回溯法实验

1 一般原理说明

1.1 回溯法的一般原理

回溯法是一种系统化搜索策略，探索所有可能的路径来寻找问题的解。

其核心思想是从某一状态出发，逐步选择路径并深入探索，直到找到一个满足条件的解，或无法进一步探索时返回上一步并尝试其他路径。算法在当前路径不符合解的要求时，即“走不通”时，及时“退回”到先前的状态，重新选择另一条路径。

其基于**递归思想**，工作机制与**深度优先搜索**（DFS）相似，本质上是按DFS顺序穷举所有解，实际计算时需要和**剪枝策略**结合起来。

1.2 深度优先搜索

回溯法的搜索策略本质上是深度优先的。在解空间中，回溯法会尽可能向下探索每一条可能的路径，直到发现一个有效解或者遇到阻碍。在每一次递归时，算法会从当前节点选择一个路径继续向下深入，直到搜索到一个潜在的解或遇到无法继续的局面时，回溯法会返回上一层，重新选择一个不同的路径。

这种自上而下、逐步探索的策略使得回溯法能够在解空间中尽可能迅速地找到答案，但也可能因为解空间过大而导致较高的时间复杂度。

1.3 剪枝策略

为了优化回溯法的效率，通常会采取剪枝策略来减少不必要的计算。剪枝的目标是在探索过程中尽早发现并终止那些不可能得到解的路径，避免继续深入。常见的剪枝策略包括：

1. **条件剪枝**：当发现某个分支路径的当前状态不符合问题的要求时，立即停止对该路径的进一步探索。例如，如果某个路径已经超出了问题的时间、空间限制，就可以提前“剪掉”该路径。
2. **启发式剪枝**：在某些情况下，可以采用启发式策略，即根据当前节点的信息来**判断**哪些路径更有可能通向解。通过对子节点的排序或者优先选择潜力较大的路径进行探索，可以进一步提高搜索的效率。

通过剪枝策略，回溯法能够**显著减少搜索空间**的大小，避免无效的搜索，从而节省计算资源并加速解的发现。

1.4 回溯法的应用和优势

回溯法是一种通用的解题方法，尤其是在组合优化、约束满足问题和图的搜索等问题中，具有独特的解效应。N皇后问题、图的着色问题、旅行商问题（TSP）等都可以通过回溯法有效解决。回溯法

的优势在于它能够适应解空间大且复杂的场景，尤其是当问题的解具有一定结构性或约束性时，回溯法能显著减少无效计算。

然而，回溯法的缺点也很明显，尤其是在解空间极为庞大的情况下，其时间复杂度往往较高。因此，优化回溯法是一个持续研究的方向，除了剪枝外，结合动态规划、贪心算法等其他优化策略也可以进一步提升效率。

1.5 回溯法与穷举法的关系

回溯法和穷举法（暴力搜索法）有相似之处，都尝试所有可能解来找到问题的答案。但回溯法的不同之处在于，它通过某些条件来提前**排除不必要**的路径，会动态判断每一条路径是否符合问题的要求，避免了不必要的计算。而在穷举法中，算法通常会尝试**所有**的组合或排列。

如果当前路径无法继续，回溯法会及时停止当前路径的探索并返回上一层状态，转而尝试其他路径。换句话说，回溯法通过“剪枝”优化，能够节省大量的计算时间，从而提高效率。

1.6 一般步骤

1.定义解空间的结构

在使用回溯法之前，首先需要明确问题的解空间。

按照我们分析得到的本质，解空间可以被视为一个树形结构，其中每个节点代表问题的一个状态。解空间中至少应包含一个解，且每个解的路径可通过回溯法逐步探索。在定义解空间时，必须明确每个状态之间的转换关系，以便在搜索过程中逐步生成解的候选路径。

2.设计解空间的层次结构

解空间的层次结构需要合理设计，以确保回溯法能够高效地遍历整个解空间。由上一点，我们科研得知，通常情况下解空间被构建成树形结构，每一层代表从初始状态到当前状态的某一次选择过程。每个节点表示某种可能的选择或决策，每条边表示从一个状态到另一个状态的过渡。通过这种层次结构，回溯法可以从根节点开始，逐步向下探索各个可能的路径。

3.执行深度优先搜索

回溯法的核心步骤是执行深度优先搜索（DFS）以遍历整个解空间。在每次递归调用中，回溯算法会沿着当前路径深入，直至找到一个满足条件的解，或者无法继续深入时回退并尝试其他路径。在遍历过程中，回溯法会实时判断当前路径是否符合问题的要求。如果路径不符合条件，则通过剪枝策略提前终止对当前路径的探索，避免无意义的计算。算法会继续执行，直到找到满足条件的解，或者所有路径都被探索过一遍。

---

2 待解决的问题

2.1 旅行商问题 (TSP)

```
1 旅行商问题（Traveling Salesman Problem，TSP）是给定一系列城市及其之间的距离，要求找到
```

一条最短路径，使得旅行商从某个城市出发，经过每个城市恰好一次，并返回到起始城市。

这个问题是一个NP-困难问题，目前没有已知的多项式时间算法可以解决所有问题实例。随着城市数目的增加，所有可能的路径组合呈指数增长。例如，对于 n 个城市，可能的路径数为 $(n-1)!/2$ 。

因此，对于较小的城市数（通常 $n \leq 20$ ），可以使用暴力搜索或动态规划方法；对于较大的城市数，通常使用启发式或近似算法。

2.2 NP 完全问题证明

TSP与哈密尔顿回路问题有着密切的联系，为了方便研究这一类问题，我们先来给问题定性。

2.2.1 旅行商问题与哈密尔顿回路问题的关系

TSP与哈密尔顿回路问题（HAM-CYCLE）有着密切的联系。我们可以通过将哈密尔顿回路问题归约到旅行商问题，来证明旅行商问题是 NP 完全的。

在旅行商问题中，一名商人必须访问 n 个城市。将此问题模型化为一个具有 n 个顶点的完全图，该商人希望进行一次巡回旅行，即经过一个哈密尔顿回路，恰好访问每个城市一次，并最终回到出发城市。商人从城市 i 到城市 j 的旅行费用为一个整数 $c(i,j)$ ，而旅行所需的全部费用是商人旅行经过的所有边的费用之和，商人希望使得整个旅行的费用最低。

与旅行商问题对应的判定问题的形式语言为：

$$TSP = (G, c, k) : G = (V, E)$$

其中， G 是一个完全图， c 是 $V \times V \to \mathbb{Z}$ 上的一个费用函数， $k \in \mathbb{Z}$ 表示一个最大花费为 k 的旅行回路。

2.2.2 证明：旅行商问题是 NP 完全的

首先，说明 TSP 属于 NP 类问题。给定一个问题实例，我们可以通过回路中包含的 n 个顶点序列来证明该问题的解。验证该序列是否恰好包含每个顶点一次，并计算这些边的总费用，最后检查其是否不超过 k 。这些操作都可以在多项式时间内完成，因此，TSP 属于 NP 类。

接下来，我们要证明 TSP 是 NP 难度问题。为此，我们需要证明哈密尔顿回路问题可以多项式时间归约到旅行商问题，即证明 $\text{HAM-CYCLE} \leq_p \text{TSP}$ 。

