《算法设计与分析》
课程实验报告

专业：计算机科学与技术
班级：2021211304
姓名：杨晨
学号：2021212171

1 概述

1.1 实验内容

1. 书面作业
   参照讲义 PPT 中（p26-28）给出的面向最大化问题（如 0-1 背包问题）的分支限界法算法框架，设计面向最小化问题，e.g. 旅行商问题，的分支限界法算法框架
   将算法框架附在实验报告中
2. 编程作业
   采用回溯法、分支限界法，编程求解不同规模的旅行商问题 TSP，并利用给定数据，验证算法正确性，对比算法的时间复杂性、空间复杂性
3. 统计记录
   从起始城市出发的最短旅行路径
   路径总长度
   扫描过的搜索树结点总数 L
   程序运行时间 T

1.2 开发环境

- Windows10
- Visual Studio Code 1.84.2

2 实验过程

2.1 回溯法求解 TSP

2.1.1 介绍

旅行商问题（Traveling Salesman Problem，TSP）是一个经典的组合优化问题，在计算机科学和运筹学领域有着广泛的应用。给定一组城市和它们之间的距离或成本，TSP 的目标是找到一条最短路径，使得旅行商从起始城市出发，经过每个城市恰好一次，最后回到起始城市。

回溯法是一种解决组合优化问题的常用方法，它通过穷举所有可能的解空间，并利用剪枝策略来减少搜索空间的规模，从而找到最优解。在 TSP 问题中，回溯法通过递归地遍历所有可能的路径来找到最短路径。

2.1.2 算法描述

Algorithm 1 TSP 问题的回溯法求解

```
sum_node ← sum_node + 1                                                              ▷ 搜索节点数加 1
if count = |graph| then
    if graph[current_city][start_city] ≠ NO_EDGE and now_cost + graph[current_city][start_city] < min_cost then
        min_cost ← now_cost + graph[current_city][start_city]
        best_path ← path
        best_path.append(start_city)
    return
for i 从 0 到 |graph| do
    if graph[current_city][i] ≠ NO_EDGE and !vis[i] and graph[current_city][i] + now_cost < min_cost then
        vis[i] ← true                                                                ▷ 标记为已访问
        path.append(i)                                                               ▷ 加入路径
        TSP(count + 1, i, start_city, min_cost, now_cost + graph[current_city][i], best_path, path, vis, graph, sum_node)
        path.pop_back()                                                              ▷ 回溯
        vis[i] ← false
```

2.1.3 分析和改进

回溯法求解 TSP 问题的时间复杂度和空间复杂度如下：

- 时间复杂度：回溯法的时间复杂度是指数级别的，因为它需要遍历所有可能的解空间。在 TSP 问题中，假设有 n 个城市，则解空间的规模是 $n!$。因此，回溯法的时间复杂度为 $O(n!)$。然而，由于算法使用了剪枝策略（当当前花费加上当前节点到下一个节点的花费大于等于最小花费时，不再考虑某些状态），实际的时间复杂度可能会低于 $O(n!)$。
- 空间复杂度：回溯法的空间复杂度取决于递归调用的深度，即解空间的深度。在 TSP 问题中，递归调用的深度最多为 n，即遍历所有城市一次。此外，还需要使用额外的空间存储路径、访问标记和邻接矩阵等信息。使用了两个长度为 n 的数组来记录当前的路径和访问状态，以及一个长度为 n 的数组来存储最优路径，其中 n 是城市的个数。，所以回溯法的空间复杂度为 $O(n)$。请注意，这个分析是基于最坏情况的。由于剪枝策略的使用，实际的时间和空间需求可能会低于这个上限。

回溯法通过穷举搜索的方式可以找到 TSP 问题的最优解，但由于其时间复杂度的指数级增长，对于大规模问题可能会面临计算资源和时间的限制。在我的计算机上，对于超过 20 个节点的图，使用回溯法会爆栈，因此，我设计了模拟递归栈，消除递归的改进版本，详细代码见附录。

2.2 分支限界法求解 TSP

2.2.1 介绍

旅行商问题（Traveling Salesman Problem，TSP）是一个经典的组合优化问题，目标是找到一条路径，使得旅行商从起始城市出发，经过所有城市恰好一次，然后返回起始城市，并且总路径长度最小。TSP 在计算机科学和运筹学中具有重要的应用，它是一个 NP 困难问题，没有已知的多项式时间解法。分支限界法是一种常用的求解 TSP 问题的方法，通过不断剪枝和搜索空间的划分，寻找最优解。

