离散数学实验报告

南京航空航天大学 计算机科学与技术学院 软件工程专业 161630213- 薛方岗

.

实验一：专业论文阅读

论文的基本信息

论文题目与来源

论文的题目为“An Implementation of the Behavior Annex in the AADL-toolset Osate2”，发表在2011年4月的Software Engineering Institute | Carnegie Mellon University期刊（或会议录），引用格式为：Lasnier G, Pautet L, Hugues J, et al. An Implementation of the Behavior Annex in the AADL-Toolset Osate2[C]// IEEE International Conference on Engineering of Complex Computer Systems. IEEE, 2011:332-337.

论文作者信息

论文的主要作者：

Gilles Lasnier, Laurent PautetInst. TELECOM - TELECOM ParisTech – LTCI Paris, F-75634 CEDEX 13, France Email: {firstname.lastname}@telecom-paristech.fr
Jérôme Hugues ISAE - Toulouse University Toulouse, 31056, France Email: jerome.hugues@isae.fr
Lutz Wrage SEI - Carnegie Mellon University Pittsburgh, PA, 15213, USA Email: lwrage@sei.cmu.edu

论文的主要内容

论文摘要

论文的研究背景：

Abstract-AADL是一种用于设计和分析高分辨率分布式和实时系统的建模语言。 作为AADL的拓展分支，该建模语言扩展了AADL模型以加强其分析功能。 其指定了AADL应用程序模型的行为。 因此，本附件的植入允许执行行为分析。 另外，由于有几个AADL的拓展，实施通用机制来支持其中的每一个都是具有挑战性的。

要解决的问题：

1）解析和分析几个AADL的子语言。
2）对不同的AST生产需要连接进行分析。
3）完成分析，要求确保与核心语言的一致性。

主要采用的方法：

开发了一个可扩展的开源平台AADL工具集OSATE2，它包括AADL前端，架构分析功能和扩展机制，将外部后端作为插件进行集成。

得到的研究结果：

通过重用多个OSATE2模块来驱动AADL-BA元素进而完成了对AADL模型的分析，开发出了新的编译器插件。

论文主体内容

论文对问题的描述：

Abstract-AADL是一种用于设计和分析高分辨率分布式和实时系统的建模语言。 作为AADL附件出版的嵌入式子语

言扩展了AADL模型以加强分析。但是，由于有几个AADL附件，实施通用机制来支持其中的每一个都是具有挑战性的。

论文解决问题的步骤：

1：将AADLBA编译器作为集成；
2：实现OSATE2的ECLIPSE插件；
3：将AADL-BA元模型作为构建编译器的几个模块的骨干分支。

论文结果的体现方式：

通过展示了研究人员如何使用AADL-BA模型来开发编译器的几个模块来说明研究结果。
具体产品为OSATE2，这是一个提供了AADL前端和附件的插件，驱动行为附件分析（解析器+分析器）。用于定义AADL和AADL-BA元模型的相同技术，可轻松跨两个元模型的导航，以及生成独特的持久XMI表示（AADL模型+行为元素），有助于将其用作外部后台的输入。
论文的创新点：
通过以一个可拓展的开源平台AADL工具集OSATE2为基础，将前后端结合起来对问题进行研究和实验，而不是局限于传统的分析方法。

相关工作和展望

论文的相关工作：

为了研究的需要开发了一个可扩展的开源平台AADL工具集OSATE2，它包括AADL前端，架构分析功能和扩展机制，将外部后端作为插件进行集成。

下一步工作：

通过外部后端的集成来分析行为自动机属性，例如作为模型检查器来验证死锁和基于模型的工具，以通过改进WCET估计和阻止共享资源的时间来增强调度分析。

实验二：构造命题逻辑合式公式的真值表

实验内容

根据用户输入的命题公式以及指定的分量的真值得出对应命题公式的真假值。

实验环境

电脑环境

系统：Ubuntu16.04LTS（Linux 4.10.0-37-generic x86_64）
处理器：Intel® Pentium(R) CPU N3700 @ 1.60GHz × 4
内存：7.7 GiB
操作系统类型：64位操作系统
图形：GeForce 920M/PCIe/SSE2

编译运行环境

CLion2017.2.3

编程语言

C/C++语言

实验算法

数据结构定义

本实验中主要使用C++中的map和string数据结构对数据进行处理，具体使用示例如下：

对真值的存储

考虑到合式公式的真值是和字符/字符串一一对应的，而真值为int（0或1），所以本实验使用map将合式公式和真值作为键值对存储到map中，事实证明这种数据处理方式取得了较为高效的数据存储效果。

对合式公式的存储

考虑到合式公式的不规则性以及在计算过程中需要频繁移动“指针”的位置，所以本实验使用string数据类型来存储要处理的合式公式，主要应用string的const_iterator对合式公式进行高效率的操作。
2）算法描述（包含输入输出说明）
本实验的主要算法思路是将需要判断真值的合式公式分解为小的合式公式“分而治之”，以括号为划分界限，计算出括号内”小的“合式公式的真值后再对总的合式公式进行真值判断。