设 $G = (V, E)$ 是一个哈密尔顿回路问题的实例。我们通过以下步骤构造一个旅行商问题实例：

1. **构造完全图 G'** ：首先建立一个完全图 $G' = (V, E')$ ，其中 $E' = (i,j) : i,j \in V, i \neq j$ ，即所有不同城市之间都有一条边。
2. **定义费用函数 c** ：定义费用函数 $c(i,j)$ 为：

$$c(i,j) = \begin{cases} 0 & \text{若 } (i,j) \in E \\ 1 & \text{若 } (i,j) \notin E \end{cases}$$

这里，E 是原图 G 中的边集合，而 E' 是完全图 G' 中的边集合。由于 G 是无向图，且没有自环，故对于所有顶点 $v \in V$，都有 $c(v, v) = 1$。

3. **构造 TSP 实例**：因此，$(G', c, 0)$ 就是一个 TSP 的实例，它可以通过上述构造在多项式时间内产生。

2.2.3 证明哈密尔顿回路的存在与 TSP 解决方案的关系

现在，我们来证明图 G 具有哈密尔顿回路，当且仅当图 G' 中存在一个费用至多为 0 的回路。

- **如果 G 中有哈密尔顿回路，则 G' 中有费用为 0 的回路**：

假设图 G 中存在一个哈密尔顿回路 h。回路 h 中的每一条边都属于 G 中的边集合 E，因此在 G' 中，这些边的费用为 0。由于哈密尔顿回路恰好访问每个城市一次，且最终回到出发城市，因此 G' 中的回路的总费用为 0。

- **如果 G' 中有费用至多为 0 的回路，则 G 中有哈密尔顿回路**：

反过来，假设在 G' 中存在一个费用至多为 0 的回路 h'。由于 G' 中的每条边的费用只能是 0 或 1，因此回路 h' 的费用为 0，意味着回路 h' 中的每一条边都必须属于原图 G 中的边集合 E。因此，回路 h' 仅包含 G 中的边，这表明 h' 是图 G 中的一个哈密尔顿回路。

2.2.4 结论

通过上述证明，我们可以得出结论，哈密尔顿回路问题能够多项式时间归约到旅行商问题。因此，由于哈密尔顿回路问题是 NP 完全的，我们可以得出结论，旅行商问题也是 NP 完全的。

---

3 算法设计

3.1 回溯法求解

3.1.1 算法设计策略

旅行商问题 (TSP) 是一个典型的 NP 完全问题，它的解空间可以通过一个排列树来表示。每个节点对应于一个旅行商的路径，这个路径表示从起始城市开始，依次访问每个城市，最终回到起点，且不重复任何城市。回溯法 (Backtracking) 是解决该问题的常见方法，它通过深度优先搜索解空间树来找到最优路径。

在这段描述中，`x` 表示一个排列数组，其中 `x[i]` 表示旅行商访问的第 `i` 个城市。解空间树的每一层都表示访问一个城市的选择。

1. 解空间树的结构

旅行商问题的解空间树可以看作一个排列树，其中每个节点表示一个城市，树的深度为 `n`（城市的数量）。初始时，树的根节点表示从起点城市开始的路径，树的每一层对应于选择一个城市，直到所有城市都被访问一次。

- **根节点**：表示起始城市。
- **每个非叶子节点**：表示已访问的城市路径。
- **叶子节点**：表示所有城市都被访问过，且可以通过一条边回到起点城市，形成一个完整的回路。

2. 回溯算法步骤

回溯算法逐层深入树结构，每次选择下一个城市作为当前路径的一部分。在递归的过程中，回溯法会根据当前路径的费用判断是否继续深入或剪枝。

函数初始化：

- `best_cost` 初始化为无穷大，表示当前没有找到有效解。
- `best_solution` 初始化为 `None`，表示没有找到最优路径。

回溯递归函数（`Backtrack(i, current_path, current_cost)`）：

- `i` 表示当前已经访问的城市数。
- `current_path` 是当前已访问城市的路径。
- `current_cost` 是当前路径的旅行费用。

递归终止条件：

- 当 `i == n` 时，意味着所有城市都已访问。此时，检查从最后一个城市到起点的边是否存在。如果存在，则形成一个完整的回路。
- 计算该回路的费用，并与 `best_cost` 比较，如果费用较小，则更新 `best_cost` 和 `best_solution`。

递归分支：

- 当 `i < n` 时，意味着还有城市未访问。通过遍历所有城市 `j`，检查城市 `j` 是否未在当前路径中，且是否存在从当前城市到城市 `j` 的边。如果条件满足，则将城市 `j` 加入路径，并递归进入下一层。
- `new_cost = current_cost + G[current_path[i-1], j]` 表示更新路径的费用。
- **剪枝**：如果当前费用 `new_cost` 小于 `best_cost`，则继续递归。如果不满足条件，直接跳过此分支。

路径更新：

- 如果所有城市都已访问且费用有效，则计算总费用并检查是否需要更新最优解。
- 最终返回 `best_solution` 和 `best_cost`。

3.1.2 回溯函数伪代码

```
Input:
G: 邻接矩阵，表示城市间的旅行费用
n: 城市总数
Output:
best_solution: 最优路径
best_cost: 最优费用

Procedure TSP_Backtrack(G, n):
    Initialize best_cost = infinity
    Initialize best_solution = None

    Define Backtrack(i, current_path, current_cost):
        if i == n:  // 如果已经访问所有城市
            if G[current_path[n-1], current_path[0]] exists:  // 检查是否有从最后一个城市返回到起点的路径
                total_cost = current_cost + G[current_path[n-1], current_path[0]]  // 计算当前路径的总费用
                if total_cost < best_cost:  // 如果当前路径费用优于最优解
                    best_cost = total_cost  // 更新最优费用
                    best_solution = current_path + [current_path[0]]  // 更新最优路径
            return

        for each vertex j from 1 to n:  // 遍历所有城市
            if vertex j is not in current_path and G[current_path[i-1], j] exists:  // 判断当前城市是否已访问过，且是否有从当前城市到城市 j 的路径
                new_cost = current_cost + G[current_path[i-1], j]  // 更新路径的费用
                if new_cost < best_cost:  // 如果当前费用小于最优费用，继续回溯搜索
                    Backtrack(i + 1, current_path + [j], new_cost)  // 递归搜索下一个城市

    Start with the first vertex as the current path: Backtrack(1, [1], 0)  // 从第一个城市开始回溯搜索

    return best_solution, best_cost  // 返回最优路径和最优费用


```

3.1.3 主函数伪代码

```
Algorithm: Main
Input: None
Output: 最优路径和最优费用
```

```
4
5  Procedure Main:
6      // 获取用户输入的城市数量
7      Print "请输入城市数量: "
8      Read n  // 城市数量
9
10     // 获取邻接矩阵
11     Print "请输入邻接矩阵（每行用空格分隔）: "
12     Initialize G as an empty 2D array of size n x n
13     for i = 1 to n:
14         Read row of n numbers into G[i]  // 读取第 i 行并存入矩阵 G
15
16     // 调用回溯算法求解旅行商问题
17     best_solution, best_cost = TSP_Backtrack(G, n)
18
19     // 输出结果
20     Print "最优路径: ", best_solution
21     Print "最优费用: ", best_cost
22
23 End Procedure
```

3.1.4 算法复杂度分析

时间复杂度分析

回溯法的时间复杂度主要由递归搜索的深度和每次递归所进行的操作决定。在最坏情况下，算法需要枚举所有可能的城市排列。对于 `n` 个城市，回溯树的深度为 `n`，每层的分支数为 `n-1`，即每次选择一个尚未访问的城市。因此，回溯树的总节点数为 `O(n!)`，这表示回溯法在最坏情况下会进行 `O(n!)` 次递归调用。

