1 问题描述

- 用分支定界算法求以下问题:
  某公司于乙城市的销售点急需一批成品，该公司成品生产基地在甲城市。
  甲城市与乙城市之间共有 n 座城市，互相以公路连通。
  甲城市、乙城市以及其它各城市之间的公路连通情况及每段公路的长度由矩阵 M1 给出。
  每段公路均由地方政府收取不同额度的养路费等费用，具体数额由矩阵 M2 给出。
  请给出在需付养路费总额不超过 1500 的情况下，该公司货车运送其产品从甲城市到乙城市的最短运送路线。
  具体数据参见文件:
  M1.txt: 各城市之间的公路连通情况及每段公路的长度矩阵 (有向图); 甲城市为城市 Num.1，乙城市为城市 Num.50。
  M2.txt: 每段公路收取的费用矩阵 (非对称)。

2 算法思路

2.1 距离下界和费用下界

为了在搜索时进行尽可能的剪枝，可以首先根据 $dijkstra$ 算法计算出没有费用限制的最短路径和最小费用，然后在遍历搜索树时利用当前节点对应路径的长度 $+dijkstra$ 算法得到的后继序列的最短长度作为距离下界，利用当前当前节点对应路径的费用 $+dijkstra$ 算法得到的后继序列的最小费用作为费用下界。

1. 距离下界 $= length\ of\ current\_sequence + minimal\ length\ from\ current\_sequence[-1]\ to\ B$
2. 费用下界 $= cost\ of\ current\_sequence + minimal\ cost\ from\ current\_sequence[-1]\ to\ B$

2.2 分支定界法

在构建搜索树时可以有 2 种选择：二叉搜索树和多叉搜索树。下面分别介绍基于两者的分支定界求解过程。

2.2.1 二叉搜索树

Algorithm 1 基于二叉搜索树的分支定界法求最短路径

INPUT: 城市个数 n, 所有城市之间的距离矩阵 $distance\_matrix$ 和费用矩阵 $cost\_matrix$

OUTPUT: 最短路径 $solution$

function SEARCH($sequence, remove, solution$)
    $last\_node = sequence[-1]$
    剪枝条件 1 ← sequence 中刚刚添加了新元素 (或 remove 为空)
    剪枝条件 2 ← $last\_node$ 可达 B 且 $sequence$ 长度 +$shortest\_path[last\_node]$ 长度 ≥ 当前解 $solution$ 的长度
    剪枝条件 3 ← sequence 费用 +$min\_cost[last\_node]$ 费用 >1500
    if 满足剪枝条件 $1 \wedge (2 \vee 3)$ **then**
        剪枝回溯
    else
        if $sequence$+$shortest\_path[last\_node]$ 是可行解且优于当前最优解 solution **then**
            更新 solution
        end if
    end if
    找到一个不在 sequence 和 remove 且与 sequence 最后一个城市连通的城市 c
    $SEARCH(sequence + [c], [], solution)$
    $SEARCH(sequence, remove + [c], solution)$
end function

Initialize:
    $sequence \leftarrow [A], remove \leftarrow [], solution \leftarrow []$
for $i = 0 \rightarrow n - 1$ **do**
    $shortest\_path[i] \leftarrow dijkstra$ 算法计算从 i 到 B 的最短路径
    $min\_cost[i] \leftarrow dijkstra$ 算法计算从 i 到 B 的最小费用
end for
$SEARCH(sequence, remove, solution)$
return $solution$

2.2.2 多叉搜索树

Algorithm 2 基于多叉搜索树的分支定界法求最短路径

INPUT: 城市个数 n, 所有城市之间的距离矩阵 $distance\_matrix$ 和费用矩阵 $cost\_matrix$

OUTPUT: 最短路径 $solution$

function SEARCH($sequence, solution$)
    $last\_node = sequence[-1]$
    剪枝条件 1 ← $last\_node$ 可达 B 且 $sequence$ 长度 +$shortest\_path[last\_node]$ 长度 ≥ 当前解 $solution$ 的长度
    剪枝条件 2 ← sequence 费用 +$min\_cost[last\_node]$ 费用 >1500
    if 满足剪枝条件 $1 \vee 2$ **then**
        剪枝回溯
    else
        if $sequence$+$shortest\_path[last\_node]$ 是可行解且优于当前最优解 solution **then**
            更新 solution
        end if
    end if
    $cities$ ← 所有不在 $sequence$ 且与 $sequence$ 最后一个城市连通的城市
    for $all\ c \in cities$ **do**
        $SEARCH(sequence + [c], solution)$
    end for
end function

