《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：张梓良

学号：2021212484

算法设计与分析（第 5-6 章）：回溯、分支限界编程实验报告

张梓良　2021212484

日期：2023 年 12 月 25 日

1　概述

1.1　实验内容及要求

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
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

1.2　实验环境

- gcc version 8.1.0
- Visual Studio Code 1.82.2
- python 3.9.13
- Anaconda 22.9.0
- OS: Windows_NT x64 10.0.22621

2　数据预处理

通过以下 python 代码将所给的 xls 格式的数据转换为 txt 格式，便于 C++ 代码读入数据。

```
import xlrd

# 打开xls文件
workbook = xlrd.open_workbook('DATA/data.xls')

# 读入15个基站的数据
worksheet = workbook.sheet_by_index(0)
with open('DATA/tsp_15.txt', 'w') as file:
    for i in range(2, 17):
        for j in range(2, 17):
```

```
            file.write(str(worksheet.cell_value(i, 0)) + '␣' + str(worksheet.cell_
                value(0, j)) + '␣' + str(worksheet.cell_value(i, j)) + '\n')

# 读入20个基站的数据
worksheet = workbook.sheet_by_index(1)
with open('DATA/tsp_20.txt', 'w') as file:
    for i in range(2, 22):
        for j in range(2, 22):
            file.write(str(worksheet.cell_value(i, 0)) + '␣' + str(worksheet.cell_
                value(0, j)) + '␣' + str(worksheet.cell_value(i, j)) + '\n')

# 读入22个基站的数据
worksheet = workbook.sheet_by_index(2)
with open('DATA/tsp_22.txt', 'w') as file:
    for i in range(2, 24):
        for j in range(2, 24):
            file.write(str(worksheet.cell_value(i, 0)) + '␣' + str(worksheet.cell_
                value(0, j)) + '␣' + str(worksheet.cell_value(i, j)) + '\n')

# 读入30个基站的数据
worksheet = workbook.sheet_by_index(3)
with open('DATA/tsp_30.txt', 'w') as file:
    for i in range(2, 32):
        for j in range(2, 32):
            file.write(str(worksheet.cell_value(i, 0)) + '␣' + str(worksheet.cell_
                value(0, j)) + '␣' + str(worksheet.cell_value(i, j)) + '\n')

print('Finished␣writing␣data␣to␣file.')
```

3 分支限界法算法框架

以最小化问题为例，e.g. TSP

1. 选择初始解对应的根节点 v0，根据限界函数 bound，估计根节点的目标函数上下界 bound(v0)，确定目标函数的界 [down, up]
   - 最优目标解的上下界
2. 将活结点表 ANT 初始化为空
3. 生成根结点 v0 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）bound(v)
   (b) 若 bound(v)<=up，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）bound(vi) 值最小的结点 vi ，扩展其子结点
   (b) 对结点 vi 的每个子结点 c，执行下列操作
      i. 估算 c 的目标函数值 bound(c)—下界

ii. 如果 bound(c)<=up，将 c 加入 ANT 表

iii. 如果 c 是叶结点 and bound(c) 在表 ANT 中最小，则将结点 c 对应的完全解输出，算法结束

iv. 如果 c 是叶结点 and bound(c) 在表 ANT 中不是最小，则：

A. 令 up=bound(c)

B. 对表 ANT 中所有满足 bound(vj)>up=bound(c) 的结点 vj，从表 ANT 中删除该结点

4 回溯法

4.1 算法设计

4.1.1 算法逻辑

以深度优先的方式，从树根结点开始，依次扩展树结点，直到达到叶结点。搜索过程中动态产生解空间。

剪枝条件:

- 如果当前正在考虑的顶点 j 与当前已经走过的部分路径中末端结点 i（代表没有走过的其它城市）没有边相连，即 $w[i,j]=\infty$, 则不必搜索 j 所在分支
- 如果 $B(i) \geq bestw$—$x[i]$ 无希望达到更优的路径，则停止搜索 $x[i]$ 分支及其下面的层，其中，$bestw$ 代表到目前为止, 在前面的搜索中，从其它已经搜索过的路径中，找到的最佳完整回路的权和（总长度）

