华东师范大学数据科学与工程学院实验报告

课程名称：AI 基础　　**年级**：2022 级　　**上机实践日期**：2024 年 3 月 25 日

指导教师：杨彬　　**姓名**：朴祉燕

上机实践名称：实验2，project1　　**学号**：10224602413

一、 实验任务

1.小任务：画出狼草羊农民过河问题的状态空间图

2. 算法回顾：图最短路问题，有测评机

- 2.1: BFS（每条边的权重为 1）

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环。所有边的长度(权重)都是 1，点的编号为 1~n。请你求出 1 号点到 n 号点的最短距离，如果从 1 号点无法走到 n 号点，输出-1。

- 2.2: 朴素版 Dijkstra（图最短路）

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为正值。请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出-1。

输入格式：第一行包含整数 n 和 m。接下来 m行每行包含三个整数 x,y,z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式：输出一个整数，表示 1 号点到 n 号点的最短距离。如果路径不存在，则输出-1。

数据范围：1≤n≤ 500，1<m≤10^5，图中涉及边长均不超过 10000。

- 2.3: 堆优化版 Dijkstra（图最短路）

给定一个几个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为正值。请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到几号点，则输出一 1。

输入格式：第一行包含整数 n 和 m。接下来 m 行每行包含三个整数 x,y,z，表示存在一条从 x 点到点 y 的有向边，边长为 z。

输出格式：输出一个整数，表示 1 号点到 n 号点的最短距离。如果路径不存在，则输出一 1。

数据范围：1≤n,m< 1.5 x10^5，图中涉及边长均不小于 0，且不超过 10000。数据保证: 如果最短路存在，则最短路的长度不超过 10^9

3. 八数码问题：经典搜索问题，有测评机

- 3.1: DFS（解存在性问题）
- 3.2: BFS（求解最少步数）
- 3.3: Dijkstra（特殊的 A star）

- 3.4: A star（求解最少步数）

4. 迷宫问题：该问题需要进行Presentation，可视化代码已给（无测评机）

- 4.1: DFS
- 4.2: BFS
- 4.3: Dijkstra
- 4.4: A star
- 补全格子染色的可视化代码

给定一个 n x m 的二维整数数组，用来表示一个迷宫，数组中只包含 0 或 1，其中 0 表示可以走的路，1 表示不可通过的墙壁。

最初，有一个人位于左上角(1,1)处，已知该人每次可以向上、下、左、右任意一个方向移动一个位置,请问，该人从左上角移动至右下角(n,m)处，至少需要移动多少次。数据保证(1,1)处和(n,m)处的数字为 0，且一定至少存在一条通路。