Initialize:
    $sequence \leftarrow [A], solution \leftarrow []$
for $i = 0 \rightarrow n - 1$ **do**
    $shortest\_path[i]$ ← $dijkstra$ 算法计算从 i 到 B 的最短路径
    $min\_cost[i]$ ← $dijkstra$ 算法计算从 i 到 B 的最小费用
end for
$SEARCH(sequence, solution)$
return $solution$

3 代码实现和结果分析

3.1 代码

3.1.1 时间统计 timer.h

```
#include <sys/time.h>

#define TI(tag)\
```

```
    struct timeval time_start_##tag, time_end_##tag;\
    do {\
        gettimeofday(&time_start_##tag, NULL);\
    } while (0)

#define TO(tag, func)\
    do {\
        gettimeofday(&time_end_##tag, NULL);\
        cout << func << " " << #tag":\t" << (time_end_##tag.tv_sec - time_start_##tag.tv_sec) * 1000.0 +
            (time_end_##tag.tv_usec - time_start_##tag.tv_usec) / 1000.0 << " ms" << endl;\
    } while (0)
```

3.1.2 二叉搜索树

```
/*
 * solve single source shortest path problem using branch and bound algorithm
 */

#include <iostream>
#include <fstream>
#include <string.h>
#include <vector>
#include <algorithm>
#include "timer.h"

#define MAX_NODE_NUM 50
#define INT_MAX_VALUE 9999
#define COST_UPPER_BOUND 1500

using std::ifstream;
using std::vector;
using std::find;
using std::cout;
using std::endl;
using std::reverse;

void parse_input(char *filename, int distance_matrix[][MAX_NODE_NUM])
{
    ifstream f(filename);
    for (int i = 0; i < MAX_NODE_NUM; i++) {
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            f >> distance_matrix[i][j];
        }
    }
}

void print_result(int current_shortest_dist, vector<int> current_shortest_path, int cost_matrix[][
    MAX_NODE_NUM])
{
    cout << "shortest_distance: " << current_shortest_dist << endl;
    cout << "shortest_path: ";
    for (int i = 0; i < current_shortest_path.size(); i++) {
        if (i == current_shortest_path.size() - 1)
            cout << current_shortest_path[i]+1;
        else
```

```
            cout << current_shortest_path[i]+1 << "->";
    }
    cout << endl;
    int sum = 0;
    for (int i = 1; i < current_shortest_path.size(); i++) {
        sum += cost_matrix[current_shortest_path[i-1]][current_shortest_path[i]];
    }
    cout << "cost: " << sum << endl;
}

void dijkstra(int distance_matrix[][MAX_NODE_NUM], int source,
              int shortest_distance[], int shortest_path[])
{
    // found用于标记已经得到最短距离的目标结点
    static int found[MAX_NODE_NUM];
    // 初始化为1
    memset(found, 0, sizeof(found));

    // 找出从源点到其相邻节点的最短路径
    found[source] = 1;
    for (int i = 0; i < MAX_NODE_NUM; i++) {
        shortest_distance[i] = distance_matrix[source][i];
        shortest_path[i] = source;
    }

    int min_distance;
    int min_distance_idx;

    // 找到从源点出发到其他各结点的最短路径
    for (int i = 1; i < MAX_NODE_NUM; i++) {
        min_distance = INT_MAX_VALUE;
        // 找到未被标记的路径最短的结点
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            if (found[j] == 0 && shortest_distance[j] < min_distance) {