2.2.2 算法描述（含算法框架）

算法框架以最小化问题为例，e.g. TSP 问题

1. 选择初始解对应的根节点 v_0，根据限界函数，估计根节点的目标函数上下界, 确定目标函数的界 [lowerBound, upperBound] 问题最优的上下界
2. 将活结点表 ANT 初始化为空
3. 生成根结点 v_0 的全部子结点-宽度优先；
   对每个子结点 v，执行以下操作
   (a) 估算 v 的目标函数值（下界）calculateLowerBound(v)
   (b) 若 calculateLowerBound(v)<= upperBound，将 v 加入 ANT 表
4. 循环，直到某个叶结点的目标函数值在表 ANT 中最小（找到 1 个具有最小值的完全解）
   (a) 从表 ANT 中选择（下界）lowerBound(v_i) 值最小的结点 v_i，扩展其子结点（从活结点表中，选择 1 个具有最小可能目标值的扩展结点 v_i）
   (b) 对结点 v_i 的每个子结点 c，执行下列操作
      i. 估算 c 的目标函数值 calculateLowerBound(v)，即下界
      ii. 如果 calculateLowerBound(v)<= upperBound，将 c 加入 ANT 表（子结点 c 有可能产生更优的解，将其加入活结点表，以后考虑对其进行扩展）
      iii. 如果 c 是叶结点且 lowerBound(c) 在表 ANT 中最小，则将结点 c 对应的完全解输出，算法结束（结点 c 对应了 1 个新找到的、具有最小目标值（e.g. TSP 路径长度）的完全解——最优解）
      iv. 如果 c 是叶结点但 lowerBound(c) 在表 ANT 中不是最小，则：
      结点 c 对应了 1 个新找到的完全解，但该完全解的目标函数值与已经找到的、或未来可能找到完全解相比，并非更优
      i) upperBound = value(c)
      ii) 对表 ANT 中所有满足 calculateLowerBound(v_j) > value(c) 的结点 v_j，从表 ANT 中删除该结点！

下面是使用分支限界法求解 TSP 问题的伪代码：

Algorithm 2 TSP 分支限界法

```
Initialize priority queue pq                          ▷ 优先队列，按照 lowerBound 从小到大排序
Initialize boolean array vis                          ▷ 标记节点是否被访问过
Mark start_city as visited                            ▷ 起始节点标记为已访问
Initialize empty path path                            ▷ 当前路径
Add start_city to path                                ▷ 起始节点加入路径
Calculate lowerBound                                  ▷ 计算当前路径的下界
Add State(start_city, 1, 0, lowerBound, path, vis) to pq   ▷ 起始节点加入优先队列
while pq is not empty do
    Get State State from pq                           ▷ 取出优先队列中的第一个状态
    Remove State from pq
    Increment sum_node                                ▷ 搜索节点数加 1
    if State.lowerBound > upperBound then             ▷ 如果当前状态的下界大于上界，剪枝
        Continue to next iteration
    if State.count = graph.size() then                ▷ 所有节点都访问过了，回到起始节点
        if graph[State.current_city][start_city] ≠ NO_EDGE and State.now_cost + graph[State.current_city][start_city] < min_cost then
            Update min_cost, best_path
            Add start_city to best_path
            if min_cost ≤ State.lowerBound then       ▷ 如果最小花费小于等于下界，直接返回，不再继续搜索，因为已经找到了最优解
                Return
            if min_cost < upperBound then             ▷ 如果最小花费小于上界，更新上界和 pq
                Update upperBound
                Update pq
        Continue to next iteration
    for i = 0 to graph.size() do
        if graph[State.current_city][i] ≠ NO_EDGE and !State.vis[i] and graph[State.current_city][i] + State.now_cost < min_cost then
            Initialize new path newPath               ▷ 新路径
            Copy State.path to newPath                ▷ 加入下一个节点
            Initialize new boolean array newVis       ▷ 新的标记数组
            Copy State.vis to newVis                  ▷ 下一个节点标记为已访问
            Calculate lowerBound
            if lowerBound ≤ upperBound then           ▷ 如果下界小于等于上界，加入优先队列
                Add State(i, State.count + 1, State.now_cost + graph[State.current_city][i], lowerBound, newPath, newVis) to pq
```

