实验报告：多关键字排序，全国交通咨询模拟

组员：刘士祺(1706) 陈国凯(1703) 李非冬(1706) 日期：2019.7.5

一．需求分析

1. 多关键字排序：按用户需求对数据按次序进行多关键字排序，要求使用 LSD 方法进行排序
2. 全国交通咨询模拟：为不同目的的旅客提供不同策略进行路径查找，提供三种不同的旅行方案。支持用户自行编辑城市、路线信息。

二．设计

多关键字排序

本程序分为了三个部分，分别是文件选择，关键字选择并排序，时间展示与结果保存。在打开程序时，将接受用户选择并将待处理文件传递给下一部分。在下一部分中，程序将获得用户选择的若干关键字并基于此进行排序。在最后一部分中，程序将把排序后数据存入用户所指定的文件。

具体实现：

对于每一条记录，构造了以下结构进行存储:

```
public class Row {
    public string[] elems;

    public Row(string[] elems) {
        this.elems = elems;
    }
}
```

然后对于多组数据进行归并排序:

```
delegate bool Compare(Row i, Row j);

void merge_sort(Row[] elems, Compare comp) {
    int len = elems.length;
    if (len == 1 || len == 0) return;
    Row[] left = elems[0:len / 2];
    Row[] right = elems[len / 2:len];
    merge_sort(left, comp);
    merge_sort(right, comp);
    int i = 0, j = 0, k = 0;
    while (k < elems.length) {
        if (i < left.length && (j == right.length || !comp(left[i],
right[j]))) {
            elems[k] = left[i];
            i += 1;
```

```
        } else {
            elems[k] = right[j];
            j += 1;
        }
        k += 1;
    }
}
```

由于用户输入未知，其最高优先级与最低优先级的数据类型可能为字符串，整数，小数等，所以难以使用”分配””收集”的方法进行排序，程序将使用归并进行排序。

将其调用：

```
public void inner_sort() {
    int i = key_word_1.get_active(), j = key_word_2.get_active(), k
= key_word_3.get_active();
    int    type_1    =    this.type_1.get_active(),    type_2    =
this.type_2.get_active(), type_3 = this.type_3.get_active();
    int   order_1   =   this.order_1.get_active(),   order_2   =
this.order_2.get_active(), order_3 = this.order_3.get_active();
    merge_sort(rows, (a, b) => {
        if (i == -1 || equal(a, b, i)) {
            if (j == -1 || equal(a, b, j)) {
                if (k != -1)
                    return compare(a, b, k, type_3, order_3);
                else return true;
            } else
                return compare(a, b, j, type_2, order_2);
        } else {
            return compare(a, b, i, type_1, order_1);
        }
    });
}
```

在代码中，原本使用三次稳定的归并排序，为了优化运行时间，将三次排序改为了一次排序，其中对比较函数进行了改写。以下为 compare 与 equal 函数：

```
bool compare(Row a, Row b, int key_word, int type, int order) {
    if (type == 0) {
        if (order == 0)
            return a.elems[key_word] > b.elems[key_word];
        else
            return a.elems[key_word] < b.elems[key_word];
   } else if (type == 1) {
       if (order == 0)
            return          int.parse(a.elems[key_word])          >
int.parse(b.elems[key_word]);
        else
            return          int.parse(a.elems[key_word])          <
```

```
int.parse(b.elems[key_word]);
    } else {
        if (order == 0)
            return          double.parse(a.elems[key_word])          >
double.parse(b.elems[key_word]);
        else
            return          double.parse(a.elems[key_word])          <
double.parse(b.elems[key_word]);
    }
}

bool equal(Row a, Row b, int key_word) {
    return a.elems[key_word] == b.elems[key_word];
}
```