此外，在每次递归中，算法可能需要检查当前路径的费用，并根据条件更新最优路径。每次更新最优路径时，都需要遍历当前路径中的所有城市来计算路径费用，这一过程的时间复杂度是 `O(n)`。因此，在最坏情况下，回溯法需要更新最优解 `O(n!)` 次，每次更新的计算时间是 `O(n)`。

综上所述，回溯法的总体时间复杂度为：

$$O(n!) \times O(n) = O(n!)$$

这表明，回溯法的时间复杂度主要由路径的枚举数量决定，而每次路径的更新只占用较小的时间。

空间复杂度分析

回溯法的空间复杂度由以下几个部分决定：

1. **邻接矩阵**：由于图的表示使用邻接矩阵，存储所有的城市之间的边关系需要占用 `O(n^2)` 的空间。对于 `n` 个城市，邻接矩阵的大小是 `n x n`。

2. **当前路径**：在回溯过程中，当前路径需要存储已访问的城市。由于路径的最大长度为 `n`，因此这部分空间的复杂度为 `O(n)`。
3. **递归调用栈**：由于回溯算法是递归的，每次递归都需要保存当前的状态。最深的递归深度为 `n`，因此递归栈的空间复杂度为 `O(n)`。

综合上述各部分，回溯法的空间复杂度主要由邻接矩阵的存储需求决定，因此总体空间复杂度为：

$$O(n^2)$$

3.2 动态规划求解

3.2.1 最优子结构性质

旅行商问题（TSP）并不具备严格意义上的最优子结构性质，这是因为TSP的解是一个闭环路径，要求经过所有城市一次并返回起点，而路径的选择不仅依赖于当前子路径，还与**后续选择和整体路径的排列顺序**紧密相关。

最优子结构的定义

最优子结构意味着，一个问题的最优解可以通过其子问题的最优解来构建。然而，在TSP中，某段子路径的最优性未必能够直接组合到整体最优解中。这是因为TSP的路径是一个环路，每一段路径的选择都会影响到后续路径的构建，从而对整体最优解产生影响。

TSP中的子路径问题

考虑TSP中的某段路径，比如路径 $1 \to 2 \to 3$，假设在某个子问题中，这条路径是最优的。然而，如果将这段路径直接组合到整体路径中，可能会导致整体路径的最优性受到影响。例如，选择路径 $1 \to 2 \to 3$ 后，后续路径 $4 \to 5 \to 1$ 的总长度可能会增加，从而破坏整体的最优解。这种现象表明，TSP的解并不具备最优子结构，因为子路径的最优性不能直接保证整体路径的最优性。

闭环结构与全局约束

TSP的解是一个闭环路径，即要求从某个城市出发，经过所有城市一次并最终回到起点。在这种情况下，每个城市的选择都会受到全局约束：选择当前城市时，必须考虑后续路径的影响，而不仅仅是局部的路径长度。因此，TSP的路径不仅依赖于当前子路径，还受全局选择顺序的影响。

动态规划与子集最优路径

尽管TSP在严格意义上不具备最优子结构，但它仍然可以通过动态规划的方法来求解。动态规划通过将问题分解成多个子问题来求解，而TSP可以通过考虑每个城市的子集以及访问这些城市的顺序来解决。这种方法不再是简单的“最优子路径”，而是通过逐步计算**不同子集的最优路径**来构建最终解。

动态规划方法的核心思想是**避免重复计算**，通过逐步构建子集的最优解来最终求得整个问题的最优解。虽然子路径的最优解**不能直接**构成全局最优解，但通过巧妙地分解和递推，动态规划能够**有效**地解决TSP问题。

3.2.2 递归算法设计策略

虽然TSP不具备严格意义上的最优子结构性质，但通过动态规划（DP）技术，可以有效地解决该问题。下面是基于动态规划的解决方案的设计方法。

1. 状态定义

首先，我们定义以下状态变量：

- n：城市的数量。
- $a[u][v]$：表示城市 u 到城市 v 的路径长度。如果城市之间没有路径，表示为无穷大，通常用 `Infinite` 表示。
- $dp[S][u]$：表示从起点 0 出发，经过城市集合 S 并最终到达城市 u 的最短路径长度。这里的集合 S 是通过二进制掩码表示的，意味着通过位掩码可以标识已访问的城市。

2. 状态转移

对于一个城市集合 S 和终点 u，最优路径的长度 $dp[S][u]$ 可以通过子问题的最优解递归地构造出来。具体而言，当前状态 $dp[S][u]$ 的值是通过以下步骤递归计算得到的：

- 从集合 $S \setminus \{u\}$ 中的一个城市 v 到达 u 。
- 在此路径上，最优的解就是从起点到 $S \setminus \{u\}$ 中的某个城市 v 的最短路径，再加上从城市 v 到城市 u 的距离。

因此，递归的状态转移方程为：

$$dp[S][u] = \min_{v \in S, v \neq u} \{dp[S \setminus \{u\}][v] + a[v][u]\}$$

其中：

- $dp[S \setminus \{u\}][v]$：表示从起点到城市集合 $S \setminus \{u\}$ 中城市 v 的最短路径长度。
- $a[v][u]$：表示城市 v 到城市 u 的路径长度。

3. 初始条件

在 TSP 的动态规划中，我们需要设置初始状态。当路径只有起点和一个城市时，即集合 S 只包含起点和一个城市 u （即 $S = \{0, u\}$），最短路径就是直接从起点到城市 u 的路径长度。因此，初始条件为：

$$dp[1 << u][u] = a[0][u]$$

其中，$1 << u$ 是表示包含起点和城市 u 的二进制掩码。

4. 位掩码的使用

为了优化空间复杂度和提高计算效率，我们使用位掩码（mask）表示城市的集合。使用一个整数的二进制形式来表示当前访问的城市集合。每一位 i 表示城市 i 是否在当前集合中。例如，$\text{mask} = 00101$ 表示路径经过城市 0 和城市 2 。

通过位运算 $\text{mask} \& (1 << u)$，可以判断城市 u 是否属于集合 S。这种方法使得在递归过程中，不必显式地表示所有城市集合，而是通过一个整数来高效地表示。

5. 目标状态

最终的目标是通过动态规划表计算出所有城市都被访问并回到起点的最短路径。我们通过遍历所有的状态 $dp[(1 << n) - 1][u]$ 来计算出包含所有城市的路径最短距离，并最终回到起点 0 ：

$$\text{result} = \min_{u \neq 0} \{dp[(1 << n) - 1][u] + a[u][0]\}$$

其中 $(1 << n) - 1$ 表示所有城市都已经被访问的状态， $dp[(1 << n) - 1][u]$ 是最后一个城市是 u 的最短路径长度，最后加上从城市 u 返回起点的路径长度 $a[u][0]$ 。