2.2.3 算法分析

时间复杂度

算法的时间复杂度主要取决于两个部分：计算下界和搜索过程。

- 计算下界：calculateLowerBound 函数用于计算当前路径的下界。在最坏情况下，需要遍历每个节点，计算每个节点的最短路径和次短路径。由于预处理好了 $min2$ 数组，因此时间复杂度为 $O(n)$，其中 n 是节点的数量。
- 搜索过程：在最坏情况下，需要遍历所有可能的路径，即 $n!$ 个排列（n 是节点的数量）。每个节点都需要检查与其他节点的连边，因此时间复杂度为 $O(n^2)$。因此，整个搜索过程的时间复杂度为 $O(n! * n)$。

综上所述，算法的时间复杂度为 $O(n! * n)$。然而，由于算法使用了剪枝策略（当下界大于上界时，不再考虑某些状态），实际的时间复杂度会低于 $O(n! * n)$

空间复杂度

算法使用了以下额外空间：

- 优先队列：在最坏情况下，优先队列的大小可以达到 n!，因此空间复杂度为 $O(n!)$。
- vis 数组、path 数组和 newPath 数组：这些数组的大小与节点的数量 n 相同，因此空间复杂度为 $O(n)$。
- min2 二维数组和 graph 二维数组：这些二维数组的大小为 $n * 2$ 和 $n * n$，因此空间复杂度为 $O(n^2)$。

综上所述，算法的空间复杂度为 O(n!)，其中 n 是节点的数量。然而，这个分析是基于最坏情况的。由于剪枝策略的使用，实际的时间和空间需求可能会低于这个上限。此外，这个算法的效率也取决于 calculateLowerBound 函数的收益，该函数用于计算每个状态的下界。如果这个函数能够快速并准确地估计下界，那么算法的效率将会提高。

3 实验结果

3.1 回溯法

```
filename: 15.txt
min_cost: 5506.88
best_path: 20 9 7 16 3 13 12 21 10 8 19 11 22 5 17 20
time: 0.0083363s
search node: 256955

filename: 20.txt
min_cost: 6987.51
best_path: 20 9 7 16 3 13 2 15 12 14 21 10 1 8 18 19 11 22 5 17 20
time: 2.91684s
search node: 76329668

filename: 22.txt
min_cost: 7690.8
```

```
best_path: 20 9 7 16 3 13 2 15 12 14 21 10 1 4 6 18 8 19 11 22 5 17 20
time: 20.1967s
search node: 487370492

filename: 30.txt
min_cost: 11426.6
best_path: 20 15 25 26 27 21 28 23 19 9 3 5 6 7 30 13 1 2 14 10 17 4 29 24 11 18 16
 8 22 12 20
time: 142.337s
search node: 3893952727
```

3.2 分支限界法

```
filename: 15.txt
min_cost: 5506.88
best_path: 20 17 5 22 11 19 8 10 21 12 13 3 16 7 9 20
time: 0.0081651s
search node: 5422

filename: 20.txt
min_cost: 6987.51
best_path: 20 9 7 16 3 13 2 15 12 14 21 10 1 8 18 19 11 22 5 17 20
time: 0.0539328s
search node: 28368

filename: 22.txt
min_cost: 7690.8
best_path: 20 9 7 16 3 13 2 15 12 14 21 10 1 4 6 18 8 19 11 22 5 17 20
time: 0.0225743s
search node: 12594

filename: 30.txt
min_cost: 11426.6
best_path: 20 15 25 26 27 21 28 23 19 9 3 5 6 7 30 13 1 2 14 10 17 4 29 24 11 18 16
 8 22 12 20
time: 31.8229s
search node: 11135403
```

3.3 表格记录

问题	求解算法	最短回路	路径总长度(单位：m)	搜索过的结点总数	程序运行时间(单位：s)

15 个基站	回溯	20 9 7 16 3 13 12 21 10 8 19 11 22 5 17 20	5506.88	256955	0.0083363
	分支限界	20 17 5 22 11 19 8 10 21 12 13 3 16 7 9 20	5506.88	5422	0.0081651
20 个基站	回溯	20 9 7 16 3 13 2 15 12 14 21 10 1 8 18 19 11 22 5 17 20	6987.51	76329668	2.91684
	分支限界	20 9 7 16 3 13 2 15 12 14 21 10 1 8 18 19 11 22 5 17 20	6987.51	28368	0.0539328
22 个基站	回溯	20 9 7 16 3 13 2 15 12 14 21 10 1 4 6 18 8 19 11 22 5 17 20	7690.8	487370492	20.1967
	分支限界	20 9 7 16 3 13 2 15 12 14 21 10 1 4 6 18 8 19 11 22 5 17 20	7690.8	12594	0.0225743
30 个基站	回溯	20 15 25 26 27 21 28 23 19 9 3 5 6 7 30 13 1 2 14 10 17 4 29 24 11 18 16 8 22 12 20	11426.6	3893952727	142.337
	分支限界	20 15 25 26 27 21 28 23 19 9 3 5 6 7 30 13 1 2 14 10 17 4 29 24 11 18 16 8 22 12 20	11426.6	11135403	31.8229