全国交通咨询模拟

本程序分为数据收集，信息处理与路径查找三个组件。

数据收集部分我们实现了一个 12306 网站爬虫，爬取了一周内全国车辆信息（具体实现见 12306/12306.py）。在数据处理部分我们对获取的列车信息进行标准化，去除无用的结构信息，便于进行数据处理。在路径寻找部分采用 STL set 优化的 Dijkstra 算法，不同乘车策略下使用不同的路径权值。

我们独创的加入了 UI 界面，对用户十分友好。用户可以通过下拉条对偏好方案（即时间、金钱或换乘次数优先）及起始与终止位置进行选择，也可通过添加按钮自行增删城市与路径。

由于 12306 网站能集中获取全国铁路信息而航班信息的获取没有统一的平台，我们约定城际高铁（C 字头动车组列车）在程序中被当作航班处理。由于题目要求不考虑航班和火车的相互换乘，则本程序中实际不考虑 C 字头列车与其他列车的相互换乘。

基本定义

在算法部分，我们对于读取到的列车数据采用如下结构体存储：

```cpp
struct trans{
    int v_type;
    /*
     * type 0: flight (actually C trains)
     * type 1: trains (actually non-C trains)
    */
    std::string v_name;
    /*
     * the train's name in string
    */
    std::string v_start;
    /*
     * the name of the start station
    */
    std::string v_end;
    /*
     * the name of the end station
    */
    std::string start_day;
    /*
     * the day of setting off, in string
    */
    std::string end_day;
    /*
     * the day of arrival, in string
    */
    std::string start_clock;
    /*
     * the time of setting off, in string
    */
    std::string end_clock;
    /*
     * the time of arrival
    */
    int start_time; // the start time in integer
    int end_time;   // the end time in integer
    int length;     // the length of the train/flight
    std::string length_str; // the length of the train/flight in string
    std::string seat_type;  // the type of seat, in string
    double price;           // the price of the train/flight

    /*
     * sample input format:
     * K6735 乌鲁木齐 Mon 13:28:00 石河子 Mon 14:53:00 1:25:00 硬座 21.5
    */
};
```

从代码中注释可见各记录信息的用途。

为进一步简化信息存储格式，我们额外使用了 STL vector 存储边权信息，使用 STL map 存储城市与其数字编码的对应关系，使用 STL set 在 Dijkstra 算法中加速最短边的提取。其定义如下：

```
struct node {
    /*
     * node is designed for storing the edges in our graph.
     * we need to store the destination and the departure point for Dijkstra to work
     * the offset stores where the edge came from in vector, so we can find
     * info that are not included in this struct
     */
    int to; // the destination
    int offset;
    int beg;
    double len; // the path's length

};
extern std::vector<trans>v; // the vector for all the trains available
extern std::set<std::string>cities; // the set of different cities
extern int city_counts;             // the total city count
extern std::map<std::string,int>m;  // make connections between cities' name and the node number
extern std::vector<node>edge[40000];// adjacent lists
```

即我们使用 v 存储图上所有的边的信息，在最短路算法中转换成邻接表（后续的 edge）使用；使用 cities 存储所有城市信息，在检查某城市是否存在于当前数据中时可快速（O(logn)复杂度）完成查找；使用 m 建立城市名称与城市编号间的映射，便于使用最短路算法时用数字编码城市节点信息；使用 edge 建立邻接表，在 Dijkstra 算法中使用。