输入格式

第一行包含两个整数 n 和 m。

接下来 n 行，每行包含个整数(0 或 1)，表示完整的二维数组迷宫，

输出格式

输出一个整数，表示从左上角移动至右下角的最少移动次数，

数据范围

1 ≤n,m ≤ 100

二、 使用环境

Python、pycharm、visio 画图工具

三、 实验过程

第三周，我们主要学习了课本中的第三章：。
下面是实验进行过程：

1. 狼草羊过河问题：

一个农民要带着一只狼、一只羊、一颗白菜过河。人不在的时候，狼会吃羊、羊会吃草；猎人每次只能带一样东西过河。

思路：

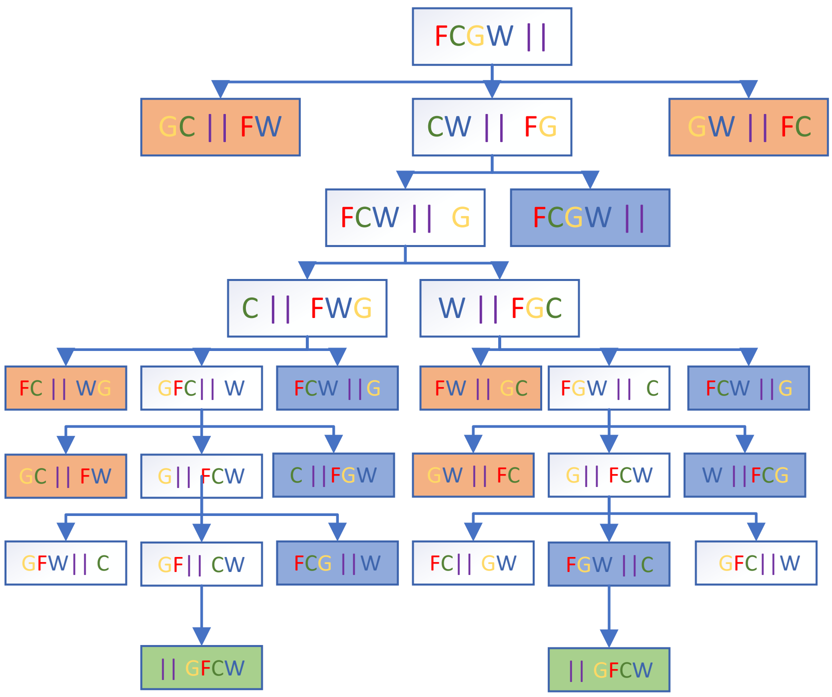
首先，在狼、羊和菜构成的食物链中，关键在于中间的羊，因为狼不吃菜。为了安全地过河，农夫首先需要带着羊离开，打破这个食物链。然而，计算机无法理解这一关键点，因此我们采用穷举法来解决这一问题，通过穷举搜索找出所有可行的过河方法。

穷举搜索所有状态的结果形成了一棵状态树，根节点是初始状态，叶子节点是问题的解。由于只有农夫能划船，每次只能带一个伙伴过河，同时狼、羊和菜不能同时留在一起，导致许多状态是非法的，从客观上起到了剪枝作用。最后利用状态树进行遍历即可。

状态图如下：

相关表示：F：农民；C：白菜 G：羊；W：狼

蓝色表示重复状态；红色表示非法状态；绿色表示结束状态



2.图最短路问题：

2.1: BFS（每条边的权重为 1）：

思路：首先，由于我们需要通过构建邻接表和使用广度优先搜索算法计算从 1 号点到 n 号点的最短距离，定义函数 shortest_path 来接受三个参数：节点数量 n、边数量 m 和边的列表 edges。

在函数内部，首先创建一个空的邻接表 graph 来表示有向图的结构，其中 graph[i] 存储了从节点 i 出发可以到达的节点列表。之后遍历输入的边列表 edges，将边信息添加到邻接表 graph 中相应的位置。

接着初始化一个距离数组 dist，用来记录从节点 1 到各个节点的最短距离。初始时，除了节点 1 的距离为 0，其余节点的距离均设为 -1 表示未访问过。

使用一个队列 queue 来进行广度优先搜索（BFS）。将起始节点 1 加入队列，并开始循环处理队列中的节点。

对于队首节点 node，遍历其邻接节点，如果邻接节点的距离尚未被更新，则更新其距离为当前节点的距离加上 1，并将该邻接节点加入队列中以便后续处理。

不断循环直到队列为空，所有可达节点的最短距离都会被计算出来。

最后返回 dist[n]，即节点 1 到节点 n 的最短距离。

2.2 朴素版 Dijkstra

思路：根据输入的有向图数据，利用朴素版的 Dijkstra 算法来求解从 1 号点到 n 号点的最短距离。在朴素版 Dijkstra 算法中，直接遍历未访问过的节点来找到当前最短路径，不使用堆优化。

首先，同样需要初始化距离数组和访问标记数组：创建一个距离数组 dist 来存储从起始节点到各个节点的最短距离，初始化为无穷大。创建一个访问标记数组 visited 来记录节

点是否被访问过，初始状态为全未访问。

构建图的邻接表：根据给定的边的信息，构建图的邻接表表示，使用字典来表示邻接表，字典的键为节点编号，值为与其相邻的节点及对应的边权重。

之后执行 Dijkstra 算法：从起始节点开始，重复进行以下步骤：找出未访问节点中距离起始节点最近的节点，并将其标记为已访问。更新与该节点相邻未访问节点的最短距离，如果经过当前节点到达相邻节点的距离比原先记录的距离更短，则更新对应节点的最短距离。