3.2.3 DP 函数伪代码

```
Procedure TSP_DP(a, n, bestx):
    Initialize N = 2^n
    Initialize dp as a 2D array of size N x n, filled with Inf
    Initialize parent as a 2D array of size N x n, filled with -1

    Set dp[1][0] = 0  # Starting point is city 0

    For each mask from 1 to N-1:
        For each u from 0 to n-1:
            If (mask & (1 << u)) != 0 and dp[mask][u] != Inf:
                For each v from 0 to n-1:
                    If (mask & (1 << v)) == 0 and a[u][v] != Inf:
                        next_mask = mask | (1 << v)
                        If dp[next_mask][v] > dp[mask][u] + a[u][v] or
dp[next_mask][v] == Inf:
                            dp[next_mask][v] = dp[mask][u] + a[u][v]
                            parent[next_mask][v] = u

    Set final_mask = (1 << n) - 1
    Initialize min_cost = Inf
    Initialize last_city = -1

    For each u from 1 to n-1:
        If a[u][0] != Inf and dp[final_mask][u] != Inf:
            total_cost = dp[final_mask][u] + a[u][0]
            If total_cost < min_cost or min_cost == Inf:
                min_cost = total_cost
                last_city = u

    If min_cost == Inf:
        Return Inf

    bestx = ReconstructPath(parent, final_mask, last_city, n)

    Return min_cost
```

```
35
```

3.2.4 算法复杂度分析

时间复杂度：

该算法的时间复杂度为 $O(n^2 \cdot 2^n)$，因为对于每个城市集合 S 和每个终点 u，我们需要枚举集合 S 中的每个城市 v，总共进行 $O(n^2)$ 次操作。由于有 2^n 个城市集合，需要进行 $O(2^n \cdot n^2)$ 次操作。

空间复杂度：

空间复杂度为 $O(n \cdot 2^n)$，因为我们需要存储 dp 表格，表格大小为 $n \times 2^n$，即存储每个城市集合和每个终点的最短路径长度。

3.2.5 二者印证

由两个算法的计算结果可知，回溯法与动态规划法的计算结果正确性可以保障。

---

4 拓展实验

4.1 近似求解

4.1.1 贪心算法设计策略

贪心算法直接选择最相近的未访问城市作为下一个移动，迅速产生一个有效较短的路线。尽管在某些情况下表现良好，但在特殊排列的城市分布下，可能得不到最优解。

4.1.2 贪心算法伪代码

```
定义 Infinite 为无穷大（表示无路径）

函数 calculate_cost(path, a):
    初始化 cost 为 0
    对于 path 中的每一对相邻城市 (path[i], path[i+1])，执行:
        cost += a[path[i]][path[i+1]]
    cost += a[path[-1]][path[0]]  # 添加从最后一个城市回到起点的费用
    返回 cost

函数 greedy_algorithm(a, n):
    初始化 visited 数组为长度为 n，所有元素为 False
    初始化 path 数组为 [0]  # 从城市0开始
    标记 visited[0] 为 True  # 标记城市0为已访问
    设置 current_city 为 0  # 当前城市
```

```
    对于 n - 1 次，执行:
        初始化 nearest_city 为 -1
        初始化 min_cost 为 Infinite
        对于每个城市 next_city 在 0 到 n-1 范围内，执行:
            如果 next_city 没有被访问且 a[current_city][next_city] < min_cost:
                设置 nearest_city 为 next_city
                设置 min_cost 为 a[current_city][next_city]
        将 nearest_city 加入 path
        标记 visited[nearest_city] 为 True
        更新 current_city 为 nearest_city

    使用 calculate_cost 函数计算路径的总费用并返回路径和费用

函数 get_adjacency_matrix(n):
    输出提示信息，要求用户输入 n x n 邻接矩阵
    初始化一个空矩阵 a
    对于每一行 i 从 0 到 n-1，执行:
        从用户输入获取该行城市间的旅行费用，并将其转换为整数列表
        将该行添加到矩阵 a 中
    返回 a

主程序:
    输入城市数量 n
    调用 get_adjacency_matrix 获取邻接矩阵 a
    调用 greedy_algorithm 函数获取最优路径和费用
    如果最优费用是 Infinite，输出 "No solution"
    否则，输出最优路径和最优费用
```

4.1.3 时间复杂度

由于每个城市都需要检查当前城市与其他城市之间的距离，因此时间复杂度为 $O(n^2)$。

4.2 模拟蚁群

4.2.1 模拟蚁群算法设计策略

这一方法模仿真实蚂蚁在寻找食物源与巢穴之间最短路径时的行为。这种行为是通过个体蚂蚁偏向跟随其他蚂蚁留下的信息素轨迹而自发产生的。

在模拟蚁群中，首先会派出大量的虚拟``蚂蚁''代理，在地图上探索可能的路径。每只蚂蚁在选择下一座城市时，会基于一个综合启发式规则，该规则结合了城市之间的距离和通往目标城市的路径上

已留下的虚拟信息素的数量。信息素的浓度表示其他蚂蚁曾经偏好该路径的程度。

蚂蚁在探索的过程中会沿着它们经过的每条边上留下信息素，直到所有蚂蚁完成一轮旅行。在这一轮结束后，完成最短路径的那只蚂蚁会在它整个旅行路径上进行``全局轨迹更新''，即沿着它的路径增加额外的虚拟信息素。更新的信息素量与路径长度成反比：路径越短，信息素量越多。

这种模拟过程通过多轮迭代逐步优化路径，最终趋近于全局最优解或接近最优的解。ACS利用了群体智能的特性，通过个体的简单行为和局部信息的累积，涌现出解决复杂问题的能力。

为了更好的模拟蚁群，我们将起点设置为随机选取。

4.2.2 模拟蚁群算法伪代码

算法的伪代码如下所示。

```
初始化常数:
    Infinite ← ∞   // 无路径的无穷大
    Q ← 100       // 信息素常数
    alpha ← 1     // 信息素重要程度因子
    beta ← 2      // 启发函数重要程度因子
    rho ← 0.1     // 信息素挥发因子
    n_ants ← 20   // 蚂蚁数量
    n_iter ← 100 // 最大迭代次数

函数 calculate_cost(path, a):
    初始化 cost ← 0
    对于路径中的每一对城市 (i, i+1):
        cost += a[path[i]][path[i+1]]
    cost += a[path[-1]][path[0]]  // 回到起点
    返回 cost

函数 update_pheromone(pheromone, paths, costs):
    对于每只蚂蚁路径:
        对路径中的每一对城市 (i, i+1):
            pheromone[path[i]][path[i+1]] += Q / costs[i]
            pheromone[path[i+1]][path[i]] += Q / costs[i]  // 对称
    返回更新后的 pheromone

函数 select_path(probabilities):
    生成一个随机数 r
    初始化 total ← 0
    对于每个概率 prob:
        total += prob
        如果 total ≥ r:
            返回当前城市索引
    返回最后一个城市
```

```
函数 ant_colony_algorithm(a, n):
    初始化 pheromone 为 n×n 的矩阵，初始值为 1.0
    初始化 best_cost 为 Infinite
    初始化 best_solution 为 None

    对于每一轮迭代 iteration 从 1 到 n_iter:
        初始化 paths 为一个空列表
        初始化 costs 为一个空列表

        对于每只蚂蚁 ant 从 1 到 n_ants:
            初始化路径 path 为 [随机选择一个城市]
            初始化 visited 集合，记录已经访问的城市

            对于每个剩余城市:
                选择当前城市 current
                初始化 probabilities 为空列表
                对于每个未访问过的城市 next_city:
                    如果 a[current][next_city] 不是无穷大:
                        计算该路径的选择概率 prob
                        将 prob 添加到 probabilities 列表
                归一化概率列表 probabilities

                根据概率列表选择下一城市 next_city
                将 next_city 添加到路径 path
                将 next_city 添加到 visited

            计算路径的总费用 cost
            将路径 path 和费用 cost 添加到 paths 和 costs

            如果 cost < best_cost:
                更新 best_cost 和 best_solution 为当前路径和费用

        更新信息素 pheromone:
            调用 update_pheromone 更新 pheromone

        信息素挥发:
            对 pheromone 矩阵中的每一对城市 (i, j):
                pheromone[i][j] = (1 - rho) * pheromone[i][j]

    返回最优路径 best_solution 和最优费用 best_cost

函数 get_adjacency_matrix(n):
    打印提示：请输入城市数量
    初始化邻接矩阵 a 为一个 n×n 的矩阵
    对于每行城市输入:
        读取该行的城市间费用，填入 a 矩阵中
```

```
    返回邻接矩阵 a

主程序:
    获取城市数量 n
    获取邻接矩阵 a
    调用 ant_colony_algorithm 求解 TSP
    如果 best_cost == Infinite:
        输出 "No solution"
    否则:
        输出最优路径 [best_solution] 和最优费用 best_cost

