# 旅行商问题的几种方法的探讨

**摘要：**旅行商问题（Traveling Salesman Problem, TSP）是计算机科学和运筹学领域中一个经典且具有挑战性的组合优化问题。

**关键词：**旅行商问题

**Exploration of Several Methods for Traveling Salesman Problems**

**Abstract**: The Traveling Salesman Problem (TSP) is a classic and challenging combinatorial optimization problem in the fields of computer science and operations research.

**Key Words**: The Traveling Salesman Problem; TSP

—————————————

**1 引 言**

旅行商问题（Traveling Salesman Problem, TSP）是计算机科学和运筹学领域中一个经典且具有挑战性的组合优化问题。自1930年代首次提出以来，TSP一直是研究的热点，因为它不仅在理论上具有重要的学术价值，而且在实际应用中有着广泛的应用场景，如物流配送、电路板设计、DNA测序等。TSP的核心目标是在给定一组城市及其之间的距离后，找到一条访问所有城市并返回起点的最短路径。

尽管TSP的定义简单明了，但其求解难度却非常高。随着城市数量的增加，问题的复杂性呈指数级增长，这使得TSP成为了一个典型的NP-hard问题。因此，寻找高效的求解方法一直是研究的重点。传统的精确算法，如动态规划和分支定界法，虽然在理论上可以找到最优解，但在实际应用中往往受限于计算资源的限制，难以处理大规模问题。

为了应对这一挑战，研究者们提出了多种近似算法和启发式算法，如贪心算法、Christofides算法、模拟退火、遗传算法和蚁群算法等。这些方法虽然在时间复杂度上具有优势，但通常只能保证找到近似最优解，而非精确最优解。因此，如何在保证解的质量的同时，提高算法的效率，成为了当前研究的重要方向。

本文旨在探讨旅行商问题的多种求解方法，并分析其优缺点。我们将首先介绍TSP的基本定义和数学模型，然后详细讨论几种常见的求解方法，包括精确算法、近似算法和启发式算法。最后，我们将对一些算法进行优化，并通过实验验证这些方法的性能，探讨未来的研究方向。

**2 数学模型的建立**

旅行商问题（Traveling Salesman Problem，TSP）是一个经典的组合优化问题，属于NP-hard问题。问题的描述如下：

假设有一个旅行商，他需要访问一组城市，并且每个城市只能访问一次，最后回到出发城市。问题的目标是找到一条路径，使得旅行商访问所有城市的总旅行距离最短。

问题的数学描述:

给定一个完全图G=(V,E),其中V是城市的集合，E是边的集合，每条边(i,j)有一个权重dij表示城市i和城市j之间的距离。旅行商问题就是要找到一个哈密顿回路（Hamiltonian Circuit），即一个包含所有顶点的回路，使得回路的总权重最小。

**3 精确算法的介绍**

旅行商问题（TSP）是一个经典的组合优化问题，目标是找到一条经过所有给定城市且每个城市只经过一次的最短路径。由于其NP-hard性质，精确算法在处理大规模问题时可能会非常耗时。以下是几种常见的精确算法：

**3.1 穷举法（Exhaustive Search）**

穷举法是最直接的解决TSP的方法，即枚举所有可能的路径并计算每条路径的总距离，选择最短的路径。

步骤：

1. 生成所有可能的城市排列。

2. 计算每种排列的总距离。

3. 选择总距离最小的排列。

复杂度：O(n!)，其中n是城市的数量。

适用范围：仅适用于非常小的城市数量（通常n < 10）。

**3.2 动态规划（Dynamic Programming）**

动态规划通过将问题分解为子问题来减少计算量。常见的动态规划方法是使用状态压缩动态规划（State Compression Dynamic Programming, SCDP）。