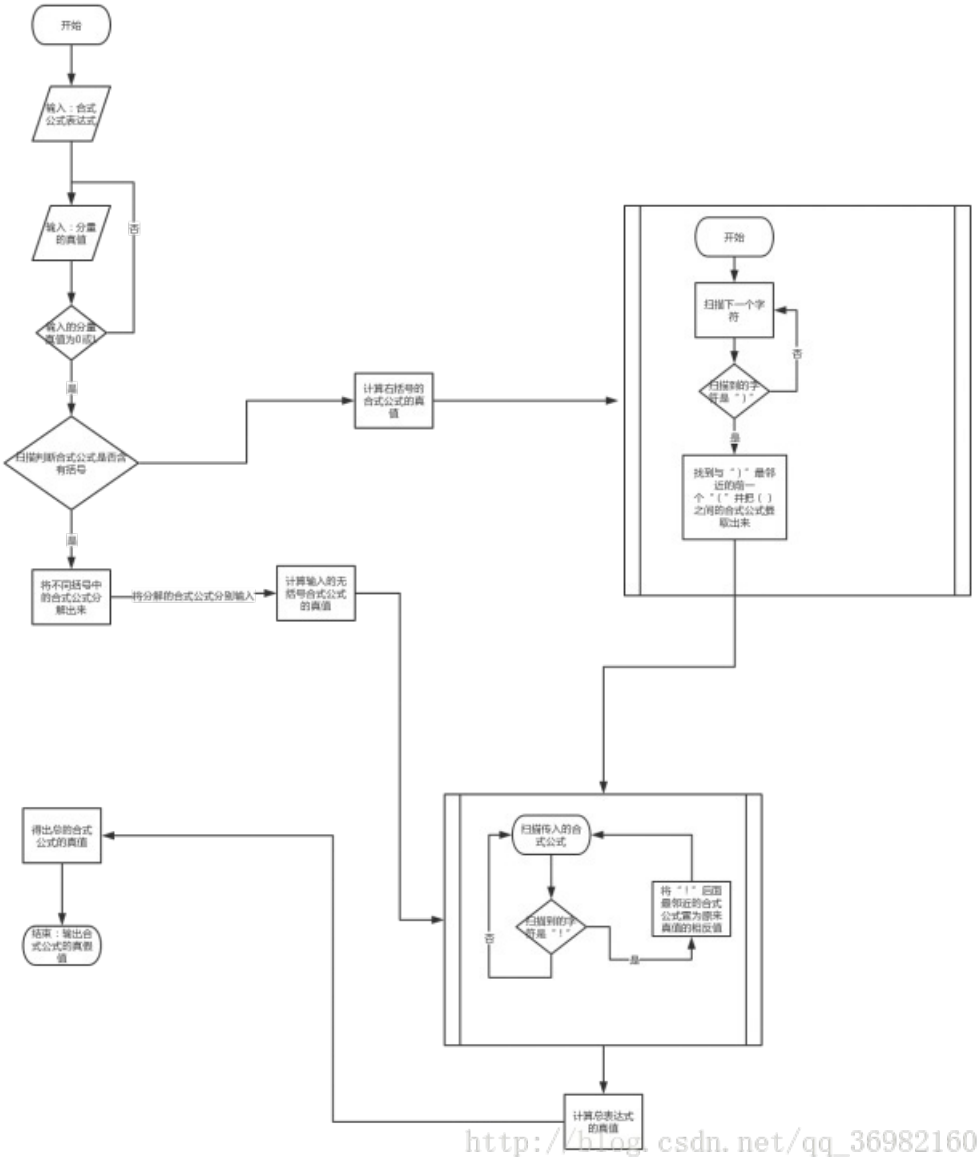
输入说明

输入合式公式时考虑到电脑输入字符的限制性，规定用！表示 否定 ， 用& 表示 合取 ， 用| 表示 析取

输出说明

本实验将会输出用户输入的总合式公式的真值，0代表”假“，1代表”真“。

算法流程图



实验结果

运行界面

运行结果

实验分析

实验优点分析：

1：在求解较为复杂的合式公式的真值时采用”分而治之“的思想，将复杂的合式公式分解为多个简单的合式公式今进而求解，事实证明这种解决问题的思想起到了较为理想的效果。
2：在进行具体的计算时可以充分利用C++语言的特性，巧妙地使用了C++中的map和string数据结构，达到了事半功倍的效果。

实验不足分析：

1：未能将完整合式公式的各个分部分的真值也列出来
2：未能实现图形界面从而增强用户人机交互的体验感

实验不足的改进方案：

1：在对总的合式公式尽情求解时另外再使用一张map存储括号内的合式公式及其真值，但是要注意的是在存储时要特别处理”！“
2：可以通过C++实现底层算法、java实现图形界面的方法实现本实验的图形界面，即在本实验C++代码的基础上做一个套壳封装

实验源码

部分核心代码如下

扫描括号并进行计算的方法（函数）：

```
int run_expr(string &Expr) { //扫描括号
    int sum_kh = 0;//括号数
    string::const_iterator iter;
    string::const_iterator addr_kh[50];//用于记录括号的位置
    char kind_kh[50];//用于记录括号的类型
    for (iter = Expr.begin();
         iter != Expr.end(); iter++) {
        if (*iter == '(' || *iter == ')') {
            addr_kh[sum_kh] = iter;
            kind_kh[sum_kh] = *iter;
            sum_kh++;
        }
    }
    if (0 == sum_kh)//如果没有括号
    {
        value = run_unkh_expr(Expr);
        return value;
    } else {
        int i = 0;
        for (; i <= sum_kh; i++) {
```

```
            if (kind_kh[i] == ')')//找到最内级的括号并跳出循环
                break;
        } //取出最内层没有括号的字符串
        string in_str = string(addr_kh[i - 1] + 1, addr_kh[i]); //算出最内层表达式的值
       ////((!p&q)|(p|q))&(p|v)
        value = run_unkh_expr(in_str);
        v_map[in_str] = value;


        static char var = '1';
        value_map[var] = value;//将括号整体设为一个字符‘1’
        string::const_iterator ite = addr_kh[i - 1];//"字符(的位置" //判断（是不是表达式的开头
        string::const_iterator init_i;
        bool is_begin = false;
        if (ite == expr.begin()) { is_begin = true; }
        else { init_i = addr_kh[i - 1] - 1; }
        {
            expr.erase(ite, addr_kh[i] + 1);//删除掉最内层表达式包括括号在内
        }   //再在删除的地方插入新的字符‘1’作为标记


        if (is_begin == true) { expr = var + expr; }
        else { expr.insert(init_i + 1, var); }
        var = var + 1;
        value = run_expr(expr);
        return value;
    }
} //求主析取范式和主合取范式的函数
```

处理具体合式关系的方法（函数）：

```
int deal_cal(int par1, char par2,
             int par3) {

    switch (par2) {
        case '&'://合取
            return par1 && par3;

        case '|'://析取
            return par1 || par3;

        default:
            cout << "有某些命题的真值错误" << endl;
            break;
    }
}
```

具体项目见：

https://github.com/DMRFWIN/-DiscreteMathematicsExperiment_TruthTable.git

实验三：TSP问题求解（图形界面版）

实验内容

TSP问题求解

实验环境

电脑环境

系统：Ubuntu16.04LTS（Linux 4.10.0-37-generic x86_64）
处理器：Intel® Pentium(R) CPU N3700 @ 1.60GHz × 4
内存：7.7 GiB
操作系统类型：64位操作系统
图形：GeForce 920M/PCIe/SSE2