4 附录：完整代码

4.1 回溯法

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <stack>
#include <chrono>

#define NO_EDGE 99999

/*
 * fileName: 输入文件名
 * n: 节点数
 * graph: 邻接矩阵
 * id2index: id到下标的映射
 */
void input(const char *fileName, int &n, std::vector<std::vector<double>> &graph,
    std::unordered_map<int, int> &id2index)
{
    std::ifstream inputFile(fileName);
    if (!inputFile)
    {
        std::cout << "File␣not␣found!" << std::endl;
        return;
    }

    std::string line;
    std::getline(inputFile, line); // 读取第一行（编号行）
    for (int i = 0; i < line.size(); ++i)
        if (line[i] == '\t')
            ++n;
    n--;
    std::stringstream lineStream(line);

    for (int i = 0; i < n; ++i)
    {
        std::string temp;
        lineStream >> temp;
        id2index[i] = std::stoi(temp);
    }

    // 读取id行并忽略
    std::getline(inputFile, line);
```

```cpp
    // 读取边的权值矩阵
    graph.resize(n, std::vector<double>(n));
    for (int i = 0; i < n; ++i)
    {
        std::getline(inputFile, line);
        std::stringstream lineStream(line);
        // 忽略前2列 (编号和id)
        std::string temp;
        lineStream >> temp;
        lineStream >> temp;
        for (int j = 0; j < n; ++j)
        {
            lineStream >> graph[i][j];
        }
    }

    inputFile.close();
}

/*
 * count: 当前已经访问的节点数
 * current_city: 当前所在的节点
 * start_city: 起始节点
 * min_cost: 最小花费
 * now_cost: 当前花费
 * vis: 标记是否访问过
 * graph: 邻接矩阵
 * sum_node: 搜索节点数
 */
void TSP(int count, int current_city, int start_city, double &min_cost, double
    now_cost, std::vector<int> &best_path, std::vector<int> &path, std::vector<bool>
     &vis, std::vector<std::vector<double>> &graph, long long &sum_node)
{
    sum_node++;                  // 搜索节点数加1
    if (count == graph.size()) // 所有节点都访问过了, 回到起始节点
    {
        // 如果当前节点到起始节点有边, 且当前花费加上当前节点到起始节点的花费小于最
           小花费
        if (graph[current_city][start_city] != NO_EDGE && now_cost + graph[
            current_city][start_city] < min_cost)
        {
            min_cost = now_cost + graph[current_city][start_city];
            best_path = path;
            best_path.push_back(start_city);
        }
        return;
```

```
    }

    for (int i = 0; i < graph.size(); ++i)
    {
        // 如果当前节点到下一个节点有边，且下一个节点未访问过，且当前花费加上当前节
            点到下一个节点的花费小于最小花费
        if (graph[current_city][i] != NO_EDGE && !vis[i] && graph[current_city][i]
            + now_cost < min_cost)
        {
            vis[i] = true;      // 标记为已访问
            path.push_back(i); // 加入路径
            TSP(count + 1, i, start_city, min_cost, now_cost + graph[current_city][
                i], best_path, path, vis, graph, sum_node);
            path.pop_back(); // 回溯
            vis[i] = false;
        }
    }
}

struct State // 栈中状态，引用变量不用拷贝
{
    int count;
    int current_city;
    int start_city;
    double now_cost;
    int last_index; // 上一个节点在path中的下标
    bool back_flag; // 是否是回溯
    // 构造函数
    State(int count, int current_city, int start_city, double now_cost, int
        last_index, bool back_flag) : count(count), current_city(current_city),
        start_city(start_city), now_cost(now_cost), last_index(last_index),
        back_flag(back_flag) {}
};
/*
 * count: 当前已经访问的节点数
 * current_city: 当前所在的节点
 * start_city: 起始节点
 * min_cost: 最小花费
 * now_cost: 当前花费
 * vis: 标记是否访问过
 * graph: 邻接矩阵
 */
long long TSP_stack(int count, int current_city, int start_city, double &min_cost,
    double now_cost, std::vector<int> &best_path, std::vector<int> &path, std::
    vector<bool> &vis, std::vector<std::vector<double>> &graph)
{
    long long sum_node = 0; // 搜索节点数
```