步骤：

1. 定义状态：dp[S][i]表示经过集合S中的城市且最后一个城市是i的最短路径长度。

2. 初始化：dp[1 << i][i] = dist[0][i]，其中dist[0][i]是从起点到城市i的距离。

3. 状态转移：dp[S][i] = min(dp[S - {i}][j] + dist[j][i])，其中j是集合S中除i以外的城市。

4. 最终结果：min(dp[(1 << n) - 1][i] + dist[i][0])，其中i是最后一个城市。

复杂度：O(n^2 \* 2^n)，其中n是城市的数量。

适用范围：适用于中等规模的城市数量（通常n < 20）。

**3.3 分支定界法（Branch and Bound）**

分支定界法是一种通过剪枝来减少搜索空间的算法。它通过不断分支和计算下界来逐步缩小解空间。

步骤：

1. 初始化一个优先队列，存储当前路径和路径长度。

2. 从队列中取出路径长度最小的路径。

3. 如果路径已经包含所有城市，更新最优解。

4. 否则，生成所有可能的下一步路径，计算其下界，并将其加入队列。

5. 如果某一路径的下界大于当前最优解，则剪枝。

复杂度：取决于剪枝的效果，通常比穷举法和动态规划更快。

适用范围：适用于中等规模的城市数量（通常n < 30）。

**3.4 线性规划（Linear Programming）**

线性规划可以通过整数规划来求解TSP。常见的线性规划方法是使用线性松弛和分支定界。

步骤：

1. 定义变量：x[i][j]表示是否从城市i到城市j。

2. 定义目标函数：最小化路径长度。

3. 添加约束条件：每个城市只能进入一次，每个城市只能离开一次，路径形成一个环。

4. 使用线性规划求解器（如CPLEX、Gurobi）求解。

复杂度：取决于线性规划求解器的效率。

适用范围：适用于中等规模的城市数量（通常n < 50）。

**3.5 总结**

旅行商问题的精确算法包括穷举法、动态规划、分支定界法和线性规划。每种方法都有其适用范围和复杂度。选择合适的算法取决于问题的规模和计算资源的限制。

**4 近似算法的介绍**

由于旅行商问题（TSP）是NP-hard问题，精确算法在处理大规模问题时可能会非常耗时。因此，近似算法成为解决TSP的重要手段。近似算法通过牺牲一定的精确性来换取更快的计算速度。以下是几种常见的近似算法：

**4.1 最近邻算法（Nearest Neighbor Algorithm）**

最近邻算法是一种贪心算法，每次选择距离当前城市最近的未访问城市作为下一个访问城市。

步骤：

1. 选择一个起始城市。

2. 重复以下步骤，直到所有城市都被访问：

3. 选择距离当前城市最近的未访问城市。

4. 将该城市标记为已访问，并将其加入路径。

5. 返回起始城市，形成一个闭合路径。

复杂度：O(n^2)，其中n是城市的数量。

近似比：通常情况下，最近邻算法的解与最优解的比值在1.5到2之间。

**4.2 Christofides算法**

Christofides算法是一种针对欧几里得TSP（即城市坐标在欧几里得空间中）的近似算法，其近似比为1.5。

步骤：

1. 构建最小生成树（Minimum Spanning Tree, MST）。

2. 找出所有度数为奇数的节点，形成一个子图。

3. 在子图中找到一个最小权匹配（Minimum Weight Perfect Matching）。

4. 将匹配边加入MST，形成一个欧拉图。

5. 通过欧拉回路找到一个哈密顿回路（Hamiltonian Circuit）。

复杂度：O(n^3)，其中n是城市的数量。

近似比：1.5。

**4.3 插入算法（Insertion Algorithms）**

插入算法通过逐步插入城市来构建路径。常见的插入算法包括最近插入（Nearest Insertion）、最远插入（Farthest Insertion）和随机插入（Random Insertion）。

一、最近插入（Nearest Insertion）：

1. 选择一个起始城市。

2. 重复以下步骤，直到所有城市都被插入：

3. 选择距离当前路径中任意城市最近的一个未插入城市。

4. 找到路径中插入该城市后增加路径长度最小的位置，将其插入。

二、最远插入（Farthest Insertion）：

1. 选择一个起始城市。

2. 重复以下步骤，直到所有城市都被插入：

3. 选择距离当前路径中任意城市最远的一个未插入城市。

4. 找到路径中插入该城市后增加路径长度最小的位置，将其插入。

复杂度：O(n^2)，其中n是城市的数量。

近似比：最近插入和最远插入的近似比通常在2左右。

**4.4 Lin-Kernighan算法**

Lin-Kernighan算法是一种局部搜索算法，通过不断交换路径中的边来改进路径长度。

步骤：

1. 选择一个初始路径。

2. 重复以下步骤，直到无法进一步改进：

3. 选择一对边进行交换，形成新的路径。

4. 如果新路径更短，则接受该路径。

复杂度：取决于搜索的深度和广度，通常为O(n^2)到O(n^3)。

近似比：Lin-Kernighan算法通常能找到非常接近最优解的路径，但理论上没有固定的近似比。

**4.5 遗传算法（Genetic Algorithms）**

遗传算法是一种基于自然选择和遗传机制的优化算法，适用于解决TSP等组合优化问题。

步骤：

1. 初始化一个种群，每个个体表示一条路径。

2. 重复以下步骤，直到满足终止条件：

3. 选择：根据适应度函数（路径长度）选择个体。

4. 交叉：通过交叉操作生成新的个体。

5. 变异：对新个体进行变异操作。

6. 更新种群：用新生成的个体替换部分旧个体。

7. 选择适应度最高的个体作为最终解。

复杂度：取决于种群大小和迭代次数，通常为O(n^2 \* P \* I)，其中P是种群大小，I是迭代次数。

近似比：遗传算法通常能找到较好的解，但理论上没有固定的近似比。

**4.6 总结**

旅行商问题的近似算法包括最近邻算法、Christofides算法、插入算法、Lin-Kernighan算法和遗传算法。每种算法都有其优缺点和适用范围。选择合适的算法取决于问题的规模、计算资源的限制以及对解的精确性要求。