2.3 堆优化版 Dijkstra（图最短路）

思路：通过堆优化的 Dijkstra 算法来求解从 1 号点到 n 号点的最短距离。利用构建邻接表、维护距离数组和优先队列，找到最短路径长度。

同样的，创建一个空字典 graph，用于存储有向图的邻接表结构。遍历输入的边数据 edges，将每条边的起点、终点和权重加入到对应起点的邻接列表中。之后初始化一个距离数组 dist，用于记录从节点 1 到各个节点的当前最短距离。初始时，除了节点 1 的距离为 0，其余节点的距离均设为无穷大表示未访问过。

之后创建一个优先队列 pq，使用 heapq 实现。将起始节点 1 加入队列，并开始循环处理队列中的节点。对于队首节点 node，取出其当前距离 d，遍历其邻接节点。如果通过当前节点 node 到达邻接节点的距离比已知的最短距离小，则更新邻接节点的距离为当前距离加上边权，并将该邻接节点加入优先队列以便后续处理。不断循环直到队列为空或者找到最短路径的终点 n，所有可达节点的最短距离都会被计算出来。最后返回 dist[n]，即节点 1 到节点 n 的最短距离。如果最终 dist[n] 仍为无穷大，则表示无法从 1 号点到达 n 号点，输出 -1。

3.八数码问题：

3.1 DFS（解存在性问题）

首先，定义 is_solvable(board) 函数：这个函数用于判断是否有解。使用两层循环来计算空白格从初始状态到目标状态的奇偶性，其中外层循环遍历了数组中的每个元素，内层循环遍历了当前元素之后的所有元素。因此，其时间复杂度为 O(n^2)，其中 n 是状态数组的长度（在这个问题中是 9）。没有使用额外的空间，因此其空间复杂度为 O(1)。

dfs(board, depth, x_pos, visited, max_depth) 函数：这是主要函数进行深度优先搜索算法。在这个函数中，主要的时间复杂度在于递归调用和状态转移的操作。由于在每个状态中都进行了四次状态转移尝试，因此总体时间复杂度取决于状态转移的次数和递归深度。在最坏情况下，时间复杂度为 O(4^d)，其中 d 是最短路径的长度。空间复杂度主要取决于递归调用的栈空间和已访问状态的数量，无伤大雅。综合来看，这个函数的时间复杂度为 O(4^d)，空间复杂度为 O(b*d)，其中 b 是状态转移的分支因子。

最后通过 main 函数调用上面两个函数进行求解。

3.2: BFS（求解最少步数）

这次使用广度优先搜索（BFS）算法来解决八数码问题。

solve_puzzle(grid) 函数： 首先计算了初始状态和目标状态的逆序数，时间复杂度为 O(n^2)。然后，在使用 A* 算法时，主要的时间和空间复杂度集中在队列的操作上。在最坏情况下，队列中的状态数可以达到 O(b^d)，其中 b 是分支因子（每个状态的可能后继数量），d 是最短路径的长度。由于八数码问题的状态空间是有限的，因此可以认为在实际情况下，复杂度是可接受的。而空间复杂度主要取决于队列的大小和已访问状态的数量，因此也是可接受的。综合来看，这个函数的时间复杂度为 O(b^d)，空间复杂度为 O(b^d)。

3.3: Dijkstra

与 BFS 类似，都是用于在图中找到从起点到达目标节点的最短路径，并且都使用了队列（或优先队列）来管理待处理的节点。简单概括就是“dijkstra 是 bfs 的升级版，就是说如果

求最短路径，当图从无权值变成有权值时，bfs 不再适用了，于是我们用 dijkstra 方法。换句话说，对于无权值图，dijkstra 方法跟 bfs 是一致的。”

BFS 的搜索过程中，每个节点的距离都是相等的，而 Dijkstra 算法在搜索过程中会根据已知的最短路径更新节点的距离。BFS 不考虑路径上的权值，只关心路径的长度，因此它对于无权图和权值相同的图更为适用；而 Dijkstra 算法考虑了路径上的权值，可以求解带有权值的图的最短路径。

在时间复杂度上，如果边的数量为 E，节点数量为 V，BFS 的时间复杂度为 O(V+E)，而 Dijkstra 算法的时间复杂度为 O((V+E)logV)，其中 logV 是优先队列的插入和删除操作的复杂度。