```
std::stack<State, std::vector<State>> state_stack;
state_stack.push(State(count, current_city, start_city, now_cost, 0, false));
sum_node++;
while (!state_stack.empty())
{
    State &state = state_stack.top();
    switch (state.back_flag)
    {
    case false: // 前进
    {
        if (state.count == graph.size())
        {
            if (graph[state.current_city][state.start_city] != NO_EDGE && state
                .now_cost + graph[state.current_city][state.start_city] <
                min_cost)
            {
                min_cost = state.now_cost + graph[state.current_city][state.
                    start_city];
                best_path = path;
                best_path.push_back(state.start_city);
            }
            state_stack.top().back_flag = true;
            break;
        }
        bool forward_flag = false;
        for (int i = 0; i < graph.size(); ++i)
        {
            if (graph[state.current_city][i] != NO_EDGE && !vis[i] && graph[
                state.current_city][i] + state.now_cost < min_cost)
            {
                vis[i] = true;      // 标记为已访问
                path.push_back(i); // 加入路径
                state_stack.push(State(state.count + 1, i, start_city, state.
                    now_cost + graph[state.current_city][i], i, false));
                forward_flag = true; // 有前进
                sum_node++;          // 搜索节点数加1
                break;
            }
        }
        if (!forward_flag) // 没有前进，回溯
            state_stack.top().back_flag = true;
        break;
    }
    case true: // 回溯
    {
        int last_index = state.last_index; // 上一个节点在path中的下标
        vis[last_index] = false;           // 标记为未访问
```

```
            path.pop_back();                    // 从路径中删除
            state_stack.pop();                  // 从栈中删除
            if (state_stack.empty())            // 栈空，退出
                break;
            state = state_stack.top(); // state维护的是栈顶元素的引用，所以要更新state
            bool forward_flag = false;
            for (int i = last_index + 1; i < graph.size(); ++i) // 从上一个节点的下一个节点开始找
            {
                if (graph[state.current_city][i] != NO_EDGE && !vis[i] && graph[state.current_city][i] + state.now_cost < min_cost)
                {
                    vis[i] = true;     // 标记为已访问
                    path.push_back(i); // 加入路径
                    state_stack.push(State(state.count + 1, i, start_city, state.now_cost + graph[state.current_city][i], i, false));
                    forward_flag = true; // 有前进
                    sum_node++;           // 搜索节点数加1
                    break;
                }
            }
            if (!forward_flag) // 没有前进，回溯
                state_stack.top().back_flag = true;
        }
        }
    }
    return sum_node; // 返回搜索节点数
}

// fileName: 输入文件名
void solve(const char *fileName, int start)
{
    std::cout << "filename:␣" << fileName << std::endl; // 输出文件名
    int n = 0;                                          // 节点数
    std::vector<std::vector<double>> graph;             // 邻接矩阵
    std::unordered_map<int, int> id2index;              // id到下标的映射
    input(fileName, n, graph, id2index);                // 读取输入文件
    double min_cost = NO_EDGE * n;                      // 最小花费
    std::vector<int> best_path(n + 1);                  // 最优路径，图总共n个点，回到起点又是一个点，所以是n+1个点
    int start_node = start;                             // 起始节点
    std::vector<int> path;                              // 当前路径
    std::vector<bool> vis(n, false);                    // 标记是否访问过
    vis[start_node] = true;                             // 起始节点标记为已访问
    path.push_back(start_node);                         // 起始节点加入路径
```

```
    long long sum_node = 0;                                      // 搜索节点数，初
        始化为0，longlong防止溢出
    auto start_time = std::chrono::high_resolution_clock::now(); // 计时开始
    // TSP(1, start_node, start_node, min_cost, 0, best_path, path, vis, graph,
        sum_node);
    sum_node = TSP_stack(1, start_node, start_node, min_cost, 0, best_path, path,
        vis, graph);
    auto end_time = std::chrono::high_resolution_clock::now();      // 计时结束
    std::chrono::duration<double> duration = end_time - start_time; // 计算耗时

    std::cout << "min_cost:␣" << min_cost << std::endl;
    std::cout << "best_path:␣";
    for (int i = 0; i < best_path.size(); ++i) // 输出最短路径
    {
        std::cout << id2index[best_path[i]] << "␣"; // 输出id
        // std::cout << best_path[i] << " ";           // 输出下标
    }
    std::cout << std::endl
              << "time:␣" << duration.count() << "s" << std::endl;
    std::cout << "search␣node:␣" << sum_node << std::endl; // 输出搜索节点数
}