**5 启发式算法的介绍**

启发式算法（Heuristic Algorithms）是解决旅行商问题（TSP）的一种重要方法，它们通过利用问题的特定结构和启发式规则来快速找到近似解。启发式算法通常比精确算法更快，但不一定能保证找到最优解。以下是几种常见的启发式算法：

**5.1 模拟退火算法（Simulated Annealing）**

模拟退火算法是一种基于物理退火过程的优化算法，通过模拟固体物质的冷却过程来寻找全局最优解。

步骤：

1. 初始化：选择一个初始解（路径）和初始温度T。

重复以下步骤，直到温度T降到某个阈值：

2. 生成一个新解（路径），通常是通过对当前解进行小幅度随机扰动。

3. 计算新解与当前解的目标函数差值ΔE（路径长度差）。

4. 如果ΔE < 0（新解更优），接受新解。

5. 如果ΔE > 0（新解更差），以概率exp(-ΔE/T)接受新解。

6. 降低温度T。

7. 返回最终解。

复杂度：取决于温度下降速度和迭代次数，通常为O(n^2 \* I)，其中n是城市的数量，I是迭代次数。

优点：能够跳出局部最优解，找到全局最优解的概率较高。

缺点：参数选择（如初始温度、降温速度）对结果影响较大。

**5.2 禁忌搜索（Tabu Search）**

禁忌搜索是一种基于邻域搜索的元启发式算法，通过引入禁忌表来避免重复搜索和陷入局部最优解。

步骤：

1. 初始化：选择一个初始解（路径）和禁忌表。

2. 重复以下步骤，直到满足终止条件：

3. 生成当前解的邻域解（通常是通过交换路径中的城市）。

4. 选择邻域解中最好的解，除非它被禁忌表禁止。

5. 更新禁忌表，将当前解加入禁忌表。

6. 返回最终解。

复杂度：取决于邻域大小和迭代次数，通常为O(n^2 \* I)，其中n是城市的数量，I是迭代次数。

优点：能够有效避免重复搜索和陷入局部最优解。

缺点：需要合理设置禁忌表的大小和禁忌期限。

**5.3 蚁群算法（Ant Colony Optimization, ACO）**

蚁群算法是一种基于蚂蚁觅食行为的群体智能算法，通过模拟蚂蚁在路径上留下信息素来寻找最优路径。

步骤：

1. 初始化：设置信息素矩阵和蚂蚁数量。

2. 重复以下步骤，直到满足终止条件：

3. 每只蚂蚁从起点出发，根据信息素和启发式信息选择下一个城市。

4. 蚂蚁完成路径后，更新信息素矩阵（增加路径上的信息素）。

5. 蒸发部分信息素，避免信息素过多积累。

6. 返回信息素最多的路径。

复杂度：取决于蚂蚁数量和迭代次数，通常为O(n^2 \* A \* I)，其中n是城市的数量，A是蚂蚁数量，I是迭代次数。

优点：能够有效利用群体智能，找到较好的解。

缺点：参数选择（如信息素蒸发率、蚂蚁数量）对结果影响较大。

**5.4 粒子群优化（Particle Swarm Optimization, PSO）**

粒子群优化是一种基于群体智能的优化算法，通过模拟鸟群或鱼群的行为来寻找最优解。

步骤：

1. 初始化：设置粒子群（每个粒子表示一条路径）和速度。

2. 重复以下步骤，直到满足终止条件：

3. 每个粒子根据自身历史最优解和群体历史最优解更新速度和位置。

4. 更新每个粒子的历史最优解和群体历史最优解。

5. 返回群体历史最优解。

复杂度：取决于粒子数量和迭代次数，通常为O(n^2 \* P \* I)，其中n是城市的数量，P是粒子数量，I是迭代次数。

优点：能够有效利用群体智能，找到较好的解。

缺点：参数选择（如惯性权重、学习因子）对结果影响较大。

**5.5 总结**

旅行商问题的启发式算法包括模拟退火算法、禁忌搜索、蚁群算法和粒子群优化。每种算法都有其独特的优点和适用范围。选择合适的启发式算法取决于问题的规模、计算资源的限制以及对解的精确性要求。启发式算法通常能够在较短时间内找到较好的近似解，但不一定能保证找到最优解。

**6 分支定界法的详细介绍**

分支定界法（Branch and Bound, B&B）是一种通过逐步缩小搜索空间来找到最优解的方法。它通过分支（将问题分解为子问题）和定界（通过上界和下界来剪枝）来减少计算量。

