) = 357.13
扩展北京,当前路径: 石家庄 -> 北京
添加 天津,f(天津) = 418.95
扩展济南,当前路径: 石家庄 -> 济南
添加 合肥,f(合肥) = 843.24
扩展武汉,当前路径: 石家庄 -> 武汉
添加 合肥,f(合肥) = 644.24
扩展天津,当前路径: 石家庄 -> 北京 -> 天津
扩展合肥,当前路径: 石家庄 -> 武汉 -> 合肥
添加 南京,f(南京) = 802.82
扩展南京,当前路径: 石家庄 -> 武汉 -> 合肥 -> 南京
添加 上海,f(上海) = 1100.00
扩展合肥,当前路径: 石家庄 -> 济南 -> 合肥
扩展上海,当前路径: 石家庄 -> 武汉 -> 合肥 -> 南京 -> 上海
OK!
最终结果:
最优路径: 石家庄 -> 武汉 -> 合肥 -> 南京 -> 上海
总步骤成本（路径长度）: 1100
节点搜索顺序: 石家庄->北京->济南->武汉->天津->合肥->南京->上海
```

图 4: A* 算法运行结果




| 节点名称 | 真实坐标 |
|---|---|
| 北京 | (116.4, 39.9) |
| 天津 | (117.2, 39.1) |
| 石家庄 | (114.5, 38.0) |
| 济南 | (117.0, 36.6) |
| 合肥 | (117.2, 31.8) |
| 武汉 | (114.3, 30.5) |
| 南京 | (118.7, 32.0) |
| 上海 | (121.4, 31.2) |

图 5: A* 算法搜索过程可视化（红色为路径）

# 三、 实验分析

## 1 算法原理分析

贪婪算法采用启发式函数 $h(n)$ 来评估节点优先级，其中 $h(n)$ 定义为当前节点到目标节点的距离：

$$h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2}$$

算法在每一步选择 $h(n)$ 值最小的节点进行扩展，即优先扩展距离目标最近的节点。

基于图搜索的贪婪算法具有完备性，但不具备最优性。其时间复杂度为 $O(b^m)$，空间复杂度为 $O(b^m)$（b 为分支数，m 为搜索空间的最大深度）。

$A^*$ 算法使用评估函数

$$f(n) = g(n) + h(n)$$

其中 $g(n)$ 为从起点到当前节点的实际路径成本。算法根据以上两者相加后的值进行决策。

当图中启发式函数的值满足一致性（$h(n) \leq c(n, q, n') + h(n')$，即对于每个节点 $n, n'$，从节点 n 到达目标的启发函数值小于等于从 n 到 n' 的代价与从 n' 到达目标的启发函数值之和）时，基于图搜索的 $A*$ 算法具有完备性和最优性。$A*$ 算法的时间复杂度和空间复杂度依赖于对目标状态空间的假设（一般为指数级）。一般情况下，贪婪算法的时间复杂度和空间复杂度低于 $A^*$ 算法。

## 2 实际效果分析

实际搜索过程中，贪婪算法找到的路径总成本为 1370，而 $A*$ 算法找到的路径总成本为 1100，为最优解，验证了 $A*$ 算法的最优性。贪婪算法虽然每一步都选择看似最优的节点（距离目标最近），但由于只考虑启发值而忽略实际路径成本，导致陷入局部最优，最终得到的并非最优解。

对两种算法扩展的节点数目进行分析，贪婪算法仅扩展了 5 个节点即找到解，而 $A*$ 算法扩展了 9 个节点。这验证了算法原理分析中对两种算法进行时空复杂度分析的结论。

对两种算法的算法机制进行分析，贪婪算法的决策依据是 $h(n)$，这使其具有强烈的目标导向性，能够快速朝目标方向前进。然而，这种策略忽略了已经走过的路径成本，导致最终得到的路径并非一定最优。$A*$ 算法则平衡了两方面因素：$g(n)$ 确保不会选择已经累积高成本的路径，$h(n)$ 提供目标导向。这种平衡使 $A*$ 在保证最优性的同时，仍能借助启发信息提高搜索效率。

根据以上的分析，可确定两种算法的基本使用场景：当问题规模较大且对解的最优性要求不高时，贪婪算法是一个高效的选择。而当需要确保找到最优解时，$A*$ 算法是更好的选择。但基于图搜索的 $A*$ 算法的最优性依赖于启发函数的一致性，否则不保证最优性。经验证，本实验中的地图满足这一条件。

# 四、 实验总结

通过本次实验，我深入理解了有信息搜索算法的原理和应用。贪婪算法和 $A*$ 算法都属于有信息搜索算法，利用领域知识来引导搜索过程，相比无信息搜索算法，具有更高的效率。

# 附录

## 1 贪婪算法代码（Python）

```python
import math
import heapq

graph = {
    '北京': [('天津', 120), ('石家庄', 290)],
    '天津': [('北京', 120), ('济南', 350)],
    '石家庄': [('北京', 290), ('济南', 330), ('武汉', 350)],
    '济南': [('天津', 350), ('石家庄', 330), ('合肥', 509)],
    '合肥': [('济南', 509), ('武汉', 290), ('南京', 160)],
    '武汉': [('石家庄', 350), ('合肥', 290)],
    '南京': [('合肥', 160), ('上海', 300)],
    '上海': [('南京', 300)]
}

coordinates = {
    '北京': (116.4, 39.9),
    '天津': (117.2, 39.1),
    '石家庄': (114.5, 38.0),
    '济南': (117.0, 36.6),
    '合肥': (117.2, 31.8),
    '武汉': (114.3, 30.5),
    '南京': (118.7, 32.0),
    '上海': (121.4, 31.2)
}

def euclidean_distance(node1, node2):#两点间距离
    x1, y1 = coordinates[node1]
    x2, y2 = coordinates[node2]
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def greedy_search(graph, start, goal):
    frontier = []
    initial_h = euclidean_distance(start, goal)
    heapq.heappush(frontier, (initial_h, start, [start], 0))