                min_distance = shortest_distance[j];
                min_distance_idx = j;
            }
        }
        // 标记该结点为找到最短路径
        found[min_distance_idx] = 1;

        // 利用该该结点的最短路径更新其他结点的最短路径
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            int dist_idx_j = distance_matrix[min_distance_idx][j]; // 从该结点到j的路径
            int new_distance = min_distance + dist_idx_j; // 从source到该结点的最短路径+从该结点到j的路径
            if (found[j] == 0 && dist_idx_j != INT_MAX_VALUE && new_distance < shortest_distance[j]) {
                shortest_distance[j] = new_distance;
                shortest_path[j] = min_distance_idx;
            }
        }
    }

}

void search(vector<int> sequence, vector<int> remove,
            int sequence_distance, int sequence_cost,
            int &current_shortest_dist, vector<int> &current_shortest_path,
```

```
            int shortest_path[][MAX_NODE_NUM],
            int shortest_distance[][MAX_NODE_NUM], int min_cost[][MAX_NODE_NUM],
            int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM])
{
    int seq_len = sequence.size();
    int last_node = sequence[seq_len - 1];
    int opt_distance = sequence_distance + shortest_distance[last_node][MAX_NODE_NUM-1];
    int opt_cost = sequence_cost + min_cost[last_node][MAX_NODE_NUM-1];
    // 符合剪枝条件，则回溯
    //   0. 上一步搜索时向左分支
    //   1. shortest_distance[last_node][MAX_NODE_NUM-1] > 0 && sequence_distance + shortest_distance[
        sequence[-1]][MAX_NODE_NUM-1] > current best solution
    //   2. sequence_cost + min_cost[sequence[-1]][MAX_NODE_NUM-1] > 1500
    //   剪枝条件：0 且 (1或2)
    if (remove.size() == 0 && ((shortest_distance[last_node][MAX_NODE_NUM-1] > 0 &&
        opt_distance >= current_shortest_dist) || opt_cost > COST_UPPER_BOUND)) {
        // 剪枝回溯
        return;
    } else {
        // 判断是否是可行解
        if (remove.size() == 0 && shortest_distance[last_node][MAX_NODE_NUM-1] > 0) {
            // 上一步是向左分支且当前序列去往B的后继路径存在时，才可能产生可行解
            // 合并整个路径
            vector<int> tmp_shortest_path = sequence;
            int _last = shortest_path[sequence[sequence.size()-1]][MAX_NODE_NUM-1];
            vector<int> tmp_path;
            tmp_path.push_back(49);
            while (_last != sequence[sequence.size()-1]) {
                tmp_path.push_back(_last);
                _last = shortest_path[sequence[sequence.size()-1]][_last];
            }
            reverse(tmp_path.begin(), tmp_path.end());
            tmp_shortest_path.insert(tmp_shortest_path.end(), tmp_path.begin(), tmp_path.end());
            // 计算该路径的费用代价
            int total_cost = 0;
            for (int i = 1; i < tmp_shortest_path.size(); i++) {
                total_cost += cost_matrix[tmp_shortest_path[i-1]][tmp_shortest_path[i]];
            }
            // 若费用满足要求，则更新最优解
            if (total_cost <= COST_UPPER_BOUND) {
                current_shortest_dist = opt_distance;