```

4.2.3 时间复杂度

考虑到每轮迭代中：

- 蚂蚁构建路径的时间复杂度为 `O(n_ants * n)`。
- 信息素更新的时间复杂度为 `O(n_ants * n)`。
- 信息素挥发的时间复杂度为 `O(n^2)`。

总的时间复杂度：

$$O(n\_iter * (n\_ants * n + n^2))$$

---

5 运行结果分析

5.1人工生成 n=10

对于 n = 10，即10个城市，我们手动生成了数据。

5.1.1 精确算法

分别对回溯法和动态规划法进行测试，得到的结果如图所示。

```
实验\yyx\backtrack.py'
请输入城市数量: 10
请输入 10 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 10 行:
0 10 17 20 5 30 35 24 45 50
10 0 25 30 35 40 45 50 55 60
17 25 0 35 40 45 50 55 60 65
20 30 35 0 10 15 20 25 30 35
5 35 40 10 0 5 10 15 20 25
30 40 45 15 5 0 5 10 15 20
35 45 50 20 10 5 0 5 10 15
24 50 55 25 15 10 5 0 5 10
45 55 60 30 20 15 10 5 0 5
50 60 65 35 25 20 15 10 5 0

最优路径: [1, 2, 3, 4, 6, 7, 8, 9, 10, 5, 1]
最优费用: 135
```

回溯法在n=10时的结果

```
实验\yyx\dp.py'
请输入城市数量: 10
请输入 10 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 10 行:
0 10 17 20 5 30 35 24 45 50
10 0 25 30 35 40 45 50 55 60
17 25 0 35 40 45 50 55 60 65
20 30 35 0 10 15 20 25 30 35
5 35 40 10 0 5 10 15 20 25
30 40 45 15 5 0 5 10 15 20
35 45 50 20 10 5 0 5 10 15
24 50 55 25 15 10 5 0 5 10
45 55 60 30 20 15 10 5 0 5
50 60 65 35 25 20 15 10 5 0

最优路径: [1, 5, 10, 9, 8, 7, 6, 4, 3, 2, 1]
最优费用: 135
```