3.4: A star（求解最少步数）

A*算法是一种启发式搜索算法，它综合考虑了两个因素：从起始状态到当前状态的实际代价（已经花费的移动步数）和从当前状态到目标状态的预测代价（启发函数）。

count_inversions(arr): 用于判断是否有解。

时间复杂度：O(n^2)，其中 n 是状态数组的长度（在这个问题中是 9）。

空间复杂度：O(1)，没有使用额外的空间。

heuristic(state): 这个函数计算当前状态到目标状态的曼哈顿距离作为启发函数。曼哈顿距离是两个点在一个方格网格上的距离，通过在两个点的水平和垂直距离上相加而得到。

时间复杂度：O(n)，其中 n 是状态数组的长度。

空间复杂度：O(1)，没有使用额外的空间。

move(state, direction): 这个函数根据给定的方向移动空格，并返回移动后的状态。

时间复杂度：O(1)，移动空格需要常数时间。

空间复杂度：O(n)，创建了一个新的状态数组。

astar(start): 使用 A*算法解决问题。先使用一个优先队列来存储待探索的状态，每次从队列中选择启发函数值最小的状态进行探索，并根据启发函数值和已经花费的代价来计算优先级。如果找到了目标状态，则返回移动路径，否则返回"unsolvable"。

时间复杂度：在最坏情况下，A*算法的时间复杂度是指数级的，但在实践中，通过使用启发式函数，通常能够有效地减少搜索空间，使得算法的实际运行时间更接近于线性。具体时间复杂度取决于启发函数的效率和问题的复杂度。

空间复杂度：O(b^d)，其中 b 是分支因子（每个状态的可能后继数量），d 是最短路径的长度。在八数码问题中，b 和 d 都是有界的，因此空间复杂度是可接受的。

至于输入处理部分和计算部分，输入处理部分仅仅对输入进行处理，时间和空间复杂度都是常数级别的；解决八数码问题的部分调用了 A 算法函数，其时间和空间复杂度由 A 算法决定。

4. 迷宫问题：该问题需要进行Presentation，可视化代码已给（无测评机）

4.1: DFS

visualize_maze_with_searched() 函数：可视化迷宫搜索的过程。接收迷宫地图 maze、搜索过的位置 searched、以及最终路径 path 作为输入，然后利用 matplotlib 库将迷宫以灰度图的形式展示出来。在展示迷宫时，搜索过的位置会以浅灰色显示，而最终路径则以红色标记出来。

主循环中的 dfs_search() 函数：实际执行深度优先搜索的函数，采用堆栈（stack）来模拟递归的过程。首先，将起始位置 start 以及从起始位置到该位置的路径 [start] 入栈。然后，不断从堆栈中弹出当前位置 current 和到达当前位置的路径 path，并检查当前位置是否是目标位置 goal。如果是目标位置，则返回搜索过的位置 visited 和最终路径 path。如果不是目标位置，则遍历当前位置的相邻位置，检查是否符合条件（未越界、可通行、未访问

过），如果符合条件，则将相邻位置加入到堆栈中，并将其标记为已访问。搜索过程将在找到目标位置或堆栈为空时结束。

最后，在主函数中调用 dfs_search() 函数进行搜索，并打印出最终路径的长度（即路径中的步数）。然后调用 visualize_maze_with_searched() 函数将迷宫搜索的过程可视化出来，以便更直观地观察搜索的效果。

4.2: BFS 、4.3: Dijkstra

bfs() 函数：执行广度优先搜索，先接收迷宫地图 maze、起始位置 start 和目标位置 end 作为输入。在函数内部，首先定义了四个方向的偏移量 directions，然后创建了一个双向队列 queue 用于存储待访问的位置。同时，使用与迷宫相同维度的二维数组 visited 来记录每个位置的访问状态，初始化为全零。接着，将起始位置 start 加入队列并标记为已访问。然后开始循环，不断从队列中取出位置，并检查其相邻位置是否符合条件。如果符合条件，则将相邻位置标记为已访问，并加入队列。搜索过程将在找到目标位置或队列为空时结束。

reconstruct_path() 函数：用于根据搜索得到的父节点信息回溯出最短路径。它接收父节点信息 parent、起始位置 start 和目标位置 end 作为输入。从目标位置开始，沿着父节点信息一直回溯到起始位置，构建出最短路径。

visualize_maze_with_paths() 函数：用于可视化迷宫及搜索得到的路径。它接收迷宫地图 maze、访问状态数组 visited 和最短路径 shortest_path 作为输入。首先使用 matplotlib 库绘制迷宫地图，并根据访问状态数组将已访问的位置显示为浅灰色的圆点。然后根据最短路径将路径显示为红色的线条。最后设置图例和网格，并展示出可视化结果。