                current_shortest_path = tmp_shortest_path;
            }
        }
        // 继续前进
        // 二叉树
        // 找到不在sequence和remove中且可以与last_node连通的结点c
        for (int i = 0; i < MAX_NODE_NUM; i++){
            vector<int>::iterator it_seq;
            vector<int>::iterator it_rm;
            it_seq = find(sequence.begin(), sequence.end(), i);
            it_rm = find(remove.begin(), remove.end(), i);
            if (i != last_node && it_seq == sequence.end() && it_rm == remove.end() &&
                distance_matrix[last_node][i] < INT_MAX_VALUE) {
                // 左子树
                sequence.push_back(i);
                vector<int> tmp_remove;
```

```
                search(sequence, tmp_remove, sequence_distance+distance_matrix[sequence[sequence.size()
                    -2]][sequence[sequence.size()-1]], sequence_cost+cost_matrix[sequence[sequence.size()
                    -2]][sequence[sequence.size()-1]], current_shortest_dist, current_shortest_path,
                    shortest_path, shortest_distance, min_cost, distance_matrix, cost_matrix);
                // 右子树
                sequence.pop_back();
                remove.push_back(i);
                search(sequence, remove, sequence_distance, sequence_cost, current_shortest_dist,
                    current_shortest_path, shortest_path, shortest_distance, min_cost, distance_matrix,
                    cost_matrix);
                break;
            }
        }
    }
}

int main()
{
    int distance_matrix[MAX_NODE_NUM][MAX_NODE_NUM]; // 距离矩阵
    int cost_matrix[MAX_NODE_NUM][MAX_NODE_NUM]; // 费用矩阵
    int shortest_distance[MAX_NODE_NUM-1][MAX_NODE_NUM]; // 记录各节点间的最短路径长度
    int min_cost[MAX_NODE_NUM-1][MAX_NODE_NUM]; // 记录各节点间的最小费用
    int shortest_path[MAX_NODE_NUM-1][MAX_NODE_NUM]; // 记录各节点间的最短路径
    int min_cost_path[MAX_NODE_NUM-1][MAX_NODE_NUM]; // 记录各节点间的最小费用对应的路径

    memset(shortest_distance, 0, sizeof(int)*MAX_NODE_NUM*MAX_NODE_NUM);

    char distance_file[] = "m1.txt";
    char cost_file[] = "m2.txt";
    parse_input(distance_file, distance_matrix);
    parse_input(cost_file, cost_matrix);
    TI(time);
    // 用dijkstra算法找出在没有其余条件限制下, 各节点到B的最短路径最小费用
    for (int i = 0; i < MAX_NODE_NUM - 1; i++) {
        dijkstra(distance_matrix, i, shortest_distance[i], shortest_path[i]);
        dijkstra(cost_matrix, i, min_cost[i], min_cost_path[i]);
    }

    // 分支定界法求有过路费约束的从A到B的最短路径
    vector<int> sequence;
    vector<int> remove;
    vector<int> current_shortest_path;
    sequence.push_back(0);
    int current_shortest_dist = INT_MAX_VALUE;
    search(sequence, remove, 0, 0, current_shortest_dist, current_shortest_path, shortest_path,
        shortest_distance, min_cost, distance_matrix, cost_matrix);
    TO(time, "Total");
    print_result(current_shortest_dist, current_shortest_path, cost_matrix);

    return 0;
}
```