DP法在n=10时的结果

可以看到，二者给出了相同的最优值，最小成本为135，虽然路径不同，但是二者均为最优解。

5.1.2 近似算法

分别对贪心算法和模拟蚁群算法进行测试，得到的结果如图所示。

贪心算法在n=10时的结果

模拟蚁群算法在n=10时的结果

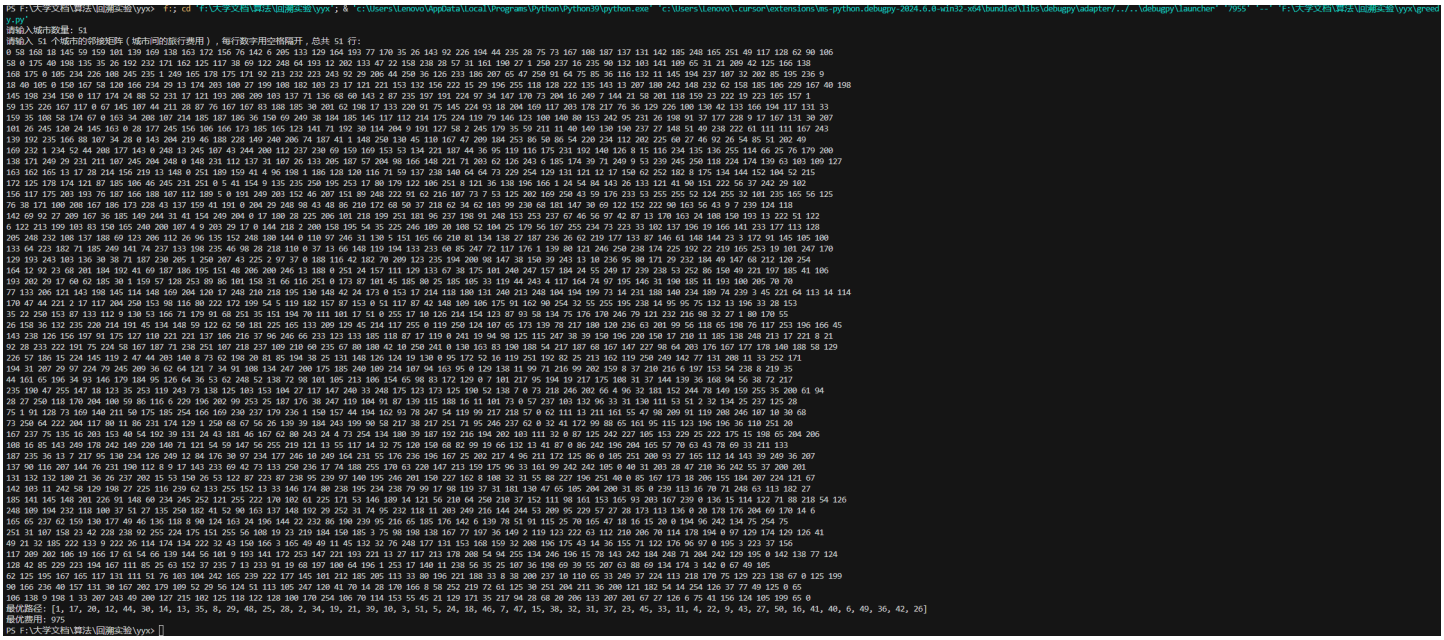
最小成本如下：

1. 贪心算法：137
2. 模拟蚁群算法：135

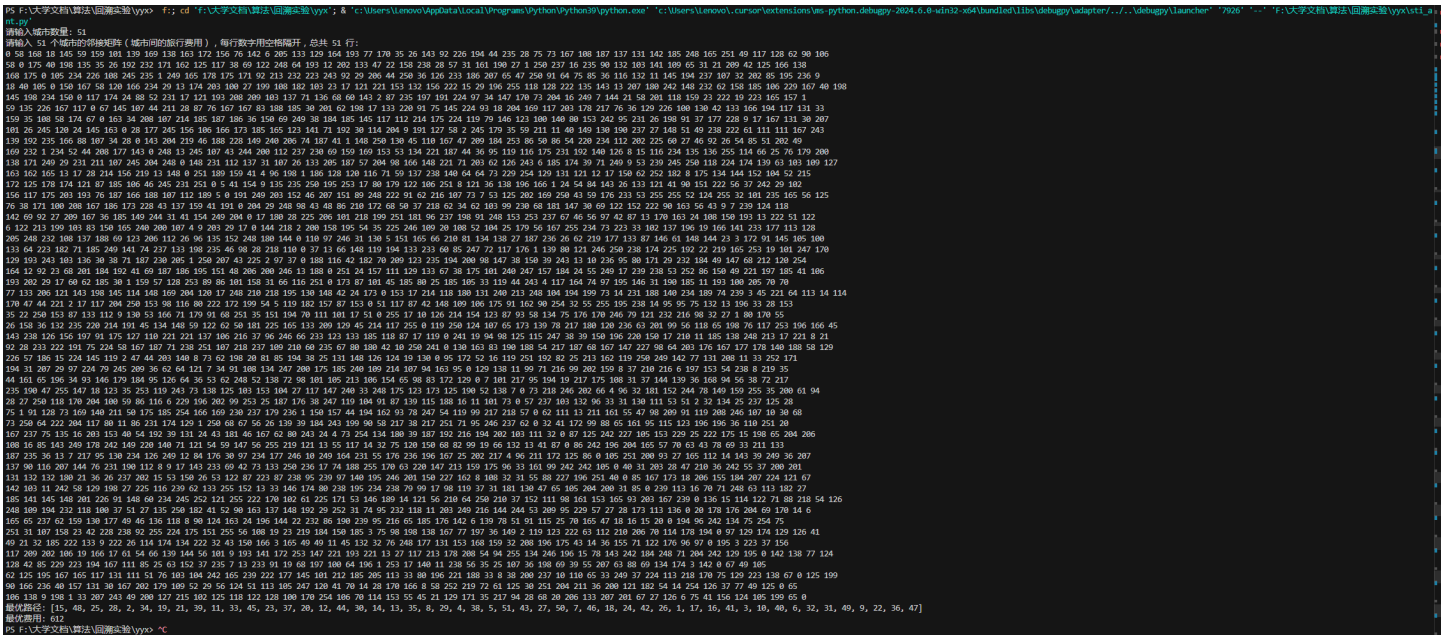
可以发现，模拟蚁群算法给出了最优解，而贪心算法给出了次优解，但相差并不大。

5.2 随机生成 n=51

分别对贪心算法和模拟蚁群算法进行测试，得到的结果如图所示。



贪心算法在n=51时的结果



模拟蚁群在n=51时的结果

最小成本如下：

1. 贪心算法：975
2. 模拟蚁群算法：612

从结果可以看出，模拟蚁群算法凭借群体智能表现最为出色。令人意外的是，贪心算法的表现优于其他两种算法。这可能是由于在随机生成的数据集中，城市之间的距离分布较为均匀且缺乏明显的模式。尽管贪心算法采用的是局部最优策略，但在这种随机环境下，其结果可能已经接近平均解的质量。

相比之下，其他优化算法的核心在于利用路径的局部特征进行优化，这使得它们在处理具有特定模式的城市距离矩阵或物流配送网络时更为有效。然而，在随机分布的矩阵中，这些优化策略可能无法带来显著的改进，甚至由于较高的计算开销，导致整体性能下降。

关键点总结：

- **模拟蚁群算法**通过群体智能在随机数据集上表现优异，获得了最低的成本。
- **贪心算法**在随机分布的数据中表现出色，可能因为其简单有效的局部最优策略在均匀分布的环境中足以接近全局最优。
- **其他优化算法**在处理具有特定结构或模式的数据时更具优势，但在随机分布的场景下，可能因计算复杂度高而效果不佳。

---

6 算法改进

6.1 回溯法改进

1. **引入下界估计与动态剪枝**：
   通过引入当前路径的下界估计，提前判断路径是否可能优于当前最优解。如果当前路径的下界已经超过最优解，则提前剪枝，从而避免继续计算不必要的路径。
2. **优先选择低成本边以加速解的发现**：
   对于每个城市，优先选择到达其他城市的低成本边。这样可以确保回溯算法更早地找到较好的解，并加速算法的收敛过程。
3. **预计算最小出边与位掩码优化访问状态**：
   通过预计算每个城市的最小出边，避免在回溯过程中重复计算最小边，从而提升效率。并采用位掩码表示城市的访问状态，进一步减少不必要的计算，并有效避免状态的重复计算。

> 这些优化是常数级别优化，没从本质上改进回溯法的时间复杂度。
>
> 经过测试，仍然可以求解出最优解，实验结果如图所示。

```
实验\yyx\backtrack_im.py'
请输入城市数量：10
请输入 10 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 10 行：
0 10 17 20 5 30 35 24 45 50
10 0 25 30 35 40 45 50 55 60
17 25 0 35 40 45 50 55 60 65
20 30 35 0 10 15 20 25 30 35
5 35 40 10 0 5 10 15 20 25
30 40 45 15 5 0 5 10 15 20
35 45 50 20 10 5 0 5 10 15
24 50 55 25 15 10 5 0 5 10
45 55 60 30 20 15 10 5 0 5
50 60 65 35 25 20 15 10 5 0

最优路径：[1, 2, 3, 4, 6, 7, 8, 9, 10, 5, 1]
最优费用：135
```

回溯法优化结果

6.2 动态规划法改进

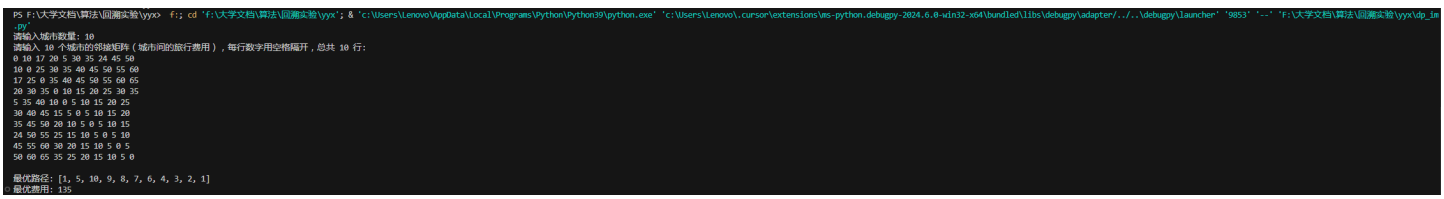
1. **压缩父节点信息：**

将 `parent[mask][u]` 数组中的每个 `parent` 项目压缩为一个单独的整数，利用位操作来表示父节点。

2. **封装状态：**

使用结构体或类封装 DP 状态和相关信息，使得每个城市的状态（例如当前的最短路径和父节点）都存储在一个对象中。

在实现时已经对动态规划法进行了较多的优化，改进空间有限。经过测试，仍然可以求解出最优解，实验结果如图所示。



DP法优化结果

7.总结

旅行商问题（TSP）作为组合优化领域的经典问题，长期以来在理论算法研究中占据核心地位。其解决不仅是对优化算法的理论考验，更是对复杂系统建模与求解策略的深刻挑战。TSP的研究不仅揭示了优化算法的优劣，也深化了我们对计算资源限制、算法效率与问题规模之间复杂关系的理解。

在精确算法的应用中，回溯法与动态规划法凭借其严密的理论基础和精致的求解方法展现了优越的性能。回溯法通过深度优先的策略遍历所有可能解，并结合剪枝策略有效减少冗余计算，其本质是对“枚举”方法的优化与精炼。动态规划法则利用明确的状态转移关系，将复杂问题逐层分解成子问题，并通过子问题的最优解逐步合成全局最优解，体现了数学建模与算法设计的深度融合。

