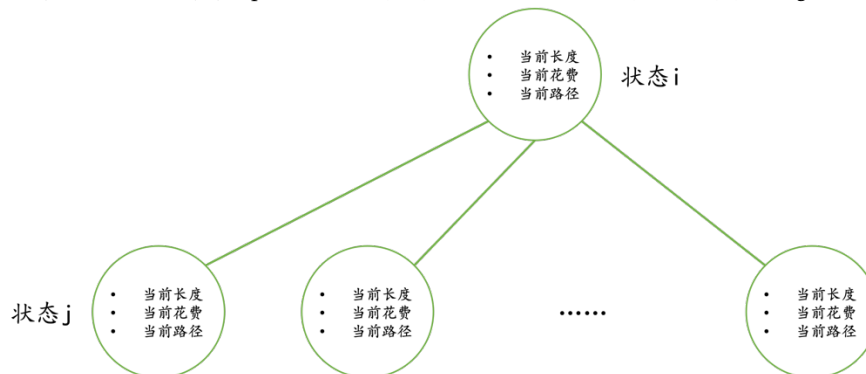


1. 问题重述

利用分支定界算法求解从甲地 (Num. 1) 到乙地 (Num. 50) 的最短路径，且需要满足花费小于 1500。一共有 50 座城市，城市之间相连的有向路径长度由 M1.txt 给出，9999 代表不连通；城市之间有向路径的花费由 M2.txt 给出。

2. 算法设计

构建状态树，树上每个节点记录状态，从出发点到当前节点的距离 length、花费 cost、路径 path[]；从节点 i，经过可行边到节点 j。



搜索操作

若某个节点没有被剪枝掉，即存在最优解的可能，进行下一步搜索。对于当前节点 i，分别去搜索每一个从该点出发的边，若边的长度不是 9999，且边的另一边的点并没有被访问过，且扩展点后的状态满足要求，则拓展该点信息，根据当前状态计算该点状态，继续搜索。

剪枝操作

设从节点 i 到终点的最少花费为 $\text{minCost}[i]$ ，若当前花费 $\text{curCost} + \text{minCost}[i] > 1500$ ，则进行剪枝；同理，若从节点 i 到终点的最短长度为 $\text{minLen}[i]$ ，若当前长度 $\text{curLen} + \text{minLen}[i] > \text{bestLen}$ ，则进行剪枝。

当扩展边时，若目标点已经被扩展过、当前花费 $\text{curCost} + \text{edge.cost} > 1500$ 、 $\text{curLen} + \text{edge.len} > \text{bestLen}$ ，则进行剪枝。

更新答案

当搜索到终点时，若当前解满足条件且更优，则更新答案（包括花费、长度、路径）。

3. 算法实现和结果

伪代码

```
dfs(int curPoint)
// dfs 搜索，分支定界算法
// 输入：当前节点 curPoint
// 输出：无，但是会更新全局变量最优解

// 到达终点
if curPoint = 终点
    if curCost 满足要求 && curLen 满足要求
```

```

        更新最优解
    end
    return;
end

// 剪枝
// (其中 curPoint 到终点的最小花费和最小长度通过 dijkstra 算法预处理得到)
if 当前花费 + curPoint 到终点最小花费 > 1500 || 当前长度 + curPoint 到终点最小长度 > 最优长度
    return;
end

// 搜索
for 边 e in curPoint 的所有有效边
    if e.end 未被访问 && 当前花费 + e.cost <= 1500 && 当前长度 + e.len <= 最优长度
        e.end 被访问;
        当前花费 += e.cost;
        当前长度 += e.len;
        当前路径 添加 e.end;
        dfs(e.end);
        e.end 未被访问;
        当前路径 删除 e.end;
        当前花费 -= e.cost;
        当前长度 -= e.len;
    end
end
end

```

预处理

为了获得各个点到终点的最短长度和最小花费，且只需要获得该数据，为了达到更好的时间复杂度，因此使用**优先队列优化的 dijkstra 算法**，利用反向图，得到终点到任意点的最短长度和最小花费，从而得到各个点到终点的最短长度和最小花费。

该预处理算法的时间复杂度为 $O(E \log V)$ 。

代码运行环境和结果

运行环境：

- 操作系统：macOS Ventura13.2.1
- 构建工具：cmake version 3.25.2; GNU Make 3.81
- 构建命令：cmake CMakeLists.txt && make
- 运行命令：./algorithm_assignment_1