首先详细了解一下精确算法。

**2.1 基本思想**

分支定界法的核心思想是将TSP问题分解为多个子问题，并通过计算每个子问题的上界和下界来决定是否继续搜索该子问题。如果某个子问题的上界大于当前找到的最优解，则可以剪枝，不再继续搜索该子问题。

**2.2 分支**

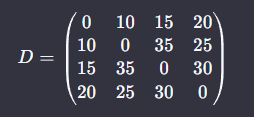
分支的过程是将当前问题分解为多个子问题。例如，假设当前问题是从城市0出发，访问城市1、2、3，我们可以选择访问城市1，然后问题分解为从城市1出发，访问城市2、3。

**2.3 定界**

定界的过程是通过计算当前路径的上界和下界来决定是否继续搜索。上界可以通过贪心算法或近似算法得到，下界可以通过计算最小生成树（MST）或线性规划得到。

**2.4 示例**

假设有4个城市V = {0, 1, 2, 3}，距离矩阵D如下：



例子说明：

1. 初始状态：

从城市0出发，访问城市1、2、3。

2. 分支：

选择访问城市1，问题分解为从城市1出发，访问城市2、3。

选择访问城市2，问题分解为从城市2出发，访问城市1、3。

选择访问城市3，问题分解为从城市3出发，访问城市1、2。

3. 定界：

计算从城市0出发，访问城市1、2、3的上界和下界。

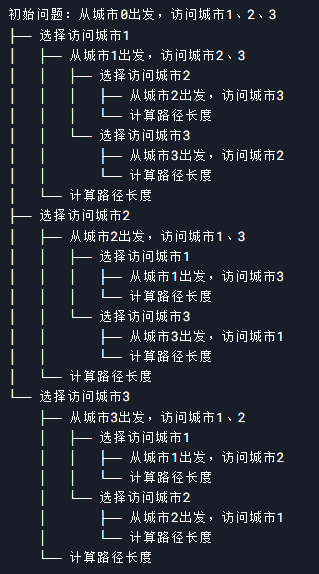
如果上界大于当前找到的最优解，则剪枝。

4. 继续分支和定界

对每个子问题继续进行分支和定界，直到找到最优解。

**2.5 图示**

以下是一个简单的分支定界法的图示：



**6 分支定界法的优化**

启发式搜索的原理：

启发式搜索是一种通过利用问题的特定结构和启发式规则来快速找到近似解的算法。在旅行商问题（TSP）中，启发式搜索可以通过估计当前路径的下界来帮助剪枝，从而减少搜索时间。常见的启发式规则包括使用最小生成树（MST）或最近邻算法（Nearest Neighbor）来估计剩余路径的最小长度。

**6.1 原版分支界定法**

分支界定法的原理刚刚已经讲过了，此处不再赘述，下面展示核心函数：

//计算分数

int CalculateScore(const vector<vector<int>>& DistanceMatrix, vector<int>& Path) {

int Score = 0;

for (int i = 0; i < N - 1; i++) {

if ((Path[i] >= 0) && (Path[i + 1] >= 0)) {//如果路径未截止

Score += DistanceMatrix[Path[i]][Path[i + 1]];

}

}

if (Path[N - 1] >= 0) {

Score += DistanceMatrix[Path[N - 1]][0];

}

return Score;

}

//分支定界法求解TSP问题

void BranchAndBoundTSP(const vector<vector<int>>& DistanceMatrix, vector<bool>& Visited, int& MinWeight, vector<int>& BestPath, vector<int>& CurrentPath, int Current, int Depth) {

//如果已经选好

if (Depth == 0) {

int Score = CalculateScore(DistanceMatrix, CurrentPath);

if (Score < MinWeight) {//若更好

MinWeight = Score;

BestPath = CurrentPath;

return;

}

else {

return;

}

}

int Score = CalculateScore(DistanceMatrix, CurrentPath);//计算当前分数

if (Score < MinWeight) {//比最小的小，再继续计算

//遍历城市选择下一个

for (int i = 0; i < N; i++) {

if (Visited[i] == false) {//如果没走过

//模拟

Visited[i] = true;

CurrentPath[N - Depth] = i;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, i, Depth - 1);

//取消模拟

Visited[i] = false;

CurrentPath[N - Depth] = -1;

}

}

}

}

//算法主函数

void SolveTSP(const vector<vector<int>>& DistanceMatrix) {

vector<bool> Visited(N, false);//已经访问过的城市，访问过的打true

int MinWeight = INT\_MAX;//最短距离

vector<int> BestPath(N, -1);//最好路径

vector<int> CurrentPath(N, -1);//当前路径

Visited[0] = true;

BestPath[0] = 0;

CurrentPath[0] = 0;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, 0, N - 1);

//输出结果

cout << "最优路径: ";

for (int i = 0; i < N; i++) {

cout << BestPath[i] << " ";