编译运行环境

Intellij IDEA Community

编程语言

JAVA语言

实验算法

数据结构定义

本实验中用到了java中较为经典的MAP、LIST等数据结构。

算法1描述（包含输入输出说明）

算法思想描述

a.从某一个城市开始，每次选择一个城市，直到所有的城市被走完。
b.每次在选择下一个城市的时候，只考虑当前情况，保证迄今为止经过的路径总距离最小。

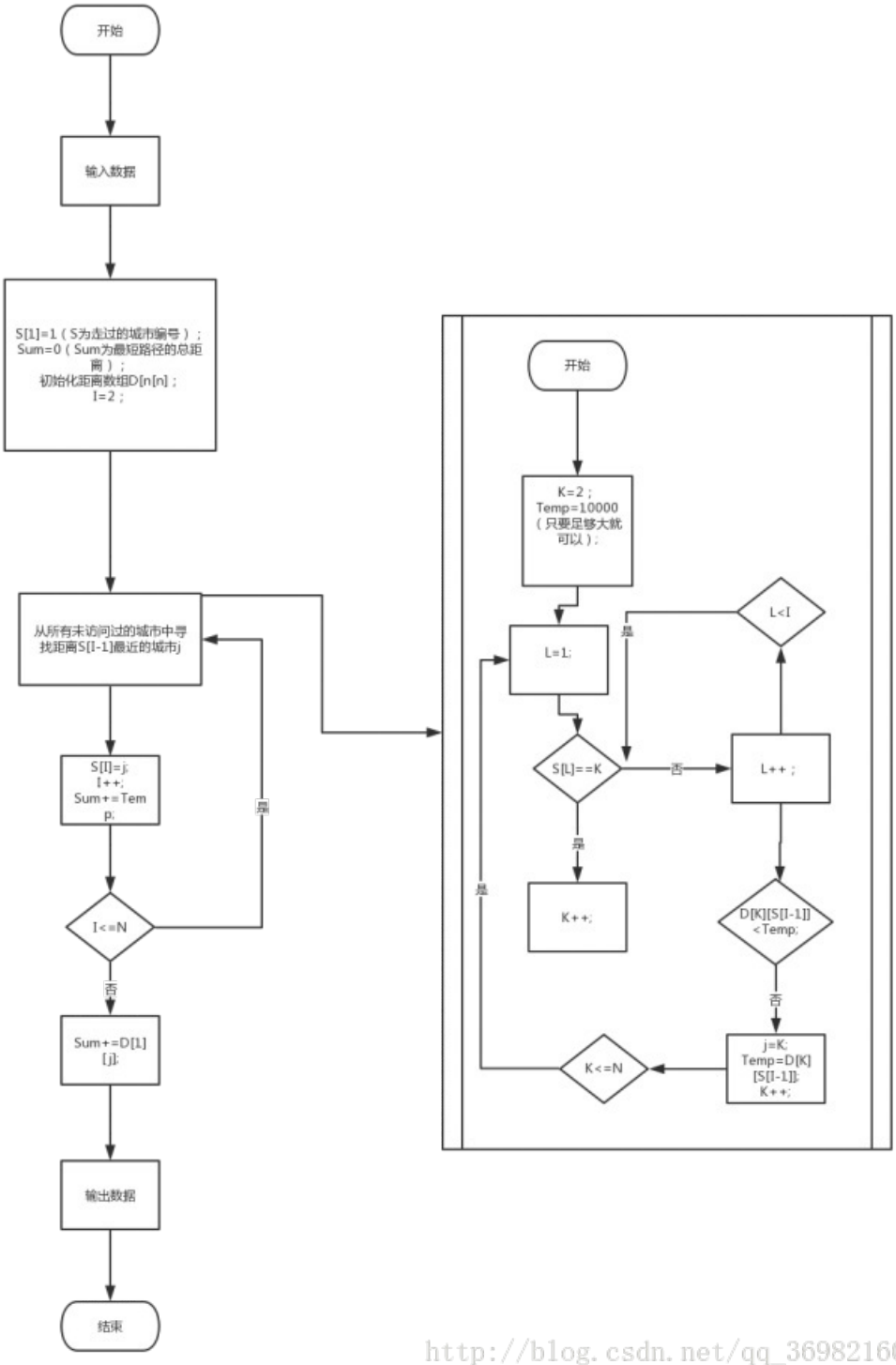
输入描述

输入数据可以选择手动输入，也可选择从软件运行的本机中选择数据文件导入计算。

输出描述

输出为两张表，一张为输入数据组成的数据表，另一张为最短路径经过的城市的代号构成的输出表，在表的末尾会输出对应算法计算的最短路径值。

算法1流程图



算法2描述

算法2使用的是回溯法（试探法）：

回溯法描述

从一条路往前走，能进则进，不能进则退回来，换一条路再试。

用回溯算法解决问题的一般步骤为：

1、定义一个解空间，它包含问题的解。
2、利用适于搜索的方法组织解空间。
3、利用深度优先法搜索解空间。
4、利用限界函数避免移动到不可能产生解的子空间。

输入描述

输入数据可以选择手动输入，也可选择从软件运行的本机中选择数据文件导入计算。

输出描述

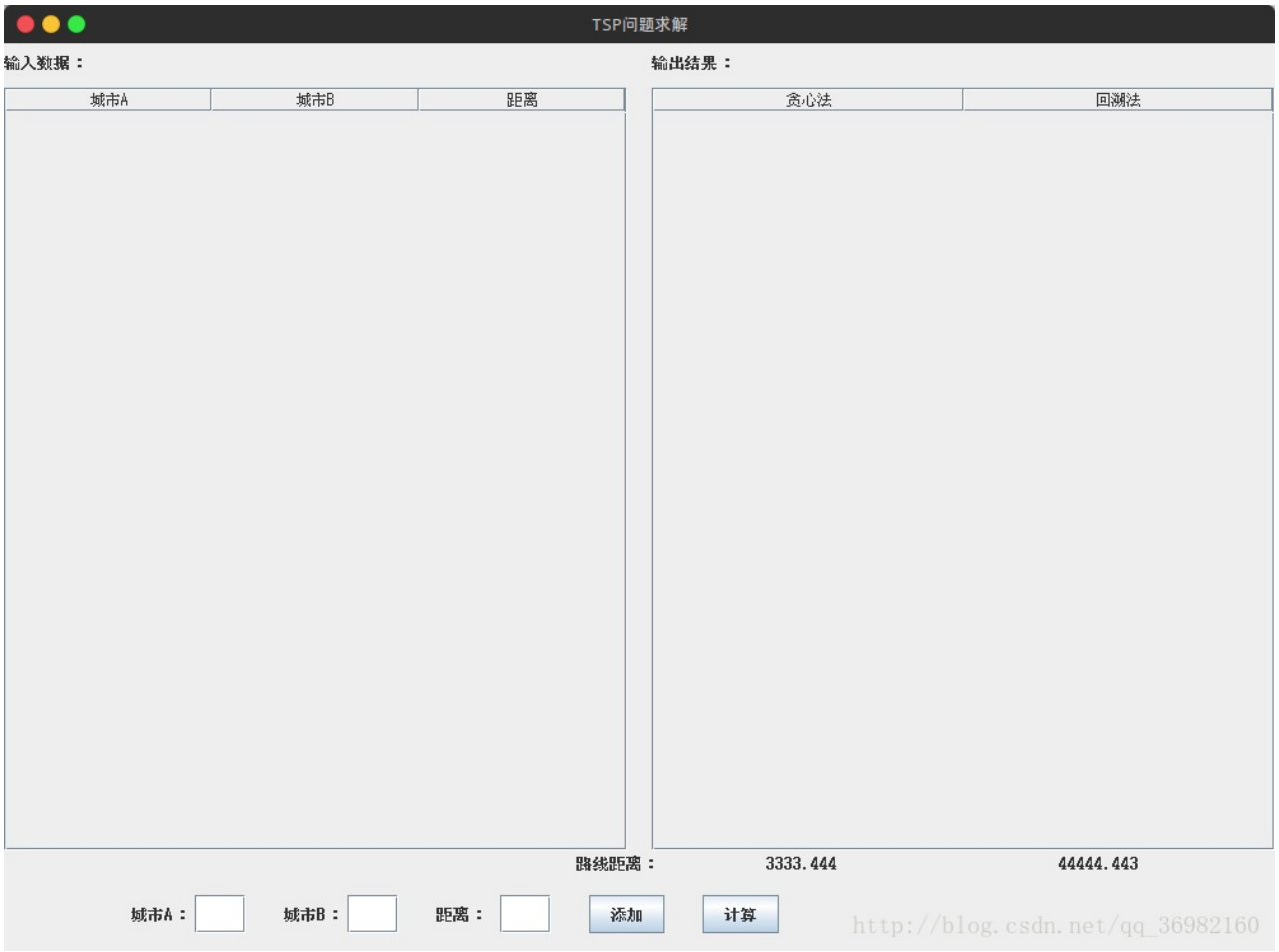
输出为两张表，一张为输入数据组成的数据表，另一张为最短路径经过的城市的代号构成的输出表，在表的末尾会输出对应算法计算的最短路径值。####算法2流程图