功能实现

从文件中读取列车信息：

```
void MainWindow::initDatabase() {
    // we try to read the database from files
    int hours,minutes,seconds;
    struct trans rec;
#define FILEPATH "/Users/cgk/Documents/CPP/data_structure/12306/merge/new.txt"
    freopen(FILEPATH,"r",stdin);
    std::ios::sync_with_stdio(false);
    int counts = 0;
    while (!(std::cin.eof())) {
        std::cout<<++counts<<"\n";
        std::cin>>rec.v_name>>rec.v_start>>rec.start_day>>rec.start_clock>>rec.v_end>>rec.end_day>>rec.end_clock>>rec.length_str>>rec.seat_

        if (rec.v_name[0] == 'C') {
            rec.v_type = 0;
        } else {
            rec.v_type = 1;
        }
        std::sscanf(rec.start_clock.c_str(),"%d:%d:%d",&hours,&minutes,&seconds);
        rec.start_time = day_judger(rec.start_day) + hours * 3600 + minutes * 60 + seconds;
        std::sscanf(rec.end_clock.c_str(),"%d:%d:%d",&hours,&minutes,&seconds);
        rec.end_time = day_judger(rec.end_day) + hours * 3600 + minutes * 60 + seconds;
        rec.length = rec.end_time - rec.start_time;
        if (rec.start_time < 0 || rec.end_time < 0) {
            // we have got a wrong data base
            std::cout<<"The database is corrupted!\n";
            std::exit(0);
        }
        cities.insert(rec.v_start);
        cities.insert(rec.v_end);
        v.push_back(rec);
    }

}
```

我们使用字符串形式读取数据，然后对于非字符串类型数据（如车次价格、运行时长等）另行转化成所需要的数据类型。

添加与删除城市：

```
// we confirmed the operation
add_city->setText("添加城市");
del_city->setText("删除城市");
search_city_log->setReadOnly(false);
if (city_mode == ADD_OP) {
    search_city_log->setPlainText("添加城市 "+search_city_input->text()+"成功");
    cities.insert(search_city_input->text().toStdString());
    m.insert(std::make_pair(search_city_input->text().toStdString(),city_counts++));
    from_station->addItem(search_city_input->text());
    to_station->addItem(search_city_input->text());
} else if (city_mode == DEL_OP) {
    search_city_log->setPlainText("删除城市 成功");
    // to erase a city's information we need to remove it from our set and map, as well as all related edges
    cities.erase(search_city_input->text().toStdString());
    m.erase(search_city_input->text().toStdString());
    for (int i=0;i<v.size();++i) {    ⚠ comparison of integers of different signs: 'int' and 'std::__1::vector<trans, std::__1::
        if (v[i].v_start == search_city_input->text().toStdString() || v[i].v_end == search_city_input->text().toStdString()) {
            v.erase(v.begin()+i);
            --i; // balance automatic ++i
        }
    }
}
```

在进行城市添加时，除图形界面的信息更新外，我们需考虑两部分：一是在所有城市的名称集合中添加该城市（cities 的插入操作），二是为该城市建立名称与编号的映射（m 的插入操作）。在进行城市删除时，除上述两部分需逆向操作外，还要删除无用的边（即从要删除的城市出发或到达要删除城市的交通路径信息）。

添加与删除路径：

```
// confirm the operationn
if (route_mode == ADD_OP) {
    QString log = "添加中\n";
    search_route_log->setPlainText(log);

    if (tmp.start_time < 0 || tmp.end_time < 0 || (cities.find(tmp.v_start) == cities.end()) || (cities.find(tmp.v_end) == cities.e
        // we have got a wrong data base
        log = log + "添加失败，输入不合法\n";
        search_route_log->setPlainText(log);
    } else {
        log = log + "添加成功\n";
        search_route_log->setPlainText(log);
        v.push_back(tmp);
    }

} else if (route_mode == DEL_OP) {
    search_route_log->setPlainText("删除成功");
    for (int i=0;i<v.size();++i) {    ⚠ comparison of integers of different signs: 'int' and 'std::__1::vector<trans, std::__1::a…
        if (v[i].v_end == tmp.v_end && v[i].v_start == tmp.v_start && v[i].v_type == tmp.v_type && v[i].v_name == tmp.v_name && v[i
            // it should be enough to identify a unique train
            v.erase(v.begin()+i);
            --i;
        }
    }
}
```

对于某一特定路径的增删，我们需要在 v 数组中更新路径信息（erase 或 push_back），由于邻接表在每次最短路计算时都重新生成，我们并不需要在此处更新邻接表信息。