4.1.2 核心代码

```
void BacktrackTSP(int i)
{
    if (i == n) // 搜索到叶节点
    {
        if (w[x[n - 1]][x[n]] != INF && w[x[n]][start] != INF && cw + w[x[n - 1]][x
            [n]] + w[x[n]][start] < bestw)
        {
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
            bestw = cw + w[x[n - 1]][x[n]] + w[x[n]][start];
        }
    }
    else // 搜索非叶节点
    {
        for (int j = i; j <= n; j++)
        {
            if (w[x[i - 1]][x[j]] != INF && cw + w[x[i - 1]][x[j]] < bestw)
            {
```

```
                L++;
                swap(x[i], x[j]);          // 交换，将x[j]加入到当前解中
                cw += w[x[i - 1]][x[i]]; // 更新当前路径长度
                BacktrackTSP(i + 1);       // 递归搜索
                cw -= w[x[i - 1]][x[i]]; // 回溯
                swap(x[i], x[j]);
            }
        }
    }
}
```

4.2 算法性能分析

时间复杂度

- 在叶节点时，需要进行一次判断操作，时间复杂度为 $O(1)$。
- 在非叶节点时，需要通过一个循环遍历可能的解空间，每次进行一次判断、交换、路径长度更新、递归搜索以及回溯操作。循环的次数取决于 $n-i+1$，其中 n 为城市总数，i 为当前已经访问过的城市数。因此，循环的总时间复杂度为$O(n!)$，其中 n 为城市总数。

空间复杂度

- 在回溯的过程中，主要占用的空间为递归调用栈和一些辅助变量的存储。在最坏情况下，递归调用栈的深度为 n，因此空间复杂度为 $O(n)$。此外，还需要使用一个路径数组来存储当前已经访问的城市顺序，这个数组的长度为 n，因此空间复杂度也为 $O(n)$。其他辅助变量的空间复杂度为 $O(1)$。
- 因此，回溯法的总空间复杂度为$O(n)$。

4.3 结果展示

```
========15=========
最佳路径为: 20 17 5 22 11 19 8 10 21 12 13 3 16 7 9 20
最佳路径长度为: 5506.88
扫描过的搜索树结点总数为: 254515
程序运行时间为: 0.01s

=========20=========
最佳路径为: 20 17 5 22 11 19 18 8 1 10 21 14 12 15 2 13 3 16 7 9 20
最佳路径长度为: 6987.51
扫描过的搜索树结点总数为: 76201708
程序运行时间为: 3.44s

=========22=========
```

```
最佳路径为: 20 9 7 16 3 13 2 15 12 14 21 10 1 4 6 18 8 19 11 22 5 17 20
最佳路径长度为: 7690.80
扫描过的搜索树结点总数为: 486666892
程序运行时间为: 22.72s


=========30=========
最佳路径为: 20 12 22 8 16 18 11 24 29 4 17 10 14 2 1 13 30 7 6 5 3 9 19 23 28
↪  21 27 26 25 15 20
最佳路径长度为: 11426.60
扫描过的搜索树结点总数为: 3894050788
程序运行时间为: 151.56s
```

5 分支限界法

5.1 算法设计

5.1.1 算法逻辑

见 3

5.1.2 核心代码

贪心法求上界

```
1  double dfs(int u, int cnt, double l) // u:当前节点, cnt:已访问节点数, l:当前路径长
       度
2  {
3      // 贪心算法求上界
4      visited[u] = true;
5      if (cnt == n)
6      {
7          if (w[u][start] == INF) // 当前点和起始点之间没有边
8          {
9              visited[u] = false; // 回溯
10             return INF;
11         }
12         return l + w[u][start];
13     }
14     double min, ret = INF;
15     bool prev[N] = {false};
16     int next;
17     while (ret >= INF)
18     {
19         min = INF;
```

```
        next = 0;
        for (int i = 1; i <= n; i++)
        {
            if (!visited[i] && w[u][i] < min && !prev[i])
            {
                min = w[u][i];
                next = i;
            }
        }
        if (!next) // 当前点和所有未访问点之间没有边
        {
            visited[u] = false; // 回溯
            return INF;
        }
        prev[next] = true;
        ret = dfs(next, cnt + 1, l + min); // ret = INF时回溯
    }

    return ret;
}

void GetUp()
{
    up = dfs(start, 1, 0);
}
```