实验结果

运行界面

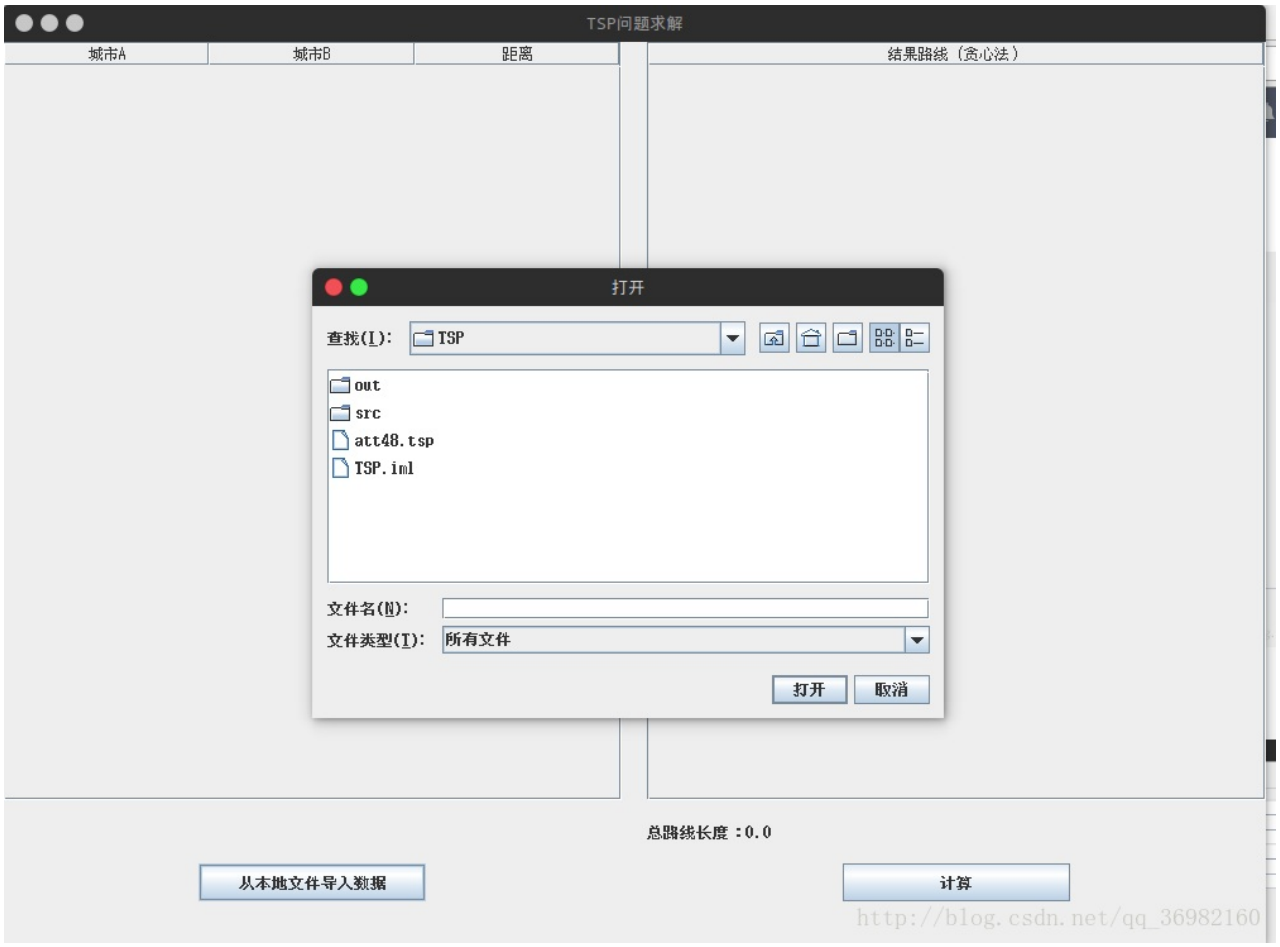
数据源选择界面：

手动输入数据界面：



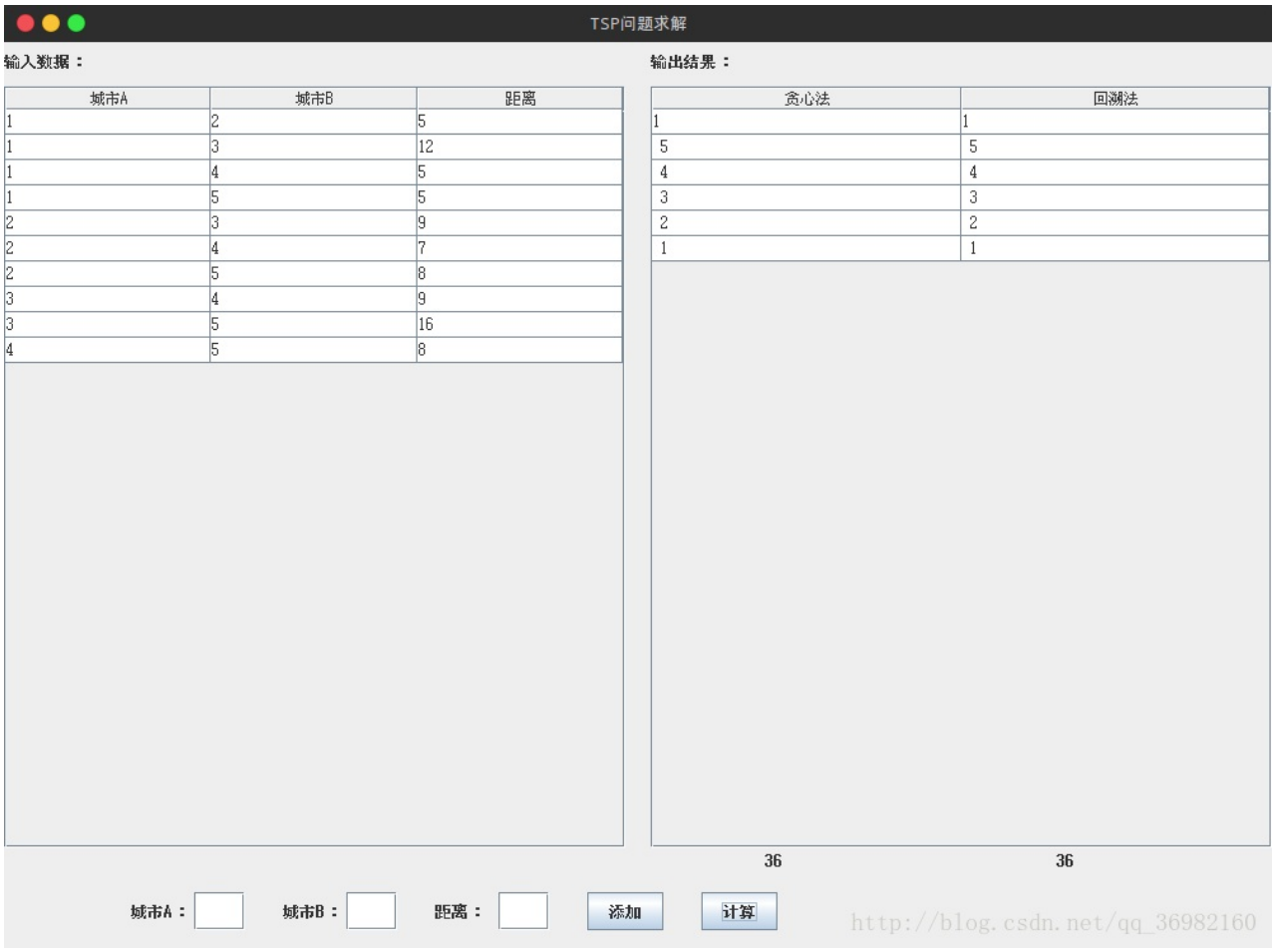
从本机选择数据文件界面：


TSP问题求解
城市A
城市B
距离
结果路线（贪心法）
总路线长度：0.0
从本地文件导入数据
计算
http://blog.csdn.net/qq_36982160