但是这些方法在面对规模较大的TSP问题时，精确算法的计算复杂度呈指数级增长，这使得它们在实际应用中难以处理大规模数据，全局最优解的计算代价通常超出限定范围。

与此相比，近似算法因其较高的适应性与计算效率，在解决大规模TSP问题时展现了独特优势。贪心算法凭借其直观且高效的策略，通过局部最优路径的选择快速逼近问题的解，尽管其在全局优化中存在短视性，导致在某些复杂或非均匀分布的数据中，表现不佳。

模拟蚁群算法借鉴自然界蚂蚁觅食的行为，利用群体智能和信息素的传递机制实现全局优化。该算法通过反复迭代和局部搜索策略的结合，能够在复杂的优化空间中保持良好的探索性和收敛性，特别是在不确定性较强的实际问题中，表现出色。

通过实验分析，精确算法在小规模TSP问题中能够提供较为理想的解，而近似算法在面对较大规模问题时，能够在保证计算效率的同时提供可接受的解质量。在实际应用场景中，贪心算法通过快速的局部决策可以在某些情况下提供较为合理的解，尤其是在问题规模适中或数据分布较为均匀时。相比之下，模拟蚁群算法凭借其高度的适应性和优异的全局搜索能力，在不同的测试条件下均能稳定地表现出较强的鲁棒性，证明其在复杂优化问题中的有效性和灵活性。

这些实验结果不仅揭示了不同算法的性能差异，还进一步加深了我们对优化方法本质的理解。每一种算法的设计与实现，均反映了其背后数学模型与计算策略的局限性与潜力。TSP问题的研究，超越了单纯寻找最优解的目标，更在于通过探索多样化的求解方法与优化思路，推动对优化科学及复杂系统问题的深入理解。

8.关键源码

回溯：

```python
# 定义一个函数，解决旅行商问题的回溯算法
def tsp_backtrack(G, n):
    # 初始化全局最优解和最优费用
    best_cost = math.inf
    best_solution = None

    # Backtrack函数
    def backtrack(i, current_path, current_cost):
        nonlocal best_cost, best_solution
        if i == n:
            # 到达叶子结点，检查是否构成一个有效回路
            if G[current_path[-1]][current_path[0]] != math.inf:  # 确保路径有效
                total_cost = current_cost + G[current_path[-1]][current_path[0]]
                if total_cost < best_cost:
                    best_cost = total_cost
                    best_solution = current_path + [current_path[0]]
            return
```

```
18
19          # 遍历所有未访问的城市
20          for j in range(n):
21              if j not in current_path and G[current_path[i-1]][j] != math.inf:
22                  new_cost = current_cost + G[current_path[i-1]][j]
23                  if new_cost < best_cost:  # 剪枝条件
24                      backtrack(i + 1, current_path + [j], new_cost)
25
26      # 初始化回溯搜索
27      backtrack(1, [0], 0)  # 从城市0开始
28
29      return best_solution, best_cost
30
31  # 获取用户输入的邻接矩阵
32  def get_adjacency_matrix(n):
33      print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共
    {n} 行:")
34      G = []
35      for i in range(n):
36          row = list(map(int, input().split()))  # 读取一行输入并转换为整数列表
37          G.append(row)
38      return G
39
```

回溯改进：

```
1   # 定义一个函数，解决旅行商问题的回溯算法
2   def tsp_backtrack(G, n):
3       # 初始化全局最优解和最优费用
4       best_cost = math.inf
5       best_solution = None
6
7       # 预计算每个城市的最小出边
8       min_edges = [min(row) for row in G]
9
10      # 定义一个估计函数，用于计算当前路径的下界
11      def lower_bound(current_path, current_cost):
12          # 估计当前路径的下界：从当前城市出发，选择每个未访问城市的最小边
13          bound = current_cost
14          for city in range(n):
15              if city not in current_path:
16                  bound += min_edges[city]  # 选择未访问城市的最小出边
17          return bound
18
19      # Backtrack函数
```

```
    def backtrack(i, current_path, current_cost):
        nonlocal best_cost, best_solution
        if i == n:
            # 到达叶子结点，检查是否构成一个有效回路
            if G[current_path[-1]][current_path[0]] != math.inf:
                total_cost = current_cost + G[current_path[-1]]
[current_path[0]]
                if total_cost < best_cost:
                    best_cost = total_cost
                    best_solution = current_path + [current_path[0]]
            return

        # 剪枝：计算当前路径的下界，若下界大于最优解，则剪枝
        lb = lower_bound(current_path, current_cost)
        if lb >= best_cost:
            return

        # 遍历所有未访问的城市，优先选择成本较低的边
        for j in range(n):
            if j not in current_path and G[current_path[i-1]][j] != math.inf:
                new_cost = current_cost + G[current_path[i-1]][j]
                if new_cost < best_cost:  # 剪枝条件
                    backtrack(i + 1, current_path + [j], new_cost)

    # 初始化回溯搜索
    backtrack(1, [0], 0)  # 从城市0开始

    return best_solution, best_cost

# 获取用户输入的邻接矩阵
def get_adjacency_matrix(n):
    print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共
{n} 行:")
    G = []
    for i in range(n):
        row = list(map(int, input().split()))  # 读取一行输入并转换为整数列表
        G.append(row)
    return G

```

dp:

```

def ReconstructPath(parent, mask, u, n):
    path = [u]
```

```
    while parent[mask][u] != -1:
        next_city = parent[mask][u]
        path.append(next_city)
        mask ^= (1 << u)  # 去掉当前城市
        u = next_city
    path.reverse()  # 从起点到终点的路径顺序
    return path

def TSP_DP(a, n):
    # 初始化变量
    N = 1 << n  # 总的子集数量
    dp = [[Infinite] * n for _ in range(N)]  # dp[mask][u]表示经过集合S到城市u的最短路径
    parent = [[-1] * n for _ in range(N)]  # parent[mask][u]存储路径的父城市

    # 设置起点，假设起点为城市 0
    dp[1][0] = 0

    # 预计算每个城市的最小出边，用于下界估计
    min_edges = [min([cost for cost in row if cost != 0]) if any(cost != 0 for cost in row) else Infinite for row in a]

    # 遍历所有子集
    for mask in range(1, N):
        for u in range(n):
            if (mask & (1 << u)) != 0 and dp[mask][u] != Infinite:
                for v in range(n):
                    if (mask & (1 << v)) == 0 and a[u][v] != Infinite:
                        next_mask = mask | (1 << v)
                        new_cost = dp[mask][u] + a[u][v]
                        if new_cost < dp[next_mask][v]:
                            dp[next_mask][v] = new_cost
                            parent[next_mask][v] = u

    # 寻找闭合回到起点的最小成本路径
    final_mask = (1 << n) - 1  # 最终子集包括所有城市
    min_cost = Infinite
    last_city = -1

    for u in range(1, n):
        if a[u][0] != Infinite and dp[final_mask][u] != Infinite:
            total_cost = dp[final_mask][u] + a[u][0]  # 回到起点城市的成本
            if total_cost < min_cost:
                min_cost = total_cost
                last_city = u

    if min_cost == Infinite:
```

```
        return Infinite, []

    # 重构最佳路径
    bestx = ReconstructPath(parent, final_mask, last_city, n) + [0]  # 加回起点
    bestx = [city + 1 for city in bestx]  # 转为1-based indexing

    return min_cost, bestx

# 获取用户输入的邻接矩阵
def get_adjacency_matrix(n):
    print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 {n} 行:")
    G = []
    for i in range(n):
        while True:
            try:
                row = list(map(int, input().split()))
                if len(row) != n:
                    raise ValueError(f"第 {i+1} 行输入的数字个数不正确，应为 {n} 个。")
                G.append(row)
                break
            except ValueError as ve:
                print(f"输入错误: {ve}")
                print("请重新输入该行数字，用空格隔开。")
    return G