使用贪心法求上界时需要对于当前点和所有未访问的点之间没有边的情况进行回溯处理，回溯后，选择次短边再次贪心。

计算下界

```
double GetDown(const Node &node)
{
    double ret = 2 * node.l; // 已经走过的路径长度的两倍

    if (node.path.size() >= 2)
    {
        double min = INF;
        for (int i = 1; i <= n; i++) // 从起始点到未访问点的最小边
        {
            if (!node.visited[i] && w[start][i] < min)
                min = w[start][i];
        }
        ret += min;
        min = INF;
        for (int i = 1; i <= n; i++) // 从当前路径最后一个点到未访问点的最小边
        {
            if (!node.visited[i] && w[node.path.back()][i] < min)
```

```
                min = w[node.path.back()][i];
        }
        ret += min;
    }

    // 进入/离开未遍历城市时，各未遍历城市带来的最小路径成本
    for (int i = 1; i <= n; i++)
        if (!node.visited[i])
            ret += w_min[i][0] + w_min[i][1];

    return ret / 2;
}
```

计算 lb 时由于需要多次用到未访问的点的最短边和次短边，每次都重新计算算法时间复杂度太高，因此采用在初始化时将所有点的最短边和次短边计算出来并保存在数组 `w_min` 中，以空间换时间。

分支限界算法

```
void BranBoundTSP()
{
    // 初始化起始节点
    Node node;
    node.l = 0;
    node.path.push_back(start);
    for (int i = 1; i <= n; i++)
        node.visited[i] = false;
    down = GetDown(node);
    node.lb = down;
    node.visited[start] = true;
    GetUp();
    cout << "下界: " << down << endl;
    cout << "上界: " << up << endl;

    if (down >= up)
        return; // 使用贪心法可以直接得到最优解

    // 将起始点加入优先队列
    priority_queue<Node> q;
    q.push(node);

    while (!q.empty())
    {
        Node t = q.top();
        q.pop();
        if (t.path.size() == n - 1) // 已经找到 n - 1 个点
        {
```

```
            L++;
            double lb_min; // 记录当前lb的最小值
            for (int i = 1; i <= n; i++)
            {
                if (!t.visited[i]) // 找到最后一个点
                {
                    t.l += w[t.path.back()][i];
                    lb_min = t.lb;
                    t.lb = t.l + w[i][start];
                    t.path.push_back(i);
                    break;
                }
            }
            if (t.lb < up)
            {
                up = t.lb; // 更新上界

                // 删除优先队列中所有lb大于当前上界的节点
                auto tp = q;
                q = {}; // 清空队列
                while (!tp.empty() && tp.top().lb <= up)
                {
                    q.push(tp.top());
                    tp.pop();
                }

                for (int i = 1; i <= n; i++) // 更新最佳路径
                    bestx[i] = t.path[i - 1];
                bestw = t.lb;        // 更新最佳路径长度
                if (t.lb <= lb_min) // 如果最佳路径长度小于等于当前lb的最小值，说明已经找到最优解
                    return;
            }
        }
        else
        {
            for (int i = 1; i <= n; i++)
            {
                if (!t.visited[i] && w[t.path.back()][i] != INF) // 未访问过且有边
                {
                    L++;
                    Node next = t;
                    next.l += w[t.path.back()][i];
                    next.path.push_back(i);
                    next.visited[i] = true;
                    next.lb = GetDown(next);
                    if (next.lb <= up) // 目标值的下界小于等于上界，才将其加入队列
```

```
                            q.push(next);
                        }
                    }
                }
            }
        }
```