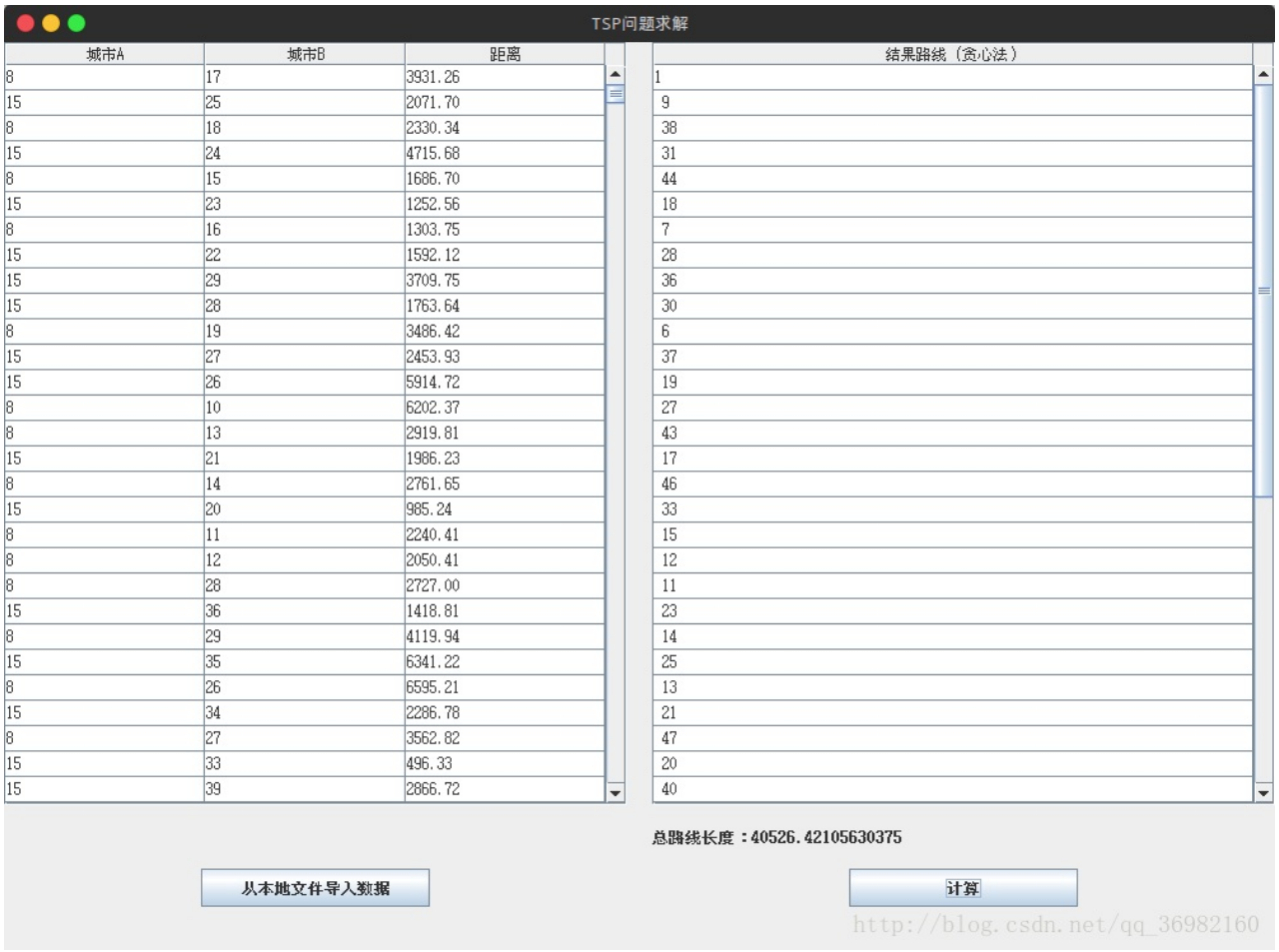


运行结果截图

手动输入界面测试运行结果：



从本机选择文件测试结果：



项目代码

核心算法代码

回溯法:

```
package Algorithm;

import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class Back {
    private int distance[][];
    private int x[];
    private int b[];
    private int cl = 0;
    private int k = 10000;


    private int cityNum;

    public Back() {
    }

    public Map<String, String> GetMinRoadByBack(Map<String, Integer> roadBeans, String str_num) {
        InitData(roadBeans, str_num);
```

```
        int i;

        Traveling(2);

        b[cityNum] = b[0];

        Map<String, String> result = new HashMap<>();
        result.put("result_road", Arrays.toString(b));
        result.put("result_value", String.valueOf(k));

        return result;
    }

    private void Traveling(int t) {
        int j;
        if (t > cityNum) {
            if (distance[x[cityNum]][1] != -1 && (cl + distance[x[cityNum]][1] < k)) {
                for (j = 1; j <= cityNum; j++)
                    b[j - 1] = x[j];
                k = cl + distance[x[cityNum]][1];
            }
        } else {
            for (j = t; j <= cityNum; j++) {
                if (distance[x[t - 1]][x[j]] != -1 && (cl + distance[x[t - 1]][x[j]] < k)) {

                    int p = x[t];
                    x[t] = x[j];
                    x[j] = p;

                    cl += distance[x[t - 1]][x[t]];
                    Traveling(t + 1);
                    cl -= distance[x[t - 1]][x[t]];

                    p = x[t];
                    x[t] = x[j];
                    x[j] = p;
                }
            }
        }
    }


    private void InitData(Map<String, Integer> roadBeans, String str_num) {

        cityNum = str_num.length();
        distance = new int[cityNum + 1][cityNum + 1];
        x = new int[cityNum + 1];
        b = new int[cityNum + 1];

        for (int i = 1; i <= cityNum; i++) {
            x[i] = i;
            b[i] = 0;
        }


        for (Map.Entry<String, Integer> entry : roadBeans.entrySet()) {

            String s = entry.getKey();
            String[] split = s.split(",");
```

```
            int a = Integer.parseInt(split[0]);
            int b = Integer.parseInt(split[1]);

            int length = entry.getValue();


            distance[a][b] = length;
        }
    }
}
```

贪心法：

```
package Algorithm;

import Bean.RoadBean;


import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class Greedy {


    private Map<RoadBean, Double> roadBeanst;

    private int cityNum; // 城市数量
    private int[][] distance; // 距离矩阵
    private Double[][] distance2;

    private int[] colable;//代表列，也表示是否走过，走过置0
    private int[] row;//代表行，选过置0


    public Map<String, String> GetMinRoadByTx(Map<String, Integer> roadBeans, String str_num) {

        Map<String, String> result = new HashMap<>();

        InitData(roadBeans, str_num);

        int[] temp = new int[cityNum];
        int[] path = new int[cityNum + 1];
        int path_num = 0;
        path[path_num++] = 1;


        int s = 0;//计算距离
        int i = 0;//当前节点
        int j = 0;//下一个节点
        //默认从0开始
        while (row[i] == 1) {
            //复制一行
            for (int k = 0; k < cityNum; k++) {
                temp[k] = distance[i][k];
                //System.out.print(temp[k]+" ");
            }
            //System.out.println();
```

```
            //选择下一个节点，要求不是已经走过，并且与i不同
            j = selectmin(temp);
            //找出下一节点
            row[i] = 0;//行置0，表示已经选过
            colable[j] = 0;//列0，表示已经走过

            path[path_num++] = j + 1;
            //System.out.println(i + "-->" + j);
            //System.out.println(distance[i][j]);
            if (distance[i][j] == 0) {
                s += distance[j][i];
            } else {
                s += distance[i][j];
            }

            i = j;//当前节点指向下一节点
        }

        result.put("result_road", Arrays.toString(path));
        result.put("result_value", String.valueOf(s));

        return result;
    }


    public int selectmin(int[] p) {
        int j = 0, m = p[0], k = 0;
        //寻找第一个可用节点，注意最后一次寻找，没有可用节点
        while (colable[j] == 0) {
            j++;
            //System.out.print(j+" ");