}

cout << "0" << endl;//回到起点

cout << "最短距离: " << MinWeight << endl;

}

此时尚未加入启发式搜索，仅仅是依靠最原始的计算分数来剪枝。

利用随机算法生成13\*13距离矩阵并进行五次实验，实验结果如下：

·用时3.70797秒。

·用时1.89055秒。

·用时2.06815秒。

·用时0.724762秒。

·用时2.33118秒。

均值为2.1445224秒

**6.2 MST优化示例**

如何将启发式搜索加入分支界定法中？

在分支界定法中加入启发式搜索，主要是通过引入启发式估计来加速剪枝过程。具体步骤如下：

1. 计算当前路径的分数：

在每次递归调用中，计算当前路径的总距离。

2. 计算剩余节点的最小生成树（MST）权重：

使用Prim算法计算未访问节点的最小生成树的权重。这个权重可以作为启发式估计，帮助我们剪枝。

3. 启发式剪枝：

在每次递归调用中，计算当前路径的分数和剩余节点的MST权重之和。如果这个和大于当前的最小权重MinWeight，则进行剪枝，不再继续搜索。

加入启发式搜索后的提升：

1. 加速剪枝：

通过引入最小生成树的启发式估计，我们可以在搜索过程中更快地剪枝。如果当前路径的分数加上剩余节点的MST权重已经大于当前的最小权重MinWeight，则不再继续搜索，从而减少不必要的计算。

2. 减少搜索时间：

启发式搜索能够显著减少搜索时间，特别是在处理较大规模的城市数量时。通过提前剪枝，我们可以避免搜索那些明显不是最优的路径，从而提高算法的效率。

3. 提高解的质量：

虽然启发式搜索不能保证找到最优解，但它通常能够找到较好的近似解。在分支界定法中加入启发式搜索，可以在较短时间内找到接近最优解的路径。

下面展示使用MST算法来进行启发式搜索的核心函数：

//计算分数

int CalculateScore(const vector<vector<int>>& DistanceMatrix, vector<int>& Path) {

int Score = 0;

for (int i = 0; i < N - 1; i++) {

if ((Path[i] >= 0) && (Path[i + 1] >= 0)) {//如果路径未截止

Score += DistanceMatrix[Path[i]][Path[i + 1]];

}

}

if (Path[N - 1] >= 0) {

Score += DistanceMatrix[Path[N - 1]][0];

}

return Score;

}

//计算最小生成树（MST）的权重

int CalculateMST(const vector<vector<int>>& DistanceMatrix, const vector<bool>& Visited) {

vector<int> key(N, INT\_MAX);

vector<bool> mstSet(N, false);

key[0] = 0;

for (int count = 0; count < N - 1; count++) {

int u = -1;

for (int v = 0; v < N; v++) {

if (!mstSet[v] && (u == -1 || key[v] < key[u])) {

u = v;

}

}

mstSet[u] = true;

for (int v = 0; v < N; v++) {

if (!mstSet[v] && DistanceMatrix[u][v] && DistanceMatrix[u][v] < key[v]) {

key[v] = DistanceMatrix[u][v];

}

}

}

int MSTWeight = 0;

for (int i = 0; i < N; i++) {

MSTWeight += key[i];

}

return MSTWeight;

}

//分支定界法求解TSP问题

void BranchAndBoundTSP(const vector<vector<int>>& DistanceMatrix, vector<bool>& Visited, int& MinWeight, vector<int>& BestPath, vector<int>& CurrentPath, int Current, int Depth) {

//如果已经选好

if (Depth == 0) {

int Score = CalculateScore(DistanceMatrix, CurrentPath);

if (Score < MinWeight) {//若更好

MinWeight = Score;

BestPath = CurrentPath;

}

return;

}

int Score = CalculateScore(DistanceMatrix, CurrentPath);//计算当前分数

int MSTWeight = CalculateMST(DistanceMatrix, Visited);//计算剩余节点的MST权重

if (Score + MSTWeight < MinWeight) {//比最小的小，再继续计算

//遍历城市选择下一个

for (int i = 0; i < N; i++) {

if (Visited[i] == false) {//如果没走过

//模拟

Visited[i] = true;

CurrentPath[N - Depth] = i;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, i, Depth - 1);

//取消模拟

Visited[i] = false;

CurrentPath[N - Depth] = -1;

}

}

}

}

//算法主函数

void SolveTSP(const vector<vector<int>>& DistanceMatrix) {

vector<bool> Visited(N, false);//已经访问过的城市，访问过的打true

int MinWeight = INT\_MAX;//最短距离

vector<int> BestPath(N, -1);//最好路径

vector<int> CurrentPath(N, -1);//当前路径

Visited[0] = true;

BestPath[0] = 0;