城市查找：

```
void MainWindow::start_city_search() {
    QString log = "开始搜索\n";
    search_city_log->setPlainText(log);
    if (cities.find(search_city_input->text().toStdString()) != cities.end()) {
        log = log + "查找成功，找到" + search_city_input->text() + "\n";
    } else {
        log = log + "查找失败，" + search_city_input->text() + "未找到\n";
    }
    search_city_log->setPlainText(log);

}
```

我们只需要在城市名称的集合中尝试查找即可。

路径查找：

```
void MainWindow::start_route_search() {
    QString log = "开始搜索";
    search_route_log->setPlainText(log);
    std::string from_city,to_city,train_no;
    from_city = search_route_start_input->text().toStdString();
    to_city = search_route_end_input->text().toStdString();
    train_no = search_route_num_input->text().toStdString();
    for (int i = 0;i < v.size();++i) {     ⚠comparison of integers of different signs: 'int' and 'std::__1::vector<trans, std::__1::alloc…
        // search whether the given info matches any route
        if (from_city == "" || v[i].v_start == from_city) {     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<trans…
            if (to_city == "" || v[i].v_end == to_city) {     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<trans, …
                if (train_no == "" || v[i].v_name == train_no) {     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<…
                    log = log + "搜索到车次" + v[i].v_name.c_str() + ",发车日期 " + v[i].start_day.c_str() + "," + v[i].v_start.c_str() + "始发
                }
            }
        }
    }
    log += "\n搜索完毕";
    search_route_log->setPlainText(log);
```

我们的程序支持模糊搜索，即车辆始发站点、终到站点、车次名称无需全部提供即可进行查找。为实现该功能，当输入的某类信息为空时直接忽略不作为过滤车次的条件，即代码中比较条件的||左侧表达式。

路径边权计算：

我们通过以不同方式计算边权满足不同的路径选择偏好，如果想尽快到达城市，我们将以交通时长作为边权；而如果想以较低花费到达，我们将以交通费用为边权；而如果想以较少换乘到达，我们以常数 1 为边权即可：

```
double len_judger(std::string mode_str,struct trans v) {
    /*
     * for different path choice preferences we have different weight calculation
     * if we want to reach a city as fast as possible, we will evaluate the transfer time
     * if we want to reach a city with less money, we will evaluate the transfer time
     * if we want to reach a city with less transfer time, we will focus on it
     */
    if (mode_str.find("快") != std::string::npos) {
        return v.length;
    } else if (mode_str.find("省") != std::string::npos) {
        return v.price;
    } else {
        // 少
        return 1;
    }
}
```