3.1.3 多叉搜索树

```
/*
```

```
 * solve single source shortest path problem using branch and bound algorithm
 */

#include <iostream>
#include <fstream>
#include <string.h>
#include <vector>
#include <algorithm>
#include "timer.h"

#define MAX_NODE_NUM 50
#define INT_MAX_VALUE 9999
#define COST_UPPER_BOUND 1500

using std::ifstream;
using std::vector;
using std::find;
using std::cout;
using std::endl;
using std::reverse;

class Path
{
    public:
        int distance;
        int cost;
        vector<int> nodes;

        Path();
        void add_node(int node, int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM]);
        void set_path(int source, int shortest_distance[], int shortest_path[]);
        void update_cost(int cost_matrix[][MAX_NODE_NUM]);
        int length();
        int last();
        void add_path(Path path, int start);
        bool has(int node);
        void pop_back(int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM]);
};

Path::Path()
{
    this->distance = 0;
    this->cost = 0;
}

void Path::add_node(int node, int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM])
{
    this->nodes.push_back(node);
    int node_num = this->nodes.size();
    if (node_num >= 2) {
        int start = this->nodes[node_num-2];
        int end = this->nodes[node_num-1];
        this->distance += distance_matrix[start][end];
        this->cost += cost_matrix[start][end];
    }
}
```

```
void Path::set_path(int source, int shortest_distance[], int shortest_path[])
{
    int i = MAX_NODE_NUM-1;
    do {
        this->nodes.push_back(i);
        i = shortest_path[i];
    } while (i != source);
    if (source != MAX_NODE_NUM-1)
        this->nodes.push_back(i);
    reverse(this->nodes.begin(), this->nodes.end());
    this->distance = shortest_distance[MAX_NODE_NUM-1];
}

void Path::update_cost(int cost_matrix[][MAX_NODE_NUM])
{
    this->cost = 0;
    for (int i = 1; i < this->nodes.size(); i++) {
        this->cost += cost_matrix[this->nodes[i-1]][this->nodes[i]];
    }
}

int Path::length()
{
    return this->nodes.size();
}

int Path::last()
{
    return this->nodes[this->nodes.size()-1];
}

void Path::add_path(Path path, int start)
{
    this->nodes.insert(this->nodes.end(), path.nodes.begin()+start, path.nodes.end());
    this->distance += path.distance;
    this->cost += path.cost;
}

bool Path::has(int node)
{
    bool flag = false;
    for (int i = 0; i < this->nodes.size(); i++) {
        if (node == this->nodes[i]) {
            flag = true;
            break;
        }
    }
    return flag;
}

void Path::pop_back(int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM])
{
    int last = this->nodes.size();
    int start = this->nodes[last-2];
    int end = this->nodes[last-1];
    this->distance -= distance_matrix[start][end];
    this->cost -= cost_matrix[start][end];
```

```
    this->nodes.pop_back();
}

void parse_input(char *filename, int distance_matrix[][MAX_NODE_NUM])
{
    ifstream f(filename);
    for (int i = 0; i < MAX_NODE_NUM; i++) {
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            f >> distance_matrix[i][j];
        }
    }
}

void print_path(Path path)
{
    if (path.length() == 0) {
        cout << "No Solution" << endl;
        return;
    }
    cout << "path: ";
    for (int i = 0; i < path.length()-1; i++) {
        cout << path.nodes[i]+1 << "->";
    }
    cout << path.last()+1 << endl;
    cout << "distance: " << path.distance << endl;
    cout << "cost: " << path.cost << endl;
}

void dijkstra(int distance_matrix[][MAX_NODE_NUM], int source, Path &path)
{
    // found用于标记已经得到最短距离的目标结点
    int shortest_distance[MAX_NODE_NUM];
    int shortest_path[MAX_NODE_NUM];
    static int found[MAX_NODE_NUM];
    // 初始化为1
    memset(found, 0, sizeof(found));

    // 找出从源点到其相邻节点的最短路径
    found[source] = 1;
    for (int i = 0; i < MAX_NODE_NUM; i++) {
        shortest_distance[i] = distance_matrix[source][i];
        shortest_path[i] = source;
    }

    int min_distance;
    int min_distance_idx;