CurrentPath[0] = 0;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, 0, N - 1);

//输出结果

cout << "最优路径: ";

for (int i = 0; i < N; i++) {

cout << BestPath[i] << " ";

}

cout << "0" << endl;//回到起点

cout << "最短距离: " << MinWeight << endl;

}

利用随机算法生成13\*13距离矩阵并进行五次实验，实验结果如下：

·用时3.02936秒。

·用时6.66653秒。

·用时4.08762秒。

·用时13.016秒。

·用时4.33758秒。

均值为6.227418秒

可以看出，加入MST后时间反而变长了，推测是每次递归调用都需要计算未访问节点的MST权重，这会增加额外的计算开销。如果MST的计算复杂度较高，可能会抵消掉剪枝带来的时间节省。

所以，应该使用更快的Kruskal算法来进行启发式搜索，从而避免这种情况。

**6.3 Kruskal优化示例**

Kruskal算法的原理：

Kruskal算法是一种用于求解最小生成树（Minimum Spanning Tree, MST）的贪心算法。最小生成树是指在一个连通无向图中，找到一棵包含所有顶点的树，并且树的边权值之和最小。Kruskal算法通过不断选择图中权值最小的边，并确保不形成环路，最终构建出最小生成树。

算法步骤：

1. 初始化：

将图中的每个顶点视为一个独立的连通分量（即每个顶点都是一个单独的树）。

将图中的所有边按权值从小到大排序。

2. 选择边：

依次选择权值最小的边，并检查该边的两个顶点是否属于同一个连通分量。

如果两个顶点属于不同的连通分量，则将这条边加入最小生成树，并将这两个连通分量合并为一个连通分量。

如果两个顶点属于同一个连通分量，则跳过这条边，继续选择下一条边。

3. 终止条件：

当最小生成树中的边数达到`n-1`（其中`n`是顶点数）时，算法终止。此时，最小生成树已经构建完成。

并查集（Union-Find）数据结构：

Kruskal算法中，判断两个顶点是否属于同一个连通分量以及合并连通分量的操作可以通过并查集（Union-Find）数据结构来高效实现。并查集支持以下两种操作：

1. 查找（Find）：

查找某个顶点所属的连通分量的根节点。

2. 合并（Union）：

将两个连通分量合并为一个连通分量。

并查集的实现：

并查集通常使用树结构来表示连通分量，每个连通分量的根节点作为代表。通过路径压缩和按秩合并（Union by Rank）等优化技术，可以提高并查集的效率。

路径压缩：在查找操作中，将查找路径上的所有节点直接连接到根节点，从而减少后续查找的时间复杂度。

按秩合并：在合并操作中，将高度较小的树合并到高度较大的树中，从而减少树的高度，提高查找效率。

时间复杂度：

Kruskal算法的时间复杂度主要由以下几个部分组成：

1. 排序边：排序边的时间复杂度为O(m log m)，其中m是边的数量。

2. 并查集操作：并查集的查找和合并操作的时间复杂度接近O(1)（使用路径压缩和按秩合并优化）。

因此，Kruskal算法的总时间复杂度为O(m log m)，其中m是边的数量。

下面展示使用了Kruskal算法的核心代码：

//计算分数

int CalculateScore(const vector<vector<int>>& DistanceMatrix, vector<int>& Path) {

int Score = 0;

for (int i = 0; i < N - 1; i++) {

if ((Path[i] >= 0) && (Path[i + 1] >= 0)) {//如果路径未截止

Score += DistanceMatrix[Path[i]][Path[i + 1]];

}

}

if (Path[N - 1] >= 0) {

Score += DistanceMatrix[Path[N - 1]][0];

}

return Score;

}

//并查集

class UnionFind {

public:

UnionFind(int n) : Parent(n), Rank(n, 0) {

for (int i = 0; i < n; i++) {

Parent[i] = i;

}

}

int find(int x) {

if (Parent[x] != x) {

Parent[x] = find(Parent[x]);

}

return Parent[x];

}

void unite(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (Rank[rootX] < Rank[rootY]) {

Parent[rootX] = rootY;

}

else if (Rank[rootX] > Rank[rootY]) {

Parent[rootY] = rootX;

}

else {

Parent[rootY] = rootX;

Rank[rootX]++;

}

}

}

private:

vector<int> Parent;

vector<int> Rank;

};

//计算最小生成树（MST）的权重

int CalculateMST(const vector<vector<int>>& DistanceMatrix, const vector<bool>& Visited) {

vector<pair<int, pair<int, int>>> edges;

for (int i = 0; i < N; i++) {

if (!Visited[i]) {

for (int j = i + 1; j < N; j++) {

if (!Visited[j]) {

edges.push\_back({ DistanceMatrix[i][j], {i, j} });

}

}

}

}

sort(edges.begin(), edges.end());