寻路算法核心：

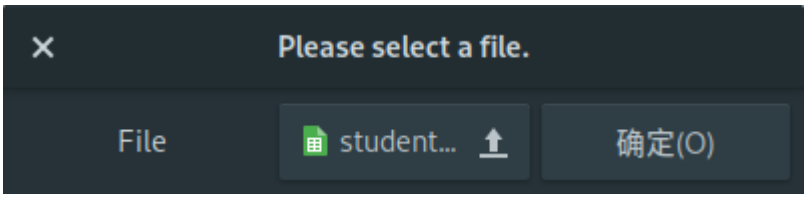
```
std::memset(reach_time,0x3f,sizeof(reach_time));
reach_time[src] = 0;
dis[src] = 0;
ins[src] = true;
int now;
s.insert(std::make_pair(0,src)); // use set to store the cities and their shortest distance
while (!s.empty()){
    now=s.begin()->second;
    s.erase(std::make_pair(dis[now],now));
    for (int i=0;i<edge[now].size();++i)     ⚠comparison of integers of different signs: 'int' and 'std::__1::vector<node, std::__1::…
        if (dis[edge[now][i].to]>dis[now]+edge[now][i].len)     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<n…
        if (reach_time[now] <= edge[now][i].beg) {     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<node, std::…
            if (ins[edge[now][i].to]){     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<node, std::__1::alloca…
                // if the destination is in our list, we do not update ins array
                s.erase(s.find(std::make_pair(dis[edge[now][i].to],edge[now][i].to)));     ⚠implicit conversion changes signedness: '…
                dis[edge[now][i].to]=dis[now]+edge[now][i].len;     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vect…
                s.insert(std::make_pair(dis[edge[now][i].to],edge[now][i].to));     ⚠implicit conversion changes signedness: 'int' to…
            }else{
                // otherwise we update ins array
                ins[edge[now][i].to]=true;     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vector<node, std::__1::al…
                dis[edge[now][i].to]=dis[now]+edge[now][i].len;     ⚠implicit conversion changes signedness: 'int' to 'std::__1::vect…
                s.insert(std::make_pair(dis[edge[now][i].to],edge[now][i].to));     ⚠implicit conversion changes signedness: 'int' to…
            }
            pre[edge[now][i].to] = edge[now][i].offset; // remember the choice, useful for the path output     ⚠implicit conversion c…
            pre_node[edge[now][i].to] = now;           // remember where the optimal choice came from     ⚠implicit conversion chang…
            reach_time[edge[now][i].to] = reach_time[now] + v[edge[now][i].offset].length; // cannot use edge's len because it is not a
        }
        ins[now]=false;                               // not in the array any more
}
```

我们采用 Dijkstra 算法配合 STL set 进行优化计算最短路。由于我们无法搭乘在到达站点前已经出发的列车，在验证最短路信息前，需先比较到达站点时间与车次出发时间，若合理才可用该车次去更新站点最短路信息。