            if (j >= cityNum) {
                //没有可用节点，说明已结束，最后一次为 *-->0
                m = p[0];
                break;
                //或者直接return 0;
            } else {
                m = p[j];
            }
        }
        //从可用节点J开始往后扫描，找出距离最小节点
        for (; j < cityNum; j++) {
            if (colable[j] == 1) {
                if (m >= p[j]) {
                    m = p[j];
                    k = j;
                }
            }
        }
        return k;
    }


    private void InitData(Map<String, Integer> roadBeans, String str_num) {

        cityNum = str_num.length();

        distance = new int[cityNum][cityNum];

        colable = new int[cityNum];
```

```
        colable[0] = 0;
        for (int i = 1; i < cityNum; i++) {
            colable[i] = 1;
        }

        row = new int[cityNum];
        for (int i = 0; i < cityNum; i++) {
            row[i] = 1;
        }

        for (Map.Entry<String, Integer> entry : roadBeans.entrySet()) {

            String s = entry.getKey();
            String[] split = s.split(",");
            int a = Integer.parseInt(split[0]);
            int b = Integer.parseInt(split[1]);

            int length = entry.getValue();

            a--;
            b--;

            distance[a][b] = length;
        }
    }

    public Map<String, String> GetMinRoadByTx(Map<String, Double> stringDoubleMap, int citynum) {
        Map<String, String> result = new HashMap<>();
        this.cityNum = citynum;
        InitData(stringDoubleMap);


        Double[] temp = new Double[cityNum];

        for (int i = 0; i < cityNum; i++) {
            temp[i] = 0.0;
        }
        int[] path = new int[cityNum + 1];
        int path_num = 0;
        path[path_num++] = 1;


        Double s = 0.0;//计算距离
        int i = 0;//当前节点
        int j = 0;//下一个节点
        //默认从0开始
        while (row[i] == 1) {
            //复制一行
            for (int k = 0; k < cityNum; k++) {
                temp[k] = distance2[i][k];
                //System.out.print(temp[k]+" ");
            }
            //System.out.println();
            //选择下一个节点，要求不是已经走过，并且与i不同
            j = selectmin(temp);
            //找出下一节点
            row[i] = 0;//行置0，表示已经选过
            colable[j] = 0;//列0，表示已经走过

            path[path_num++] = j + 1;
```

```
            //System.out.println(i + "-->" + j);
            //System.out.println(distance[i][j]);
            if (distance2[i][j] == 0) {
                s += distance2[j][i];
            } else {
                s += distance2[i][j];
            }

            i = j;//当前节点指向下一节点
        }

        result.put("result_road", Arrays.toString(path));
        result.put("result_value", String.valueOf(s));

        return result;

    }

    private int selectmin(Double[] p) {
        int j = 0, k = 0;
        Double m = p[0];
        //寻找第一个可用节点，注意最后一次寻找，没有可用节点
        while (colable[j] == 0) {
            j++;
            //System.out.print(j+" ");
            if (j >= cityNum) {
                //没有可用节点，说明已结束，最后一次为 *-->0
                m = p[0];
                break;
                //或者直接return 0;
            } else {
                m = p[j];
            }
        }
        //从可用节点J开始往后扫描，找出距离最小节点
        for (; j < cityNum; j++) {
            if (colable[j] == 1) {
                if (m >= p[j]) {
                    m = p[j];
                    k = j;
                }
            }
        }
        return k;
    }

    private void InitData(Map<String, Double> stringDoubleMap) {


        distance2 = new Double[cityNum][cityNum];

        colable = new int[cityNum];
        colable[0] = 0;
        for (int i = 1; i < cityNum; i++) {
            colable[i] = 1;
        }

        row = new int[cityNum];
        for (int i = 0; i < cityNum; i++) {
            row[i] = 1;
```

```
        }

        for (Map.Entry<String, Double> entry : stringDoubleMap.entrySet()) {

            String s = entry.getKey();
            String[] split = s.split(",");
            int a = Integer.parseInt(split[0]);
            int b = Integer.parseInt(split[1]);

            Double length = entry.getValue();
            a--;
            b--;


            distance2[a][b] = length;
        }

    }
}
```