分支限界算法的核心在于扩展节点的方式以及节点入队的条件（对于死节点进行剪枝），详细策略见 3。

5.2 算法性能分析

时间复杂度

- 贪心法求上界：递归深度为 $O(n)$，每层的计算复杂度为 $O(n)$，因此整体时间复杂度为 $O(n^2)$。
- 计算下界（lb）：因为采用了空间换时间的策略，将两层循环降低为了一层循环，因此整体时间复杂度为 $O(n)$。
- 分支限界：使用优先队列来存储待扩展的节点，并在每次循环中从队列中取出一个节点进行扩展，直到队列为空。在每次循环中，需要遍历所有待加入的节点并计算下界，然后将满足条件的节点加入队列。遍历节点总数为 $O(2^n)$，计算每个节点的 lb 的时间复杂度为 $O(n)$，因此，整个分支限界算法的时间复杂度为$O(n*2^n)$，其中 n 是基站的数量。与教材中的复杂度 $O(n^2*2^n)$ 不同的原因是：对计算下界的函数进行了优化，让时间复杂度从 $O(n^2)$ 变成了 $O(n)$

空间复杂度 分支限界的空间复杂度主要取决于优先队列中的节点总数，最坏情况下，所有叶子节点均在队列中，叶子节点总数为 $O(n!)$，因此空间复杂度为$O(n!)$。

5.3 结果展示

```
下界：4875.16
上界：5904.76
=========15=========
最佳路径为：20 17 5 22 11 19 8 10 21 12 13 3 16 7 9 20
最佳路径长度为：5506.88
扫描过的搜索树结点总数为：5367
程序运行时间为：0.03s

下界：6393.92
上界：7385.39
=========20=========
最佳路径为：20 17 5 22 11 19 18 8 1 10 21 14 12 15 2 13 3 16 7 9 20
```

```
最佳路径长度为: 6987.51
扫描过的搜索树结点总数为: 34639
程序运行时间为: 0.53s

下界: 7214.99
上界: 8433.1
=========22=========
最佳路径为: 20 17 5 22 11 19 8 18 6 4 1 10 21 14 12 15 2 13 3 16 7 9 20
最佳路径长度为: 7690.80
扫描过的搜索树结点总数为: 15277
程序运行时间为: 0.10s

下界: 10428.9
上界: 12105.4
=========30=========
最佳路径为: 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
最佳路径长度为: 11426.60
扫描过的搜索树结点总数为: 15162831
程序运行时间为: 317.81s
```

6 结果对比

问题	求解算法	最短回路	路径总长度 (m)	搜索过的结点总数	程序运行时间 (s)
15 个基站	回溯	20 17 5 ... 16 7 9 20	5506.88	254515	0.01
	分支限界	20 17 5 ... 16 7 9 20	5506.88	5367	0.03s
20 个基站	回溯	20 17 5 ... 16 7 9 20	6987.51	76201708	3.44s
	分支限界	20 17 5 ... 16 7 9 20	6987.51	34639	0.53s
22 个基站	回溯	20 9 7 ... 22 5 17 20	7690.80	486666892	22.72s
	分支限界	20 17 5 ... 16 7 9 20	7690.80	15277	0.10s
30 个基站	回溯	20 12 22 ... 26 25 15 20	11426.60	3894050788	151.56s
	分支限界	20 15 25 ... 8 22 12 20	11426.60	15162831	317.81s

表 1: 回溯法和分支限界法结果对比

可以发现回溯法和分支限界法求出的最短路径长度是相同的，但路径存在一个是另一个逆序的情况，如 22 个基站和 30 个基站的结果。同时可以发现分支限界法搜索的节点数是远远小于回溯法的，这得益于分支限界的 lb 的计算和根据 lb 的剪枝策略，但同时计算 lb 带来了计算

开销，因此分支限界法的时间不一定优于回溯法。

7 总结

在本次实验中，我分别实现了回溯法和分支限界法以解决 TSP 问题。

回溯法是一种穷举搜索算法，用于解决组合优化问题。在 TSP 问题中，回溯法通过尝试每个可能的路径来找到最优解。它从起始城市开始，逐步选择下一个未访问的城市，并更新当前路径长度，直到所有城市都被访问一次。如果找到一条更短的路径，就更新最优路径和最短路径长度。然后，回溯到上一步的选择，继续尝试其他可能的路径，直到所有可能性都被遍历。