主函数部分：首先读取输入的迷宫大小和内容，并定义起始位置 start 和目标位置 end。然后调用 bfs() 函数执行广度优先搜索，并获取搜索得到的访问状态数组 visited 和父节点信息 parent。根据父节点信息重构出最短路径，并最终调用 visualize_maze_with_paths() 函数进行可视化展示。

4.4: A star

heuristic() 函数：计算两点之间的曼哈顿距离作为启发式函数的值。曼哈顿距离是指两点在网格状结构中沿水平和垂直方向的距离之和。

a_star_search() 函数：执行 A*搜索算法。它接收迷宫地图 maze、起始位置 start 和目标位置 goal 作为输入。在函数内部，同样先定义四个方向的偏移量 neighbor_directions。然后创建了一个优先级队列 open_set 用于存储待访问的位置，其中每个位置的优先级由启发式函数值和路径代价决定。同时，使用字典 came_from 记录每个位置的父节点信息，以便后续重构路径。另外，使用字典 cost_so_far 记录从起始位置到每个位置的路径代价。在搜索过程中，不断从优先级队列中取出路径代价最小的位置，并更新其相邻位置的路径代价和父节点信息。搜索过程将在找到目标位置或队列为空时结束。

reconstruct_path() 函数：根据搜索得到的父节点信息回溯出最短路径。它接收父节点信息 came_from、起始位置 start 和目标位置 goal 作为输入。从目标位置开始，沿着父节点信息一直回溯到起始位置，构建出最短路径。

visualize_maze_with_path() 函数：可视化迷宫及搜索得到的路径。它接收迷宫地图 maze、最短路径 path 和已搜索的位置集合 searched 作为输入。首先使用 matplotlib 库绘制迷宫地图，并根据已搜索的位置集合将其显示为粉色的圆点。然后根据最短路径将路径显示为红色的线条。最后设置图例和网格，并展示出可视化结果。

主函数部分：首先读取输入的迷宫大小和内容，并定义起始位置 start 和目标位置 goal。然后调用 a_star_search() 函数执行 A*搜索，并获取搜索得到的父节点信息 came_from 和已搜索的位置集合 searched。接着根据父节点信息重构出最短路径，并最终调用 visualize_maze_with_path() 函数进行可视化展示。