UnionFind uf(N);

int MSTWeight = 0;

for (auto& edge : edges) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

if (uf.find(u) != uf.find(v)) {

uf.unite(u, v);

MSTWeight += weight;

}

}

return MSTWeight;

}

//分支定界法求解TSP问题

void BranchAndBoundTSP(const vector<vector<int>>& DistanceMatrix, vector<bool>& Visited, int& MinWeight, vector<int>& BestPath, vector<int>& CurrentPath, int Current, int Depth) {

//如果已经选好

if (Depth == 0) {

int Score = CalculateScore(DistanceMatrix, CurrentPath);

if (Score < MinWeight) {//若更好

MinWeight = Score;

BestPath = CurrentPath;

}

return;

}

int Score = CalculateScore(DistanceMatrix, CurrentPath);//计算当前分数

int MSTWeight = CalculateMST(DistanceMatrix, Visited);//计算剩余节点的MST权重

if (Score + MSTWeight < MinWeight) {//比最小的小，再继续计算

//遍历城市选择下一个

for (int i = 0; i < N; i++) {

if (Visited[i] == false) {//如果没走过

//模拟

Visited[i] = true;

CurrentPath[N - Depth] = i;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, i, Depth - 1);

//取消模拟

Visited[i] = false;

CurrentPath[N - Depth] = -1;

}

}

}

}

//算法主函数

void SolveTSP(const vector<vector<int>>& DistanceMatrix) {

vector<bool> Visited(N, false);//已经访问过的城市，访问过的打true

int MinWeight = INT\_MAX;//最短距离

vector<int> BestPath(N, -1);//最好路径

vector<int> CurrentPath(N, -1);//当前路径

Visited[0] = true;

BestPath[0] = 0;

CurrentPath[0] = 0;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, 0, N - 1);

//输出结果

cout << "最优路径: ";

for (int i = 0; i < N; i++) {

cout << BestPath[i] << " ";

}

cout << "0" << endl;//回到起点

cout << "最短距离: " << MinWeight << endl;

}

利用随机算法生成13\*13距离矩阵并进行五次实验，实验结果如下：

·用时0.146524秒。

·用时1.09253秒。

·用时1.61731秒。

·用时0.97709秒。

·用时1.84082秒。

均值为1.1348548秒，明显短于前两种方法。

**7 总结**

本文详细探讨了旅行商问题（TSP）的多种求解方法，包括精确算法、近似算法和启发式算法。通过对这些方法的介绍和分析，我们可以得出以下结论：

**7.1 精确算法**

精确算法如穷举法、动态规划、分支定界法和线性规划，虽然在理论上可以找到最优解，但在实际应用中往往受限于计算资源的限制，难以处理大规模问题。这些算法适用于小规模或中等规模的问题，但在面对大规模问题时，计算时间会呈指数级增长。

**7.2 近似算法**

近似算法如最近邻算法、Christofides算法、插入算法、Lin-Kernighan算法和遗传算法，通过牺牲一定的精确性来换取更快的计算速度。这些算法在处理大规模问题时具有显著优势，能够在较短时间内找到接近最优解的路径。然而，这些算法通常只能保证找到近似最优解，而非精确最优解。

**7.3 启发式算法**

启发式算法如模拟退火算法、禁忌搜索、蚁群算法和粒子群优化，通过利用问题的特定结构和启发式规则来快速找到近似解。这些算法在处理大规模问题时具有较高的效率，能够在较短时间内找到较好的近似解。然而，启发式算法的结果依赖于参数的选择和算法的初始条件，可能存在一定的随机性和不确定性。

**7.4 分支定界法的优化**

在分支定界法中加入启发式搜索，特别是使用Kruskal算法计算最小生成树（MST）的权重，可以显著提高剪枝效率，减少搜索时间。实验结果表明，使用Kruskal算法进行启发式搜索的分支定界法在处理大规模问题时具有显著优势，能够在较短时间内找到接近最优解的路径。

**7.5 未来研究方向**

未来的研究可以进一步探索以下方向：

1. 优化启发式估计方法：改进启发式估计方法，使其更加准确，从而提高剪枝效率。

2. 结合多种算法：结合多种算法的优势，设计混合算法，以提高解的质量和计算效率。

3. 并行计算和分布式计算：利用并行计算和分布式计算技术，加速大规模TSP问题的求解。

4. 应用场景的扩展：探索TSP在更多实际应用场景中的应用，如物流配送、电路板设计、DNA测序等。

**8 附件：使用Kruskal算法优化的分布界定法的源程序**

/\*古振-2352031\*/

#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <iomanip>

#include <chrono>

using namespace std;

const int N = 13;//城市数量

//计算分数

int CalculateScore(const vector<vector<int>>& DistanceMatrix, vector<int>& Path) {

int Score = 0;

for (int i = 0; i < N - 1; i++) {

if ((Path[i] >= 0) && (Path[i + 1] >= 0)) {//如果路径未截止

Score += DistanceMatrix[Path[i]][Path[i + 1]];