分支限界法是一种优化搜索算法，用于解决组合优化问题。在 TSP 问题中，分支限界法通过利用启发式信息和剪枝策略，减少搜索空间并快速找到最优解。它首先构建一棵搜索树，树的每个节点表示当前访问的城市和已经访问的路径。通过计算当前路径的 lb 与最优解上界 up 进行比较，可以提前剪枝掉一些不可能达到最优解的分支。然后，根据启发式信息选择下一个要访问的城市，并继续扩展搜索树，直到找到最优解或搜索完整棵树。

总的来说，回溯法的优点是简单易实现，适用于小规模问题，但对于大规模问题效率较低。分支限界法通过剪枝策略和启发式信息，能够有效地减少搜索空间，适用于中等规模的问题。

A BacktrackTSP.cpp

```cpp
/**
 * @file BacktrackTSP.cpp
 * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
 * @brief 回溯法求解TSP问题
 * @date 2023-12-21
 */
#include <iostream>
#include <fstream>
#include <sstream>
#include <time.h>
#include <iomanip>

using namespace std;

const int INF = 0x3f3f3f3f;
const int N = 100; // 最大基站数

int bestx[N];               // 记录最佳路径
double bestw = INF;         // 记录最佳路径长度
double cw = 0;              // 当前部分路径长度
int x[N];                   // 当前部分路径
double w[N][N];             // 邻接矩阵存储基站间距离
int n;                      // 基站数
int f[N];                   // 记录index和基站编号的对应关系
int start;                  // 记录起始基站index
long long L = 0;                  // 扫描过的搜索树结点总数
clock_t startTime, endTime; // 记录程序运行时间

void BacktrackTSP(int i)
{
    if (i == n) // 搜索到叶节点
    {
        if (w[x[n - 1]][x[n]] != INF && w[x[n]][start] != INF && cw + w[x[n - 1]][x
            [n]] + w[x[n]][start] < bestw)
        {
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
            bestw = cw + w[x[n - 1]][x[n]] + w[x[n]][start];
        }
    }
    else // 搜索非叶节点
    {
        for (int j = i; j <= n; j++)
        {
            if (w[x[i - 1]][x[j]] != INF && cw + w[x[i - 1]][x[j]] < bestw)
```

```
            {
                L++;
                swap(x[i], x[j]);          // 交换，将x[j]加入到当前解中
                cw += w[x[i - 1]][x[i]]; // 更新当前路径长度
                BacktrackTSP(i + 1);       // 递归搜索
                cw -= w[x[i - 1]][x[i]]; // 回溯
                swap(x[i], x[j]);
            }
        }
    }
}

bool ReadData()
{
    string filename = "DATA/tsp_" + to_string(n) + ".txt";
    ifstream file(filename, ios::in);
    if (!file.is_open())
    {
        cerr << "Failed to open file." << endl;
        return false;
    }
    string line;
    int i = 1, j = 1;
    while (getline(file, line))
    {
        stringstream ss(line);
        int a, b;
        double c;
        string _a, _b, _c;
        ss >> _a >> _b >> _c;
        a = stoi(_a), b = stoi(_b), c = stod(_c);
        f[i] = a, f[j] = b; // 记录index和基站编号的对应关系
        if (a == 20)
            start = i;  // 记录起始基站index
        if (c == 99999) // a, b不相邻
            c = INF;
        w[i][j] = c;
        j++;
        if (j > n) // 一行读完
            i++, j = 1;
    }
    return true;
}

void PrintResult(int num)
{
    cout << "=========" << num << "=========" << endl;
```

```
92      cout << "最佳路径为: ";
93      for (int i = 1; i <= n; i++)
94          cout << f[bestx[i]] << " ";
95      cout << f[bestx[1]] << endl;
96      cout << "最佳路径长度为: " << fixed << setprecision(2) << bestw << endl;
97      cout << "扫描过的搜索树结点总数为: " << L << endl;
98      cout << "程序运行时间为: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
            "s" << endl;
99  }
100
101 int main()
102 {
103     n = 30;
104     if (!ReadData())
105         return 1;
106     for (int i = 1; i <= n; i++) // 初始化
107         x[i] = i;
108     swap(x[1], x[start]); // 将起始基站放在第一个位置
109     startTime = clock();
110     BacktrackTSP(2);
111     endTime = clock();
112     PrintResult(n);
113 }
```