5. 机场问题

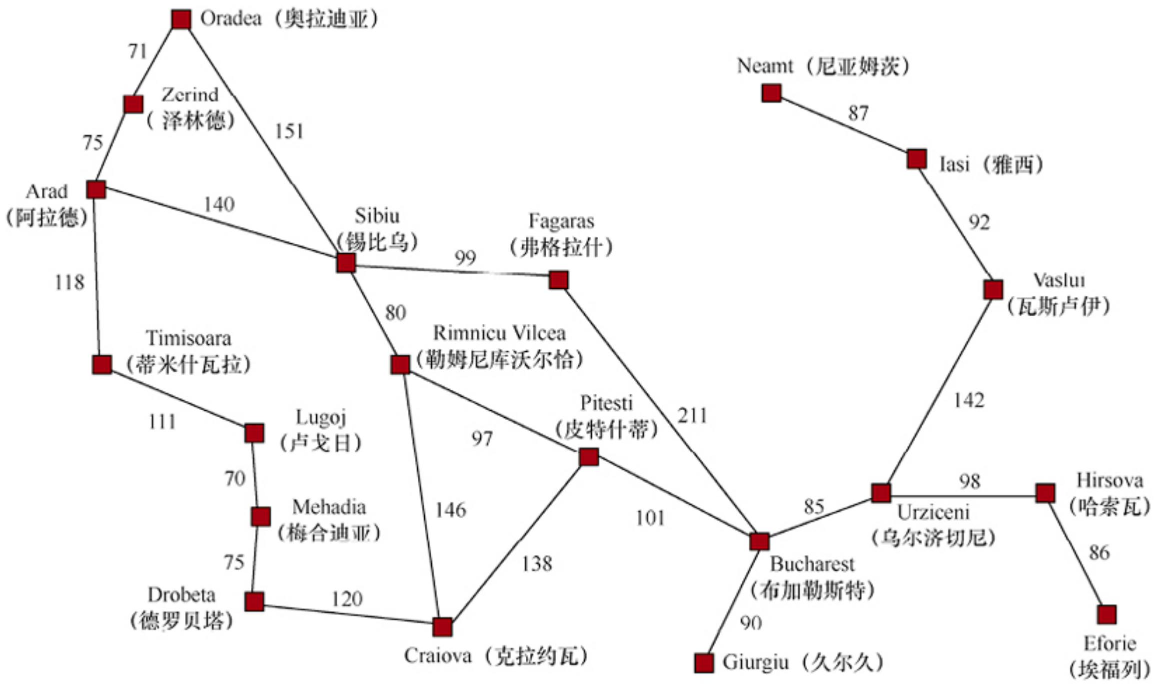
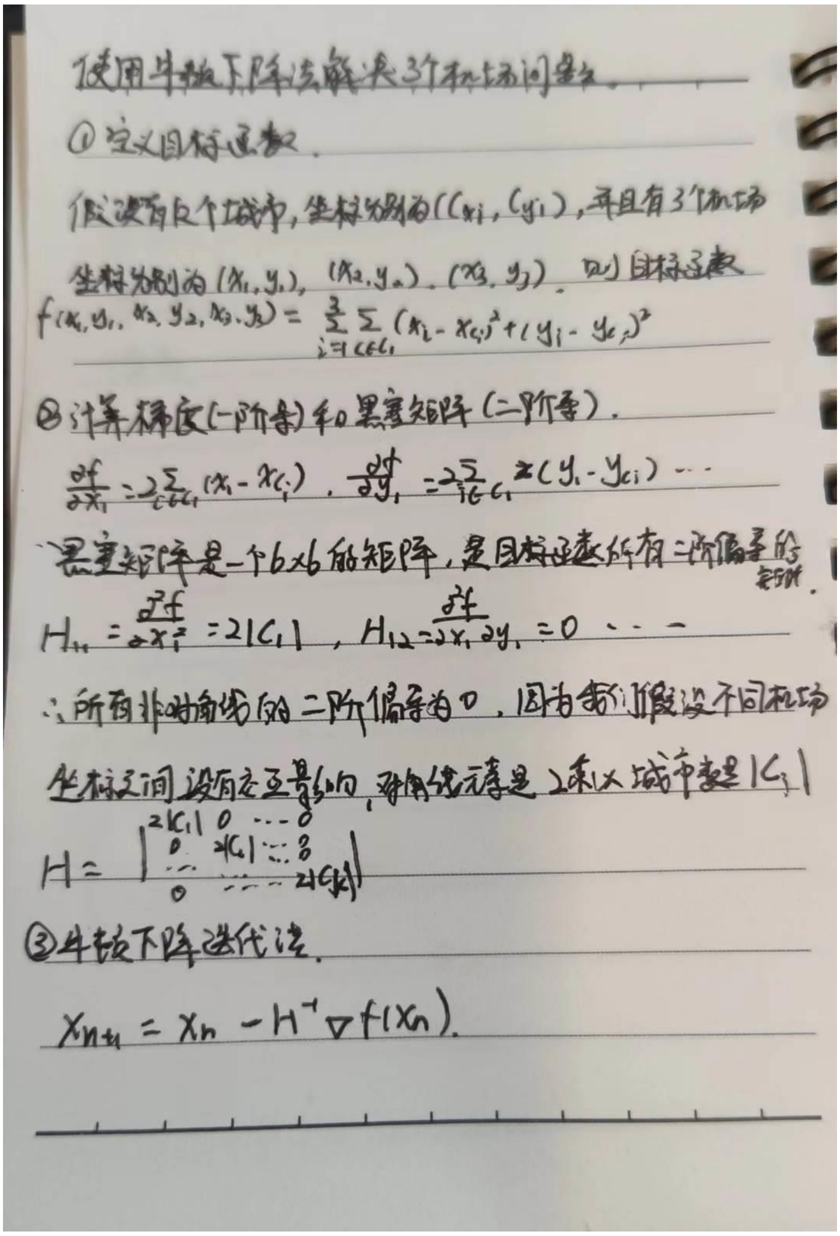