```

dp改进：

```

# 定义状态结构体，用于封装每个状态的dp值和父节点信息
class TSPState:
    def __init__(self, cost=math.inf, parent=-1):
        self.cost = cost  # 当前的旅行成本
        self.parent = parent  # 当前城市的父节点

# 用于表示无路径的无穷大
Infinite = float('inf')

def ReconstructPath(dp, mask, u, n):
    path = [u]
    while dp[mask][u].parent != -1:  # 修正这里，使用 TSPState 的 parent
        next_city = dp[mask][u].parent
        path.append(next_city)
```

```
        mask ^= (1 << u)  # 去掉当前城市
        u = next_city
    path.reverse()  # 从起点到终点的路径顺序
    return path

def TSP_DP(a, n):
    # 初始化变量
    N = 1 << n  # 总的子集数量
    dp = [[TSPState() for _ in range(n)] for _ in range(N)]  # dp[mask][u]表示经过集合S到城市u的最短路径

    # 设置起点，假设起点为城市 0
    dp[1][0].cost = 0

    # 预计算每个城市的最小出边，用于下界估计
    min_edges = [min([cost for cost in row if cost != 0]) if any(cost != 0 for cost in row) else Infinite for row in a]

    # 遍历所有子集
    for mask in range(1, N):
        for u in range(n):
            if (mask & (1 << u)) != 0 and dp[mask][u].cost != Infinite:
                for v in range(n):
                    if (mask & (1 << v)) == 0 and a[u][v] != Infinite:
                        next_mask = mask | (1 << v)
                        new_cost = dp[mask][u].cost + a[u][v]
                        if new_cost < dp[next_mask][v].cost:
                            dp[next_mask][v].cost = new_cost
                            dp[next_mask][v].parent = u

    # 寻找闭合回到起点的最小成本路径
    final_mask = (1 << n) - 1  # 最终子集包括所有城市
    min_cost = Infinite
    last_city = -1

    for u in range(1, n):
        if a[u][0] != Infinite and dp[final_mask][u].cost != Infinite:
            total_cost = dp[final_mask][u].cost + a[u][0]  # 回到起点城市的成本
            if total_cost < min_cost:
                min_cost = total_cost
                last_city = u

    if min_cost == Infinite:
        return Infinite, []

    # 重构最佳路径
    bestx = ReconstructPath(dp, final_mask, last_city, n) + [0]  # 加回起点
```

```
    bestx = [city + 1 for city in bestx]  # 转为1-based indexing

    return min_cost, bestx

# 获取用户输入的邻接矩阵
def get_adjacency_matrix(n):
    print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 {n} 行:")
    G = []
    for i in range(n):
        while True:
            try:
                row = list(map(int, input().split()))
                if len(row) != n:
                    raise ValueError(f"第 {i+1} 行输入的数字个数不正确，应为 {n} 个。")
                G.append(row)
                break
            except ValueError as ve:
                print(f"输入错误: {ve}")
                print("请重新输入该行数字，用空格隔开。")
    return G

```

贪心：

```
# 计算路径的总成本
def calculate_cost(path, a):
    cost = 0
    for i in range(len(path) - 1):
        cost += a[path[i]][path[i+1]]
    cost += a[path[-1]][path[0]]  # 回到起点
    return cost

# 贪心算法主体
def greedy_algorithm(a, n):
    visited = [False] * n  # 标记城市是否访问过
    path = [0]  # 从城市0开始
    visited[0] = True  # 标记城市0为已访问
    current_city = 0

    # 每次选择距离最近的城市
    for _ in range(n - 1):
        nearest_city = -1
        min_cost = Infinite
```

```
        for next_city in range(n):
            if not visited[next_city] and a[current_city][next_city] < min_cost:
                nearest_city = next_city
                min_cost = a[current_city][next_city]
        path.append(nearest_city)
        visited[nearest_city] = True
        current_city = nearest_city

    # 计算路径的总费用
    cost = calculate_cost(path, a)
    return path, cost

# 获取用户输入的邻接矩阵
def get_adjacency_matrix(n):
    print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 {n} 行:")
    a = []
    for i in range(n):
        row = list(map(int, input().split()))  # 读取一行输入并转换为整数列表
        a.append(row)
    return a
```

蚁群：

```

def calculate_cost(path, a):
    cost = 0
    for i in range(len(path) - 1):
        cost += a[path[i]][path[i+1]]
    cost += a[path[-1]][path[0]]  # 回到起点
    return cost

# 更新信息素
def update_pheromone(pheromone, paths, costs):
    for i in range(len(paths)):
        for j in range(len(paths[i]) - 1):
            pheromone[paths[i][j]][paths[i][j+1]] += Q / costs[i]
            pheromone[paths[i][j+1]][paths[i][j]] += Q / costs[i]  # 对称
    return pheromone

# 选择路径的启发式方法
def select_path(probabilities):
    r = random.random()
    total = 0
```

```
    for i, prob in enumerate(probabilities):
        total += prob
        if total >= r:
            return i
    return len(probabilities) - 1

# 蚁群算法主体
def ant_colony_algorithm(a, n):
    pheromone = [[1.0 for _ in range(n)] for _ in range(n)]  # 初始化信息素
    best_cost = Infinite
    best_solution = None

    # 蚁群迭代
    for iteration in range(n_iter):
        paths = []
        costs = []

        # 每只蚂蚁构造一条路径
        for ant in range(n_ants):
            path = [random.randint(0, n-1)]  # 从一个随机城市开始
            visited = set(path)
            for _ in range(n-1):
                current = path[-1]
                probabilities = []
                for next_city in range(n):
                    if next_city not in visited and a[current][next_city] !=
Infinite:
                        prob = (pheromone[current][next_city] ** alpha) * \
                               ((1 / a[current][next_city]) ** beta)
                        probabilities.append(prob)
                    else:
                        probabilities.append(0)
                total_prob = sum(probabilities)
                probabilities = [p / total_prob if total_prob > 0 else 0 for p
in probabilities]
                next_city = select_path(probabilities)
                path.append(next_city)
                visited.add(next_city)

            # 计算路径的总费用
            cost = calculate_cost(path, a)
            paths.append(path)
            costs.append(cost)

            # 更新最优解
            if cost < best_cost:
                best_cost = cost
```

```
                best_solution = path

        # 更新信息素
        pheromone = update_pheromone(pheromone, paths, costs)

        # 信息素挥发
        for i in range(n):
            for j in range(n):
                pheromone[i][j] = (1 - rho) * pheromone[i][j]

    return best_solution, best_cost

# 获取用户输入的邻接矩阵
def get_adjacency_matrix(n):
    print(f"请输入 {n} 个城市的邻接矩阵（城市间的旅行费用），每行数字用空格隔开，总共 {n} 行:")
    a = []
    for i in range(n):
        row = list(map(int, input().split()))  # 读取一行输入并转换为整数列表
        a.append(row)
    return a

```