运行结果：

bestCost: 1448

bestLen: 464

bestPath: 1->3->8->11->15->21->23->26->32->37->39->45->47->50

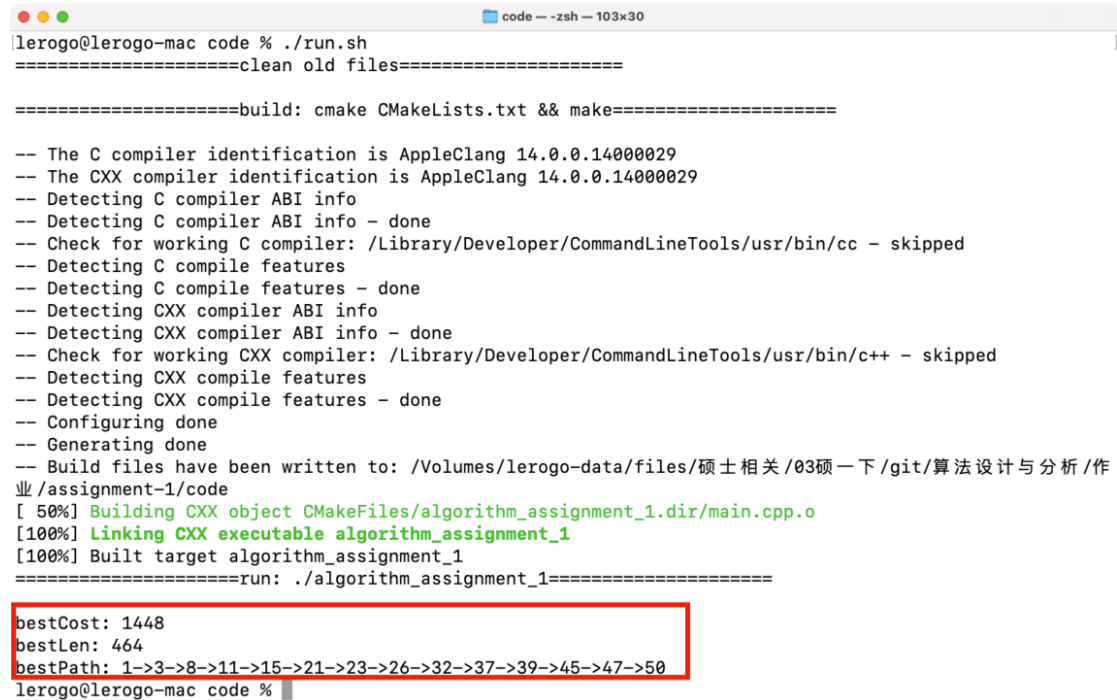
重复两千次的情况下：（更多请参考代码文件：main.cpp）

所用时间（macos、1.4 GHz 四核 Intel Core i5、release 版本）

time: 2309.51ms

time: 2.30951e+06clocks

结果截图：



```
lerogo@lerogo-mac code % ./run.sh
=====clean old files=====

=====build: cmake CMakeLists.txt && make=====

-- The C compiler identification is AppleClang 14.0.0.14000029
-- The CXX compiler identification is AppleClang 14.0.0.14000029
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Volumes/lerogo-data/files/硕士相关/03硕一下/git/算法设计与分析/作业/assignment-1/code
[ 50%] Building CXX object CMakeFiles/algorithm_assignment_1.dir/main.cpp.o
[100%] Linking CXX executable algorithm_assignment_1
[100%] Built target algorithm_assignment_1
=====run: ./algorithm_assignment_1=====
bestCost: 1448
bestLen: 464
bestPath: 1->3->8->11->15->21->23->26->32->37->39->45->47->50
lerogo@lerogo-mac code %
```

4. 附录-代码

```
/*
 * encoding: utf-8
 */
#include <iostream>
#include <fstream>
#include <utility>
#include <vector>
#include <queue>
#include <cstring>

// 城市的数量、最大的花费、最大的距离
const int cityNum = 50, maxCost = 1500, maxLen = 9999;

// 定义边集，在其中搜索；其中 reverseEdge 是反向边集，用于 dijkstra 算法
struct Edge {
    int start;
    int end;
    int cost;
```

```

    int len;
} orderEdge[cityNum][cityNum], reverseEdge[cityNum][cityNum];

// 从点 i 有多少条边
int orderEdgeNum[cityNum], reverseEdgeNum[cityNum];

// 初始化, 得到点 i 到终点的最短距离和最小花费
int minCost[cityNum], minLen[cityNum];

// 是否已经搜索过这个点
int visitPoint[cityNum];

// 记录当前花费、距离和路径
int curCost, curLen, curPath[cityNum + 1];

// 记录最优花费、距离和路径
int bestCost, bestLen, bestPath[cityNum + 1];

// 读取数据
void readData() {
    // 请注意, 这里的路径是相对于可执行文件的路径
    std::ifstream fin1("./m1.txt");
    std::ifstream fin2("./m2.txt");
    int tmpLen, tmpCost;
    for (int i = 0; i < cityNum; i++) {
        for (int j = 0; j < cityNum; j++) {
            fin1 >> tmpLen;
            if (tmpLen < maxLen) {
                orderEdge[i][orderEdgeNum[i]].start = i;
                orderEdge[i][orderEdgeNum[i]].end = j;
                orderEdge[i][orderEdgeNum[i]].len = tmpLen;
                fin2 >> orderEdge[i][orderEdgeNum[i]].cost;
                reverseEdge[j][reverseEdgeNum[j]].start = j;
                reverseEdge[j][reverseEdgeNum[j]].end = i;
                reverseEdge[j][reverseEdgeNum[j]].len = tmpLen;
                reverseEdge[j][reverseEdgeNum[j]].cost =
orderEdge[i][orderEdgeNum[i]].cost;
                orderEdgeNum[i]++;
                reverseEdgeNum[j]++;
            } else {
                fin2 >> tmpCost;
            }
        }
    }
}