int main() // 由于用了大量stl，编译时请使用"-O1"或更高级优化选项
{
    freopen("output1.txt", "w", stdout);              // 将输出重定向到output.txt
    std::unordered_map<std::string, int> file2start; // 文件名到起始节点的映射
    file2start["15.txt"] = 12;
    file2start["20.txt"] = 17;
    file2start["22.txt"] = 19;
    file2start["30.txt"] = 19;

    solve("15.txt", file2start["15.txt"]);
    std::cout << std::endl;

    solve("20.txt", file2start["20.txt"]);
    std::cout << std::endl;

    solve("22.txt", file2start["22.txt"]);
    std::cout << std::endl;

    solve("30.txt", file2start["30.txt"]);
    std::cout << std::endl;
    fclose(stdout);
    return 0;
}
```

4.2 分支限界法

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <queue>
#include <chrono>

#define NO_EDGE 99999

/*
 * fileName: 输入文件名
 * n: 节点数
 * graph: 邻接矩阵
 * id2index: id到下标的映射
 */
void input(const char *fileName, int &n, std::vector<std::vector<double>> &graph,
    std::unordered_map<int, int> &id2index, std::vector<std::vector<double>> &min2)
{
    std::ifstream inputFile(fileName);
    if (!inputFile)
    {
        std::cout << "File␣not␣found!" << std::endl;
        return;
    }

    std::string line;
    std::getline(inputFile, line); // 读取第一行（编号行）
    for (int i = 0; i < line.size(); ++i)
        if (line[i] == '\t')
            ++n;
    n--;
    min2.resize(n, std::vector<double>(2, NO_EDGE)); // 初始化min2，n行2列，每个元
        素初始化为NO_EDGE
    std::stringstream lineStream(line);

    for (int i = 0; i < n; ++i)
    {
        std::string temp;
        lineStream >> temp;
        id2index[i] = std::stoi(temp);
    }

    // 读取id行并忽略
```

```cpp
    std::getline(inputFile, line);

    // 读取边的权值矩阵
    graph.resize(n, std::vector<double>(n));
    for (int i = 0; i < n; ++i)
    {
        std::getline(inputFile, line);
        std::stringstream lineStream(line);
        // 忽略前2列 (编号和id)
        std::string temp;
        lineStream >> temp;
        lineStream >> temp;
        for (int j = 0; j < n; ++j)
        {
            lineStream >> graph[i][j];
            if (graph[i][j] < min2[i][0]) // 比最小的小, 更新最小的和次小的
            {
                min2[i][1] = min2[i][0];
                min2[i][0] = graph[i][j];
            }
            else if (graph[i][j] < min2[i][1]) // 比次小的小, 更新次小的
            {
                min2[i][1] = graph[i][j];
            }
        }
    }

    inputFile.close();
}

/*
 * 求与当前节点最近的未访问过的节点的下标
 * graph: 邻接矩阵
 * vis: 标记节点是否被访问过
 * current_city: 当前所在的城市
 * exclude: 排除的节点
 * 返回值: 下一个城市的下标
 */
int findMin(std::vector<std::vector<double>> &graph, std::vector<bool> &vis, int
    current_city, std::vector<bool> &exclude)
{
    double min_edge = NO_EDGE; // 最小边, 初始化
    int next_city = -1;
    for (int i = 0; i < graph.size(); ++i)
    {
        // 如果当前节点到i有边小于最小值, 且i没有被访问过, 且i不在排除的节点中
        if (graph[current_city][i] < min_edge && !vis[i] && !exclude[i])
```

```
        {
            min_edge = graph[current_city][i]; // 维护最小边
            next_city = i;
        }
    }
    return next_city;
}

/*
 * 计算当前路径的上界
 * graph: 邻接矩阵
 * vis: 标记节点是否被访问过
 * current_city: 当前所在的城市
 * start_city: 起始城市
 * count: 已经访问过的节点数
 * 返回值: 当前路径的上界
 */
double calculateUpperBound(std::vector<std::vector<double>> &graph, std::vector<
    bool> &vis, int current_city, int start_city, int count)
{
    if (count == graph.size())
    {
        if (graph[current_city][start_city] != NO_EDGE)
        {
            return graph[current_city][start_city];
        }
        else
            return -1;
    }
    std::vector<bool> exclude(graph.size(), false); // 排除的节点
    int next_city = findMin(graph, vis, current_city, exclude);
    while (next_city != -1)
    {
        vis[next_city] = true;
        double temp = calculateUpperBound(graph, vis, next_city, start_city, count
            + 1);
        if (temp != -1) // 找到了一条路径
            return temp + graph[current_city][next_city];
        vis[next_city] = false; // 回溯
        exclude[next_city] = true;
        next_city = findMin(graph, vis, current_city, exclude);
    }
    return -1; // 没有找到路径, 返回-1, 回溯
}

/*
 * 计算当前路径的下界
```

```
 * graph: 邻接矩阵