    # 已探索节点集合
    explored = []

    print("开始搜索")

    while frontier:
        # 弹出启发值最小的节点（贪婪策略）
        current_h, current_node, path, path_cost = heapq.heappop(frontier)
```

```
44
45          expansion_count += 1
46          print(f"当前路径: {' -> '.join(path)}")
47          print(f"启发值 h(n) = {current_h:.2f}")
48          print(f"累计路径成本: {path_cost}")
49
50          # 检查是否到达目标
51          if current_node == goal:
52              explored.append(current_node)
53              print("OK!")
54              print(f"路径: {' -> '.join(path)}")
55              print(f"总步骤成本（路径长度）: {path_cost}")
56              print(f"节点的扩展顺序: {'->'.join((explored))}")
57              return path, path_cost
58
59          # 将当前节点标记为已探索
60          if current_node not in explored:
61              explored.append(current_node)
62
63              # 探索邻居节点
64              neighbors = graph.get(current_node, [])
65
66              for neighbor, edge_cost in neighbors:
67                  if neighbor not in explored:
68                      new_path = path + [neighbor] #更新路径列表
69                      new_path_cost = path_cost + edge_cost #更新路径成本
70                      h_value = euclidean_distance(neighbor, goal) #只计算启发式函数的值
71
72                      # 贪婪搜索只使用启发值 h(n) 作为优先级
73                      heapq.heappush(frontier, (h_value, neighbor, new_path, new_path_cost))
74                      print(f"添加 {neighbor} 到边界, h({neighbor}) = {h_value:.2f}")
75
76      # 如果队列为空仍未找到目标
77      print("未找到从起点到目标的路径！")
78      return None, None
79
80  if __name__ == "__main__":
81      # 从石家庄到上海
82      start_city = '石家庄'
83      goal_city = '上海'
84
85      path, cost = greedy_search(graph, start_city, goal_city)
```

# 2 A* 算法代码（Python）

```python
import math
import heapq

graph = {
    '北京': [('天津', 120), ('石家庄', 290)],
    '天津': [('北京', 120), ('济南', 350)],
    '石家庄': [('北京', 290), ('济南', 330), ('武汉', 350)],
    '济南': [('天津', 350), ('石家庄', 330), ('合肥', 509)],
    '合肥': [('济南', 509), ('武汉', 290), ('南京', 160)],
    '武汉': [('石家庄', 350), ('合肥', 290)],
    '南京': [('合肥', 160), ('上海', 300)],
    '上海': [('南京', 300)]
}

coordinates = {
    '北京': (116.4, 39.9),
    '天津': (117.2, 39.1),
    '石家庄': (114.5, 38.0),
    '济南': (117.0, 36.6),
    '合肥': (117.2, 31.8),
    '武汉': (114.3, 30.5),
    '南京': (118.7, 32.0),
    '上海': (121.4, 31.2)
}

def euclidean_distance(node1, node2):#计算两点间距离

    x1, y1 = coordinates[node1]
    x2, y2 = coordinates[node2]
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def astar_search(graph, start, goal):
    # f(n) = g(n) + h(n)
    initial_g = 0
    initial_h = euclidean_distance(start, goal)
    initial_f = initial_g + initial_h
    frontier = []
    heapq.heappush(frontier, (initial_f, initial_g, start, [start]))

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

    # 已探索节点集合
    explored = []

```

```
    # 扩展节点计数器
    expansion_count = 0

    while frontier:
        # 弹出 f 值最小的节点
        current_f, current_g, current_node, path = heapq.heappop(frontier)

        expansion_count += 1
        current_h = euclidean_distance(current_node, goal)

        print(f"扩展{path[-1]},当前路径: {' -> '.join(path)}")


        if current_node == goal:  # 到目标就停止
            explored.append(current_node)
            print("OK!")
            print(f"最终结果: ")
            print(f"最优路径: {' -> '.join(path)}")
            print(f"总步骤成本（路径长度）: {current_g}")
            print(f"节点搜索顺序: {'->'.join(explored)}")
            return path, current_g

        # 将当前节点标记为已探索
        if current_node not in explored:
            explored.append(current_node)

            # 探索邻居节点
            neighbors = graph.get(current_node, [])

            for neighbor, edge_cost in neighbors:
                new_g = current_g + edge_cost

                # 如果找到更好的路径，或者节点未访问过，扩展节点
                if neighbor not in best_cost or new_g < best_cost[neighbor]:
                    best_cost[neighbor] = new_g
                    new_path = path + [neighbor]
                    h_value = euclidean_distance(neighbor, goal)
                    f_value = new_g + h_value

                    heapq.heappush(frontier, (f_value, new_g, neighbor, new_path))
                    print(f"添加 {neighbor},f({neighbor}) = {f_value:.2f}")

    print("未找到从起点到目标的路径！")
    return None, None

if __name__ == "__main__":
    start_city = '石家庄'
    goal_city = '上海'
```

```
94
95      path, cost = astar_search(graph, start_city, goal_city)
```