具体代码见：

java图形界面版TSP问题求解源码

实验三：TSP问题求解（没有图形界面的黑框框版）

实验内容

TSP问题求解

实验环境

电脑环境

系统：Ubuntu16.04LTS（Linux 4.10.0-37-generic x86_64）
处理器：Intel® Pentium(R) CPU N3700 @ 1.60GHz × 4
内存：7.7 GiB
操作系统类型：64位操作系统
图形：GeForce 920M/PCIe/SSE2

编译运行环境

CLion2017.2.3

编程语言

C/C++语言

实验算法

数据结构定义

本实验考虑到所要处理的城市数量不会太多，所以使用比较简单易懂的矩阵（二维数组）来存储图，即S[a][b]=c表示城市a和城市b之间的距离为c。

算法1描述（包含输入输出说明）

算法1采用的是贪心算法

算法思想描述

a.从某一个城市开始，每次选择一个城市，直到所有的城市被走完。
b.每次在选择下一个城市的时候，只考虑当前情况，保证迄今为止经过的路径总距离最小。

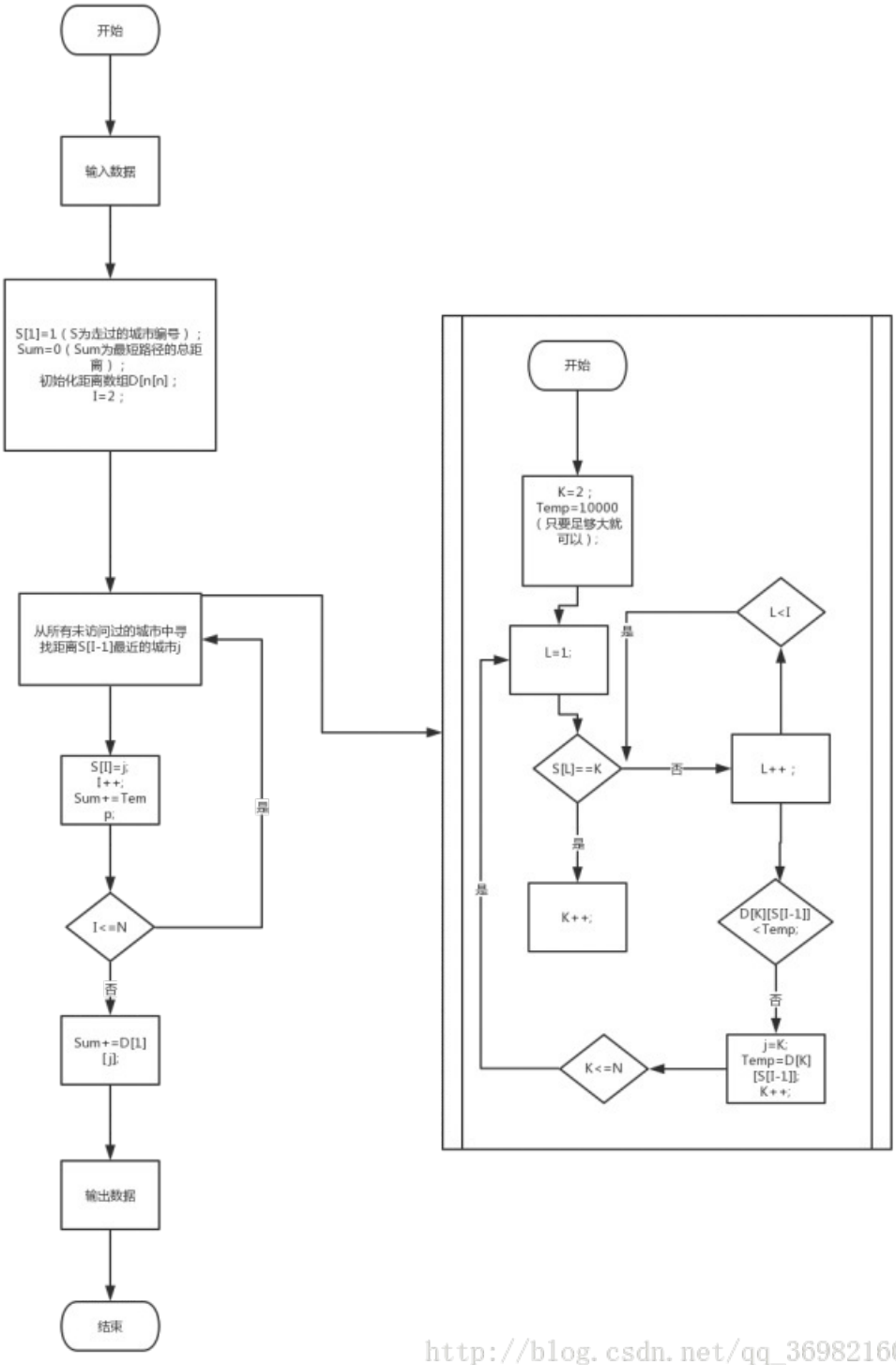
输入描述

输入时根据提示前两行输入城市数量n和道路数量m，接下来m行，每行3个数，表示m条道路的起点终点以及权重。

输出描述

输出为2行，第一行为该算法求解的经过路径，格式为“a–>b–>c–>d–>a”，第二行为使用此种走法的最短路径值。

算法1流程图



算法2描述

算法2使用的是回溯法（试探法）：

回溯法描述

从一条路往前走，能进则进，不能进则退回来，换一条路再试。

用回溯算法解决问题的一般步骤为：

1、定义一个解空间，它包含问题的解。
2、利用适于搜索的方法组织解空间。
3、利用深度优先法搜索解空间。
4、利用限界函数避免移动到不可能产生解的子空间。

输入描述：

输入时根据提示前两行输入城市数量n和道路数量m，接下来m行，每行3个数，表示m条道路的起点终点以及权重。

输出描述

输出为2行，第一行为该算法求解的经过路径，格式为“a–>b–>c–>d–>a”，第二行为使用此种走法的最短路径值。

算法2流程图


开始
输入城市数量以及距离数据
遍历树并令第一条可行回路的解和值为当前最优解和最优值
继续遍历并与当前最优值和最优解比较，最优则替换
否
到达结束点？
是
输出最优值和最优解
结束

实验结果

运行界面

运行结果截图

项目代码

核心算法代码

回溯法：

```
void GetMinRoadByHs() {
    int i;
    for (i = 1; i <= n; i++) {
        x[i] = i;
        b[i] = 0;
    }
    Traveling(2);
    cout << "城市路线：" << endl;
    for (i = 1; i <= n; i++)
        cout << b[i] << "-->";
    cout << b[1];
    cout << endl;
    cout << "最短路线长度：" << endl;
    cout << k << endl;
}

void Traveling(int t) {
    int j;
    if (t > n) {
        if (g[x[n]][1] != -1 && (cl + g[x[n]][1] < k)) {
            for (j = 1; j <= n; j++)
                b[j] = x[j];
            k = cl + g[x[n]][1];
```

```
        }
    } else {
        for (j = t; j <= n; j++) {
            if (g[x[t - 1]][x[j]] != -1 && (cl + g[x[t - 1]][x[j]] < k)) {
                swap(x[t], x[j]);
                cl += g[x[t - 1]][x[t]];