 * min2: 存储每个节点最短的2条路径的长度
 * path: 当前路径
 * 返回值: 当前路径的下界
 */
double calculateLowerBound(std::vector<std::vector<double>> &graph, std::vector<std
    ::vector<double>> &min2, std::vector<int> &path, std::vector<bool> &vis)
{
    int u = path[0], v = path[path.size() - 1]; // u, v记录当前路径的起点和终点
    double lowerBound = 0;
    int n = graph.size();
    for (int i = 0; i < path.size() - 1; i++) // 计算当前路径的花费
        lowerBound += graph[path[i]][path[i + 1]];
    lowerBound *= 2;
    if (path.size() >= 2) // 路径中有2个点以上
    {
        double min_edge1 = NO_EDGE, min_edge2 = NO_EDGE; // 分别记录回到u的最小边和
            从v出发的最小边
        for (int i = 0; i < n; i++)
        {
            if (graph[i][u] < min_edge1)
                min_edge1 = graph[i][u];
            if (graph[v][i] < min_edge2)
                min_edge2 = graph[v][i];
        }
        lowerBound += min_edge1 + min_edge2;
    }
    else // 只有1个点, 直接加上最小的2条边
    {
        lowerBound += min2[u][0] + min2[u][1];
    }
    for (int i = 0; i < n; i++)
    {
        if (!vis[i]) // 如果i不在当前路径中
        {
            lowerBound += min2[i][0] + min2[i][1]; // 加上i的最小的2条边
        }
    }
    return lowerBound / 2;
}

struct State
{
    int current_city;       // 当前所在的城市
    int count;              // 已经访问过的节点数
    double now_cost;        // 当前路径的花费
    double lowerBound;      // 当前路径的下界
```

```
    std::vector<int> path; // 当前路径
    std::vector<bool> vis; // 标记节点是否被访问过
    // 构造函数
    State(int current_city, int count, double now_cost, double lowerBound, std::
        vector<int> &path, std::vector<bool> &vis)
        : current_city(current_city), count(count), now_cost(now_cost), lowerBound(
            lowerBound), path(path), vis(vis) {}

    // 重载大于号，按照lowerBound从小到大排序
    friend bool operator>(const State &s1, const State &s2)
    {
        return s1.lowerBound > s2.lowerBound;
    }
};

/*
 * start_city: 起始节点
 * min_cost: 最小花费
 * best_path: 最优路径
 * graph: 邻接矩阵
 * min2: 存储每个节点最短的2条路径的长度
 * sum_node: 搜索节点数
 * upperBound: 上界
 */
void TSP(int start_city, double &min_cost, std::vector<int> &best_path, std::vector
    <std::vector<double>> &graph, std::vector<std::vector<double>> &min2, long long
    &sum_node, double upperBound)
{
    std::priority_queue<State, std::vector<State>, std::greater<State>> pq; // 优先
        队列，按照lowerBound从小到大排序
    std::vector<bool> vis(graph.size(), false);                             // 标记
        节点是否被访问过
    vis[start_city] = true;                                                 // 起始
        节点标记为已访问
    std::vector<int> path;                                                  // 当前
        路径
    path.push_back(start_city);                                             // 起始
        节点加入路径
    double lowerBound = calculateLowerBound(graph, min2, path, vis);        // 计算
        当前路径的下界
    // std::cout << "lowerBound: " << lowerBound << std::endl;                 //
        输出下界
    pq.push(State(start_city, 1, 0, lowerBound, path, vis));                // 起始
        节点加入优先队列
    while (!pq.empty())
    {
        State state = pq.top(); // 取出优先队列中的第一个状态
```

```
        pq.pop();
        sum_node++;                         // 搜索节点数加1
        if (state.lowerBound > upperBound) // 如果当前状态的下界大于上界，剪枝
            continue;
        if (state.count == graph.size()) // 所有节点都访问过了，回到起始节点
        {
            // 如果当前节点到起始节点有边，且当前花费加上当前节点到起始节点的花费小
                于最小花费
            if (graph[state.current_city][start_city] != NO_EDGE && state.now_cost
                + graph[state.current_city][start_city] < min_cost)
            {
                min_cost = state.now_cost + graph[state.current_city][start_city];
                best_path = state.path;
                best_path.push_back(start_city);
                if (min_cost <= state.lowerBound) // 如果最小花费小于等于下界，直接
                    返回，不再继续搜索，因为已经找到了最优解
                    return;
                if (min_cost < upperBound) // 如果最小花费小于上界，更新上界，并删
                    除优先队列中大于当前最小花费的状态
                {
                    upperBound = min_cost;
                    std::vector<State> temp;
                    while (!pq.empty())