图3-1　罗马尼亚部分地区的简化道路图，道路距离单位为英里（1英里 = 1.61千米）

假设我们希望在罗马尼亚新建 3 个机场，使得地图上每个城市到其最近机场的直线距离平方和最小。（罗马尼亚地图见
图 3-1。）状态空间定义为 3 个机场的坐标：(x1, y1)、(x2, y2)和(x3,y3)。这是一个六维空间；我们也可以说状态由 6 个变量
（variable）定义。一般地，状态定义为 n 维向量，x。在这个空间中移动对应于移动地图上的一个或多个机场。对于任一特定状态，一旦计算出最近城市，目标函数 f (x) = f (x1, y1, x2, y2, x3, y3)的计算就会变得相对容易。设 Ci 是最近机场（在状态 x 下）为机场 i 的城市集合。


使用牛顿下降法解决3个机场问题。

①定义目标函数。

假设有k个城市，坐标分别为(x_i, y_i)，并且有3个机场坐标分别为(x_1, y_1)，(x_2, y_2)，(x_3, y_3)。则目标函数

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_{c_i})^2 + (y_i - y_{c_i})^2$$

②计算梯度(一阶导)和黑塞矩阵(二阶导)。

$$\frac{\partial f}{\partial x_1} = 2\sum_{c \in C_1} (x_1 - x_{c_i}), \quad \frac{\partial f}{\partial y_1} = 2\sum_{c \in C_1} (y_1 - y_{c_i}) \cdots$$

∵黑塞矩阵是一个6x6的矩阵，是目标函数所有二阶偏导的矩阵。

$$H_{11} = \frac{\partial^2 f}{\partial x_1^2} = 2|C_1|, \quad H_{12} = \frac{\partial^2 f}{\partial x_1 \partial y_1} = 0 \cdots$$

∴所有非对角线的二阶偏导为0，因为我们假设不同机场坐标之间没有交互影响，对角线元素是2乘以城市数量$|C_i|$

$$H = \begin{vmatrix} 2|C_1| & 0 & \cdots & 0 \\ 0 & 2|C_1| & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 2|C_3| \end{vmatrix}$$

③牛顿下降迭代法。

$$x_{n+1} = x_n - H^{-1} \nabla f(x_n).$$

四、总结

本次实验对所学的3，4章内容进行了实践性总结。下面我将比对BFS、DFS、A*搜索算法和Dijkstra算法并进行简要总结：

1.BFS（广度优先搜索）：

-BFS是一种图搜索算法，从起始节点开始，逐层向外扩展搜索。

-它使用队列来管理待处理的节点，确保先处理完当前层的节点，再处理下一层的节点。

-适用于无权图或权值相同的图。

-通常用于找到最短路径或层级遍历。

2.DFS（深度优先搜索）：

-DFS是一种图搜索算法，从起始节点开始，沿着一条路径一直向前搜索，直到到达叶子节点或无法继续搜索。

-它使用栈来管理待处理的节点，优先搜索当前路径的最深处。

-适用于无环图或者需要遍历整个图的情况。

-通常用于找到所有可能的路径或状态空间搜索。

3. A*搜索算法：

-A*是一种启发式搜索算法，结合了 BFS 和启发式函数（heuristic function）的优点，可以快速找到最优解。

-它通过评估每个节点的预期代价（通常是从起始节点到该节点的实际代价加上预估的剩余代价），来选择下一个要扩展的节点。

-适用于有权图，能够快速找到最短路径。

-通常用于路径规划、游戏 AI 等领域。

4. Dijkstra 算法：

-Dijkstra 算法是一种贪婪算法，用于在带权图中找到单源最短路径。

-它使用一个优先队列来管理待处理的节点，并根据当前已知的最短路径长度来选择下一个要处理的节点。

-适用于有权图，能够找到起始节点到其他所有节点的最短路径。

-通常用于网络路由、地图导航等领域。

这些算法各有特点，在不同的问题场景中有不同的应用。BFS 和 DFS 是经典的图搜索算法，A*和 Dijkstra 算法则更适用于寻找最短路径问题。