B BranBoundTSP.cpp

```cpp
/**
 * @file BranBoundTSP.cpp
 * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
 * @brief 分支限界法求解TSP问题
 * @date 2023-12-21
 */
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <queue>
#include <iomanip>
#include <time.h>
#include <math.h>

using namespace std;

const int N = 100;           // 最大基站数
const int INF = 0x3f3f3f3f; // 无穷大

int bestx[N];                // 记录最佳路径
double bestw = INF;          // 记录最佳路径长度
double down, up;             // 下界和上界
bool visited[N];             // 访问标记
double w[N][N];              // 距离矩阵
double w_min[N][2];          // 记录每个点到其他点的最小边和次小边
int n;                       // 基站数
int start;                   // 起始基站
int f[N];                    // 记录index和基站编号的对应关系
long long L = 0;             // 扫描过的搜索树结点总数
clock_t startTime, endTime; // 记录程序运行时间

struct Node
{
    bool visited[N];  // 访问标记
    vector<int> path; // 访问路径
    double l;         // 当前路径长度
    double lb;        // 目标值的下界
    bool operator<(const Node &a) const
    {
        return lb > a.lb;
    }
};

```

```
45 double dfs(int u, int cnt, double l) // u:当前节点, cnt:已访问节点数, l:当前路径长
       度
46 {
47     // 贪心算法求上界
48     visited[u] = true;
49     if (cnt == n)
50     {
51         if (w[u][start] == INF) // 当前点和起始点之间没有边
52         {
53             visited[u] = false; // 回溯
54             return INF;
55         }
56         return l + w[u][start];
57     }
58     double min, ret = INF;
59     bool prev[N] = {false};
60     int next;
61     while (ret >= INF)
62     {
63         min = INF;
64         next = 0;
65         for (int i = 1; i <= n; i++)
66         {
67             if (!visited[i] && w[u][i] < min && !prev[i])
68             {
69                 min = w[u][i];
70                 next = i;
71             }
72         }
73         if (!next) // 当前点和所有未访问点之间没有边
74         {
75             visited[u] = false; // 回溯
76             return INF;
77         }
78         prev[next] = true;
79         ret = dfs(next, cnt + 1, l + min); // ret = INF时回溯
80     }
81
82     return ret;
83 }
84
85 void GetUp()
86 {
87     up = dfs(start, 1, 0);
88 }
89
90 double GetDown(const Node &node)
```

```
{
    double ret = 2 * node.l; // 已经走过的路径长度的两倍

    if (node.path.size() >= 2)
    {
        double min = INF;
        for (int i = 1; i <= n; i++) // 从起始点到未访问点的最小边
        {
            if (!node.visited[i] && w[start][i] < min)
                min = w[start][i];
        }
        ret += min;
        min = INF;
        for (int i = 1; i <= n; i++) // 从当前路径最后一个点到未访问点的最小边
        {
            if (!node.visited[i] && w[node.path.back()][i] < min)
                min = w[node.path.back()][i];
        }
        ret += min;
    }

    // 进入/离开未遍历城市时，各未遍历城市带来的最小路径成本
    for (int i = 1; i <= n; i++)
        if (!node.visited[i])
            ret += w_min[i][0] + w_min[i][1];

    return ret / 2;
}

void BranBoundTSP()
{
    // 初始化起始节点
    Node node;
    node.l = 0;
    node.path.push_back(start);
    for (int i = 1; i <= n; i++)
        node.visited[i] = false;
    down = GetDown(node);
    node.lb = down;
    node.visited[start] = true;
    GetUp();
    cout << "下界: " << down << endl;
    cout << "上界: " << up << endl;

    if (down >= up)
        return; // 使用贪心法可以直接得到最优解