                Traveling(t + 1);
                cl -= g[x[t - 1]][x[t]];
                swap(x[t], x[j]);
            }
        }
    }
}
```

贪心法：

```
void GetMinRoadByTx() {

    /**
     * S[n]用于存储已经访问过的城市
     * D[a][b]用于存储a和b之间的距离
     * flag 访问过为1，没访问过为0
     * i至今已经访问过的城市
     */
    int j, k, l;
    int i;
    i = 1;
    int beng = i;
    int sum = 0;
    int Dtemp;
    int flag;
    do {
        k = 1;
        Dtemp = 10000;
        do {
            l = 0;
            flag = 0;
            do {
                if (S[l] == k) {//判断该城市是否已被访问过，若被访问过，
                    flag = 1;//则flag为1
                    break;//跳出循环，不参与距离的比较
                } else
                    l++;
            } while (l < i);
            if (flag == 0 && D[k][S[i - 1]] < Dtemp) {/*D[k][S[i - 1]]表示当前未被访问的城市k与上一个
已访问过的城市i-1之间的距离*/
                j = k;//j用于存储已访问过的城市k
                Dtemp = D[k][S[i - 1]];//Dtemp用于暂时存储当前最小路径的值
            }
            k++;
        } while (k < n);
        S[i] = j;//将已访问过的城市j存入到S[i]中
        i++;
        sum += Dtemp;//求出各城市之间的最短距离，注意：在结束循环时，该旅行商尚未回到原出发的城市
    } while (i < n);
    sum += D[0][j];//D[0][j]为旅行商所在的最后一个城市与原出发的城市之间的距离
    for (j = 0; j < n; j++) { //输出经过的城市的路径
        cout << S[j] + 1 << "-->";
    }
```

```
    cout << beng;
    cout << "\n" << sum;
}
```

具体代码见：

https://github.com/DMRFWIN/DiscreteMathematicsExperiment_TSP

实验分析

2个算法的对比

从实验结果很明显可以看出由于贪心法的“只关心局部最小” 原则，导致最终求得的结果并不一定是最小的，相比之下回溯法求得的结果明显是最小的，但是回溯法的算法思想和代码复杂度明显高于贪心法，所以得出结论，当并不需要准确的最小值时我们可以用贪心法当对结果的准确性要求较高时我们应该使用回溯法。

实验总结和体会

通过本次实验我有以下收获和感受：

如何把课堂上学到的离散数学的理论知识应用到实际的编程中，懂得了离散数学是如何提高变成效率的。
懂得了如何高效地阅读英文论文、从中提取精华部分，对以后阅读英文文档/论文有很大的帮助。
对TSP问题有更加清晰的认识，知道如何编程实现不同的算法解决TSP问题。
对合式公式的概念认识更加清楚，巩固了合式公式的计算方法。

对实验的建议：

实验可以采用多人（3人以下）组队的方式合作完成，方便提高同学们的团队协作能力。