```

```

    fin1.close();
    fin2.close();
}

// 初始化, 获得每个点到终点的最短距离和最少花费
void dijkstra_q(bool isLen = true) {
    typedef std::pair<int, int> PII;
    std::priority_queue<PII, std::vector<PII>, std::greater<>> q;
    q.emplace(0, cityNum - 1);
    minLen[cityNum - 1] = 0;
    minCost[cityNum - 1] = 0;
    if (isLen) {
        while (!q.empty()) {
            PII p = q.top();
            q.pop();
            int v = p.second;

            if (minLen[v] < p.first) {
                continue;
            }
            // 更新
            for (int i = 0; i < reverseEdgeNum[v]; i++) {
                Edge e = reverseEdge[v][i];
                if (minLen[e.end] > minLen[e.start] + e.len) {
                    minLen[e.end] = minLen[e.start] + e.len;
                    q.emplace(minLen[e.end], e.end);
                }
            }
        }
    } else {
        while (!q.empty()) {
            PII p = q.top();
            q.pop();
            int v = p.second;
            if (minCost[v] < p.first) {
                continue;
            }
            for (int i = 0; i < reverseEdgeNum[v]; i++) {
                Edge e = reverseEdge[v][i];
                if (minCost[e.end] > minCost[e.start] + e.cost) {
                    minCost[e.end] = minCost[e.start] + e.cost;
                    q.emplace(minCost[e.end], e.end);
                }
            }
        }
    }
}

```

```
    }  
  }  
}
```

```
// 初始化
```

```
void init() {  
    memset(orderEdgeNum, 0, sizeof(orderEdgeNum));  
    memset(reverseEdgeNum, 0, sizeof(reverseEdgeNum));  
    memset(minCost, 0x7f, sizeof(minCost));  
    memset(minLen, 0x7f, sizeof(minLen));  
    memset(visitPoint, 0, sizeof(visitPoint));  
    memset(curPath, 0, sizeof(curPath));  
    curPath[0] = 1;  
    memset(bestPath, 0, sizeof(bestPath));  
    bestPath[0] = 1;  
    curCost = 0;  
    curLen = 0;  
    bestCost = maxCost;  
    bestLen = maxLen;  
  
    readData();  
    dijkstra_q(true);  
    dijkstra_q(false);  
}
```

```
// dfs 搜索
```

```
void dfs(int curPoint) {  
    // 到达终点  
    if (curPoint == cityNum - 1) {  
        if ((curLen < bestLen && curCost <= maxCost) || (curLen == bestLen &&  
curCost < bestCost)) {  
            bestCost = curCost;  
            bestLen = curLen;  
            memcpy(bestPath, curPath, sizeof(curPath));  
        }  
        return;  
    }  
    // 剪枝  
    if (curCost + minCost[curPoint] > maxCost || curLen + minLen[curPoint] >  
bestLen) {  
        return;  
    }  
    // 搜索  
    for (int i = 0; i < orderEdgeNum[curPoint]; i++) {
```

```

        Edge e = orderEdge[curPoint][i];
        if (visitPoint[e.end] == 0 && curCost + e.cost <= maxCost && curLen +
e.len <= bestLen) {
            visitPoint[e.end] = 1;
            curCost += e.cost;
            curLen += e.len;
            curPath[++curPath[0]] = e.end;
            dfs(e.end);
            visitPoint[e.end] = 0;
            curCost -= e.cost;
            curLen -= e.len;
            curPath[0]--;
        }
    }
}

// 打印结果
void printResult() {
    std::cout << "bestCost: " << bestCost << std::endl << "bestLen: " <<
bestLen << std::endl;
    std::cout << "bestPath: ";
    for (int i = 1; i <= bestPath[0]; i++) {
        std::cout << bestPath[i] + 1;
        if (i != bestPath[0]) {
            std::cout << "->";
        }
    }
    std::cout << std::endl;
}

int main() {
    /*
        int num = 2000;
        clock_t startTime = clock();
        for (int i = 0; i < num; ++i) {
            init();
            dfs(0);
        }
        clock_t endTime = clock();
        std::cout << "time: " << (double) (endTime - startTime) / CLOCKS_PER_SEC
* 1000 << "ms" << std::endl;
        std::cout << "time: " << (double) (endTime - startTime) << "clocks" <<
std::endl;
    */
}

```

```
/* 重复 2000 次，所用时间 (macos、1.4 GHz 四核 Intel Core i5、release 版本)
 * time: 2309.51ms
 * time: 2.30951e+06clocks
 * bestCost: 1448
 * bestLen: 464
 * bestPath: 1->3->8->11->15->21->23->26->32->37->39->45->47->50
 */

init();
dfs(0);
printResult();

return 0;
}
```