    // 找到从源点出发到其他各结点的最短路径
    for (int i = 1; i < MAX_NODE_NUM; i++) {
        min_distance = INT_MAX_VALUE;
        // 找到未被标记的路径最短的结点
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            if (found[j] == 0 && shortest_distance[j] < min_distance) {
                min_distance = shortest_distance[j];
                min_distance_idx = j;
            }
        }
```

```
        // 标记该结点为找到最短路径
        found[min_distance_idx] = 1;

        // 利用该该结点的最短路径更新其他结点的最短路径
        for (int j = 0; j < MAX_NODE_NUM; j++) {
            int dist_idx_j = distance_matrix[min_distance_idx][j]; // 从该结点到j的路径
            int new_distance = min_distance + dist_idx_j; // 从source到该结点的最短路径+从该结点到j的路径
            if (found[j] == 0 && dist_idx_j != INT_MAX_VALUE && new_distance < shortest_distance[j]) {
                shortest_distance[j] = new_distance;
                shortest_path[j] = min_distance_idx;
            }
        }
    }
    path.set_path(source, shortest_distance, shortest_path);

}

void search(Path sequence, Path &solution,
            Path shortest_path[], Path min_cost_path[],
            int distance_matrix[][MAX_NODE_NUM], int cost_matrix[][MAX_NODE_NUM])
{
    int last_node = sequence.last();
    int opt_distance = sequence.distance + shortest_path[last_node].distance;
    int opt_cost = sequence.cost + min_cost_path[last_node].cost;
    // 符合剪枝条件, 则回溯
    //   1. sequence长度 + sequence末尾节点到终点的最短路径长度 > 当前最优解
    //   2. sequence费用 + min_cost[sequence[-1]][MAX_NODE_NUM-1] > 1500
    //   剪枝条件: 0 且 (1或2)
    if (((shortest_path[last_node].distance > 0 &&
        opt_distance >= solution.distance) || opt_cost > COST_UPPER_BOUND)) {
        // 剪枝回溯
        return;
    } else {
        // 判断是否是可行解
        if (shortest_path[last_node].distance > 0) {
            // 当前序列去往B的后继路径存在时, 才可能产生可行解
            // 合并整个路径
            Path tmp;
            tmp.add_path(sequence, 0);
            tmp.add_path(shortest_path[last_node], 1);
            // 若费用满足要求, 则更新最优解
            if (tmp.cost <= COST_UPPER_BOUND) {
                solution = tmp;
            }
        }
        // 继续前进
        // 找到不在sequence和且可以与last_node连通的结点c
        for (int i = 0; i < MAX_NODE_NUM; i++){
            if (!sequence.has(i) &&
                distance_matrix[last_node][i] < INT_MAX_VALUE) {
                // 左子树
                sequence.add_node(i, distance_matrix, cost_matrix);
                search(sequence, solution, shortest_path, min_cost_path, distance_matrix, cost_matrix);
                sequence.pop_back(distance_matrix, cost_matrix);
            }
        }
    }
```

```
}

int main()
{
    int distance_matrix[MAX_NODE_NUM][MAX_NODE_NUM]; // 距离矩阵
    int cost_matrix[MAX_NODE_NUM][MAX_NODE_NUM]; // 费用矩阵
    Path shortest_path[MAX_NODE_NUM-1]; // 记录各节点间的最短路径
    Path min_cost_path[MAX_NODE_NUM-1]; // 记录各节点间的最小费用对应的路径

    char distance_file[] = "m1.txt";
    char cost_file[] = "m2.txt";
    parse_input(distance_file, distance_matrix);
    parse_input(cost_file, cost_matrix);
    TI(time);
    // 用dijkstra算法找出在没有其余条件限制下，各节点到B的最短路径和最小费用
    for (int i = 0; i < MAX_NODE_NUM - 1; i++) {
        dijkstra(distance_matrix, i, shortest_path[i]);
        // 更新最短路径的费用
        shortest_path[i].update_cost(cost_matrix);
        dijkstra(cost_matrix, i, min_cost_path[i]);
    }

    // 分支定界法求有过路费约束的从A到B的最短路径
    Path sequence;
    Path solution;
    solution.distance = INT_MAX_VALUE;
    sequence.add_node(0, distance_matrix, cost_matrix);
    int current_shortest_dist = INT_MAX_VALUE;
    search(sequence, solution, shortest_path, min_cost_path, distance_matrix, cost_matrix);
    TO(time, "Total");
    print_path(solution);

    return 0;
}
```

3.2 实验结果和耗时

两程序得到相同结果：最短路径为 1->3->8->11->15->21->23->26->32->37->39->45->47->50 对应的最短距离为 464，总的费用为 1448。分别运行两程序 1000 次，计算平均时间得：二叉搜索树版本耗时 3.044ms，多叉搜索树版本耗时 2.487ms。

3.3 复杂度分析

由于 dijkstra 算法的复杂度为 $O(n^2)$，算法对 B 之外的 n-1 个城市调用了 dijkstra 算法计算其到 B 的最短路径，所以这部分复杂度为 $O(n^3)$，而后面递归搜索的过程复杂度与所有城市之间的边数有关，由于使用了定界限制，减少了分支数目，使复杂度大大降低。而多叉树在搜索时不需进行城市的排除，所以比二叉树搜索策略更快一些。