}

}

if (Path[N - 1] >= 0) {

Score += DistanceMatrix[Path[N - 1]][0];

}

return Score;

}

//并查集

class UnionFind {

public:

UnionFind(int n) : Parent(n), Rank(n, 0) {

for (int i = 0; i < n; i++) {

Parent[i] = i;

}

}

int find(int x) {

if (Parent[x] != x) {

Parent[x] = find(Parent[x]);

}

return Parent[x];

}

void unite(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (Rank[rootX] < Rank[rootY]) {

Parent[rootX] = rootY;

}

else if (Rank[rootX] > Rank[rootY]) {

Parent[rootY] = rootX;

}

else {

Parent[rootY] = rootX;

Rank[rootX]++;

}

}

}

private:

vector<int> Parent;

vector<int> Rank;

};

//计算最小生成树（MST）的权重

int CalculateMST(const vector<vector<int>>& DistanceMatrix, const vector<bool>& Visited) {

vector<pair<int, pair<int, int>>> edges;

for (int i = 0; i < N; i++) {

if (!Visited[i]) {

for (int j = i + 1; j < N; j++) {

if (!Visited[j]) {

edges.push\_back({ DistanceMatrix[i][j], {i, j} });

}

}

}

}

sort(edges.begin(), edges.end());

UnionFind uf(N);

int MSTWeight = 0;

for (auto& edge : edges) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

if (uf.find(u) != uf.find(v)) {

uf.unite(u, v);

MSTWeight += weight;

}

}

return MSTWeight;

}

//分支定界法求解TSP问题

void BranchAndBoundTSP(const vector<vector<int>>& DistanceMatrix, vector<bool>& Visited, int& MinWeight, vector<int>& BestPath, vector<int>& CurrentPath, int Current, int Depth) {

//如果已经选好

if (Depth == 0) {

int Score = CalculateScore(DistanceMatrix, CurrentPath);

if (Score < MinWeight) {//若更好

MinWeight = Score;

BestPath = CurrentPath;

}

return;

}

int Score = CalculateScore(DistanceMatrix, CurrentPath);//计算当前分数

int MSTWeight = CalculateMST(DistanceMatrix, Visited);//计算剩余节点的MST权重

if (Score + MSTWeight < MinWeight) {//比最小的小，再继续计算

//遍历城市选择下一个

for (int i = 0; i < N; i++) {

if (Visited[i] == false) {//如果没走过

//模拟

Visited[i] = true;

CurrentPath[N - Depth] = i;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, i, Depth - 1);

//取消模拟

Visited[i] = false;

CurrentPath[N - Depth] = -1;

}

}

}

}

//算法主函数

void SolveTSP(const vector<vector<int>>& DistanceMatrix) {

vector<bool> Visited(N, false);//已经访问过的城市，访问过的打true

int MinWeight = INT\_MAX;//最短距离

vector<int> BestPath(N, -1);//最好路径

vector<int> CurrentPath(N, -1);//当前路径

Visited[0] = true;

BestPath[0] = 0;

CurrentPath[0] = 0;

//调用分支定界法函数

BranchAndBoundTSP(DistanceMatrix, Visited, MinWeight, BestPath, CurrentPath, 0, N - 1);

//输出结果

cout << "最优路径: ";

for (int i = 0; i < N; i++) {

cout << BestPath[i] << " ";

}

cout << "0" << endl;//回到起点

cout << "最短距离: " << MinWeight << endl;

}

//随机生成距离矩阵

void CreateMatrix(vector<vector<int>>& DistanceMatrix) {

vector<int> Matrix(N, 0);

for (int i = 0; i < N; i++) {

DistanceMatrix.push\_back(Matrix);

}

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

if (i == j) {

DistanceMatrix[i][j] = 0;

}

else {

DistanceMatrix[i][j] = rand() % 100;

}

}

}

}

//输出矩阵

void Output(vector<vector<int>>& DistanceMatrix) {

cout << "距离矩阵为" << endl;

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

cout << setw(3) << DistanceMatrix[i][j];

}

cout << endl;

}

}

//主函数

int main() {

srand((unsigned int)(time(0)));//时间种子

//距离矩阵

vector<vector<int>> DistanceMatrix;

CreateMatrix(DistanceMatrix);//随机生成矩阵

Output(DistanceMatrix);//展示

//解决TSP问题

auto Start = chrono::high\_resolution\_clock::now();

SolveTSP(DistanceMatrix);

auto End = chrono::high\_resolution\_clock::now();

chrono::duration<double> Time = End - Start;

cout << "用时" << Time.count() << "秒。" << endl;

return 0;

}