                    {
                        State s = pq.top();
                        pq.pop();
                        if (s.lowerBound < upperBound)
                            temp.push_back(s);
                    }
                    for (int i = 0; i < temp.size(); ++i)
                        pq.push(temp[i]);
                }

            }
            continue;
        }
        for (int i = 0; i < graph.size(); ++i)
        {
            // 如果当前节点到下一个节点有边，且下一个节点未访问过，且当前花费加上当
                前节点到下一个节点的花费小于最小花费
            if (graph[state.current_city][i] != NO_EDGE && !state.vis[i] && graph[
                state.current_city][i] + state.now_cost < min_cost)
            {
                std::vector<int> newPath = state.path; // 新路径
                newPath.push_back(i);                  // 加入下一个节点
                std::vector<bool> newVis = state.vis;  // 新的标记数组
                newVis[i] = true;                      // 下一个节点标记为已访问
```

```
                double lowerBound = calculateLowerBound(graph, min2, newPath,
                    newVis);
                if (lowerBound <= upperBound) // 如果下界小于等于上界，加入优先队列
                    pq.push(State(i, state.count + 1, state.now_cost + graph[state.
                        current_city][i], lowerBound, newPath, newVis));
            }
        }
    }
}

void solve(const char *fileName, int start)
{
    std::cout << "filename:␣" << fileName << std::endl;
                                    // 输出文件名
    int n = 0, start_node = start;
                                                        // 节点数，起始节点
    std::vector<std::vector<double>> graph;
                                                // 邻接矩阵
    std::unordered_map<int, int> id2index;
                                                // id到下标的映射
    std::vector<std::vector<double>> min2;
                                                // 存储每个节点最短的2条路径的长度
    input(fileName, n, graph, id2index, min2);
                                            // 读取输入文件
    std::vector<bool> vis(n, false);
                                                    // 标记节点是否被访问过
    vis[start_node] = true;
                                                            // 起始节点标记为已
        访问
    double upperBound = calculateUpperBound(graph, vis, start_node, start_node, 1);
         // 计算上界
    // std::cout << "upperBound: " << upperBound << std::endl;
                                // 输出上界

    long long sum_node = 0;                                 // 搜索节点数，初
        始化为0，longlong防止溢出
    double min_cost = NO_EDGE * n;                          // 最小花费
    std::vector<int> best_path(n + 1);                      // 最优路径，图总
        共n个点，回到起点又是一个点，所以是n+1个点
    vis[start_node] = true;                                 // 起始节点标记为
        已访问
    auto start_time = std::chrono::high_resolution_clock::now(); // 计时开始
    TSP(start_node, min_cost, best_path, graph, min2, sum_node, upperBound);
    auto end_time = std::chrono::high_resolution_clock::now();     // 计时结束
    std::chrono::duration<double> duration = end_time - start_time; // 计算耗时

    std::cout << "min_cost:␣" << min_cost << std::endl;
```

```
    std::cout << "best_path:␣";
    for (int i = 0; i < best_path.size(); ++i)
        std::cout << id2index[best_path[i]] << "␣";
    std::cout << std::endl
              << "time:␣" << duration.count() << "s" << std::endl;
    std::cout << "search␣node:␣" << sum_node << std::endl; // 输出搜索节点数
}

int main() // 使用了大量stl, 编译时请使用"-O1"或更高级优化选项
{
    freopen("output2.txt", "w", stdout);              // 输出重定向到output.txt
    std::unordered_map<std::string, int> file2start; // 文件名到起始节点的映射
    file2start["15.txt"] = 12;
    file2start["20.txt"] = 17;
    file2start["22.txt"] = 19;
    file2start["30.txt"] = 19;

    solve("15.txt", file2start["15.txt"]);
    std::cout << std::endl;

    solve("20.txt", file2start["20.txt"]);
    std::cout << std::endl;

    solve("22.txt", file2start["22.txt"]);
    std::cout << std::endl;

    solve("30.txt", file2start["30.txt"]);
    std::cout << std::endl;
    fclose(stdout);
    return 0;
}
```