```

```
    // 将起始点加入优先队列
    priority_queue<Node> q;
    q.push(node);

    while (!q.empty())
    {
        Node t = q.top();
        q.pop();
        if (t.path.size() == n - 1) // 已经找到 n - 1 个点
        {
            L++;
            double lb_min; // 记录当前lb的最小值
            for (int i = 1; i <= n; i++)
            {
                if (!t.visited[i]) // 找到最后一个点
                {
                    t.l += w[t.path.back()][i];
                    lb_min = t.lb;
                    t.lb = t.l + w[i][start];
                    t.path.push_back(i);
                    break;
                }
            }
            if (t.lb < up)
            {
                up = t.lb; // 更新上界

                // 删除优先队列中所有lb大于当前上界的节点
                auto tp = q;
                q = {}; // 清空队列
                while (!tp.empty() && tp.top().lb <= up)
                {
                    q.push(tp.top());
                    tp.pop();
                }

                for (int i = 1; i <= n; i++) // 更新最佳路径
                    bestx[i] = t.path[i - 1];
                bestw = t.lb;        // 更新最佳路径长度
                if (t.lb <= lb_min) // 如果最佳路径长度小于等于当前lb的最小值，说明已经找到最优解
                    return;
            }
        }
        else
        {
            for (int i = 1; i <= n; i++)
```

```
            {
                if (!t.visited[i] && w[t.path.back()][i] != INF) // 未访问过且有边
                {
                    L++;
                    Node next = t;
                    next.l += w[t.path.back()][i];
                    next.path.push_back(i);
                    next.visited[i] = true;
                    next.lb = GetDown(next);
                    if (next.lb <= up) // 目标值的下界小于等于上界，才将其加入队列
                        q.push(next);
                }
            }
        }
    }
}

bool ReadData()
{
    string filename = "DATA/tsp_" + to_string(n) + ".txt";
    ifstream file(filename, ios::in);
    if (!file.is_open())
    {
        cerr << "Failed to open file." << endl;
        return false;
    }
    string line;
    int i = 1, j = 1;
    while (getline(file, line))
    {
        stringstream ss(line);
        int a, b;
        double c;
        string _a, _b, _c;
        ss >> _a >> _b >> _c;
        a = stoi(_a), b = stoi(_b), c = stod(_c);
        f[i] = a, f[j] = b; // 记录index和基站编号的对应关系
        if (a == 20)
            start = i;  // 记录起始基站index
        if (c == 99999) // a, b不相邻
            c = INF;
        w[i][j] = c;
        j++;
        if (j > n) // 一行读完
            i++, j = 1;
    }

```

```
231     // 计算每个点到其他点的最小边和次小边
232     for (int i = 1; i <= n; i++)
233     {
234         double min1 = INF, min2 = INF;
235         for (int j = 1; j <= n; j++)
236         {
237             if (w[i][j] < min1)
238             {
239                 min2 = min1;
240                 min1 = w[i][j];
241             }
242             else if (w[i][j] < min2)
243             {
244                 min2 = w[i][j];
245             }
246         }
247         w_min[i][0] = min1;
248         w_min[i][1] = min2;
249     }
250 
251     return true;
252 }
253 
254 void PrintResult(int num)
255 {
256     cout << "==========" << num << "==========" << endl;
257     cout << "最佳路径为: ";
258     for (int i = 1; i <= n; i++)
259         cout << f[bestx[i]] << " ";
260     cout << f[bestx[1]] << endl;
261     cout << "最佳路径长度为: " << fixed << setprecision(2) << bestw << endl;
262     cout << "扫描过的搜索树结点总数为: " << L << endl;
263     cout << "程序运行时间为: " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
          "s" << endl;
264 }
265 
266 int main()
267 {
268     n = 30;
269     if (!ReadData())
270         return 1;
271     startTime = clock();
272     BranBoundTSP();
273     endTime = clock();
274     PrintResult(n);
275 }
```