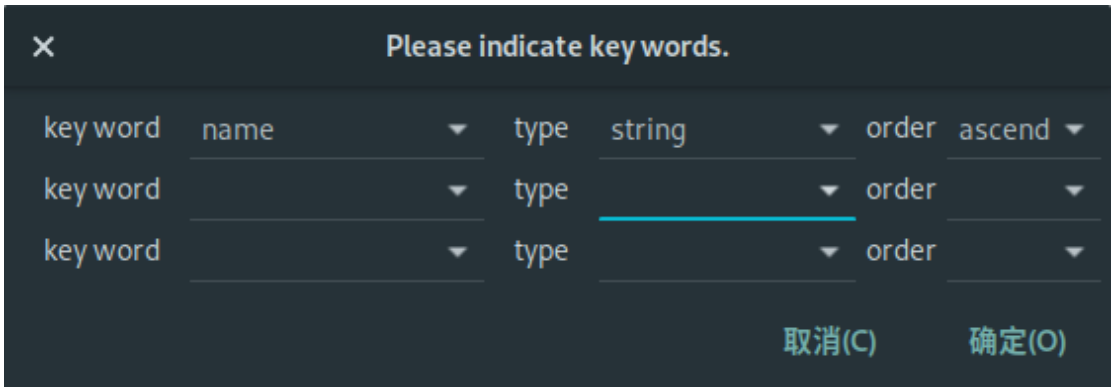
三．运行展示与分析

多关键字排序

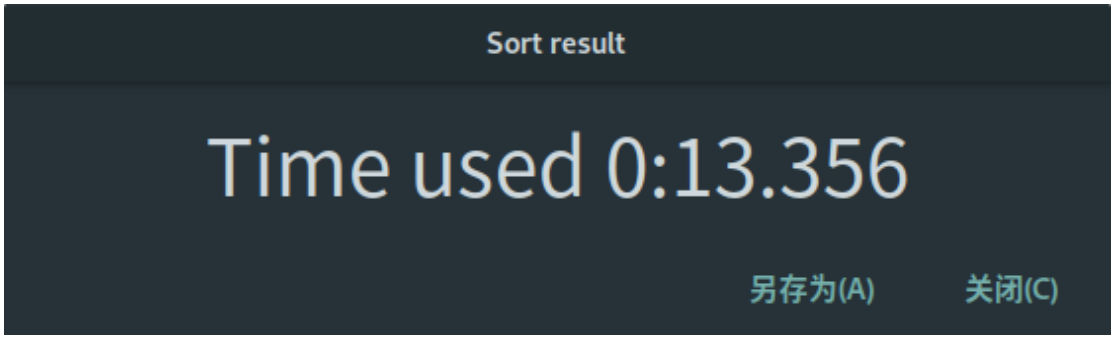
本程序需要 GTK+3.0 运行时环境。
进入程序后，将选择打开的文件



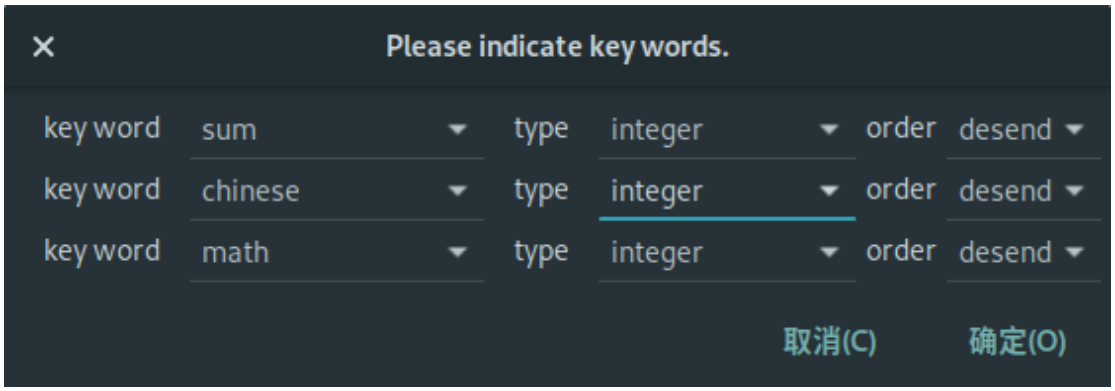
在点击确定后选择待排序的关键字，在此举出两个例子



在此例中，对一个关键字 name 进行了排序，种类为字符串，顺序为递增。点击确定后

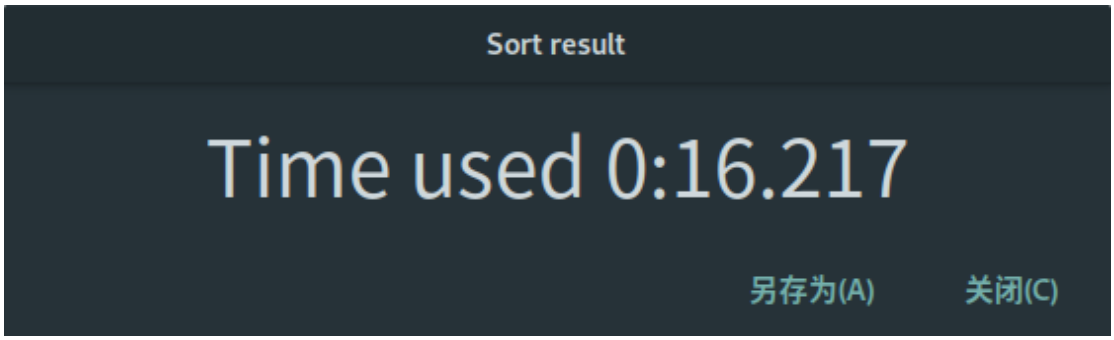


显示使用了 13.356s 完成排序，csv 文件已存于 by_name.csv
另一了例子如下：



主要关键字为 sum，次要关键字分别为 chinese 和 math，类型都为整数，顺序为递减序。

同理点击确定：



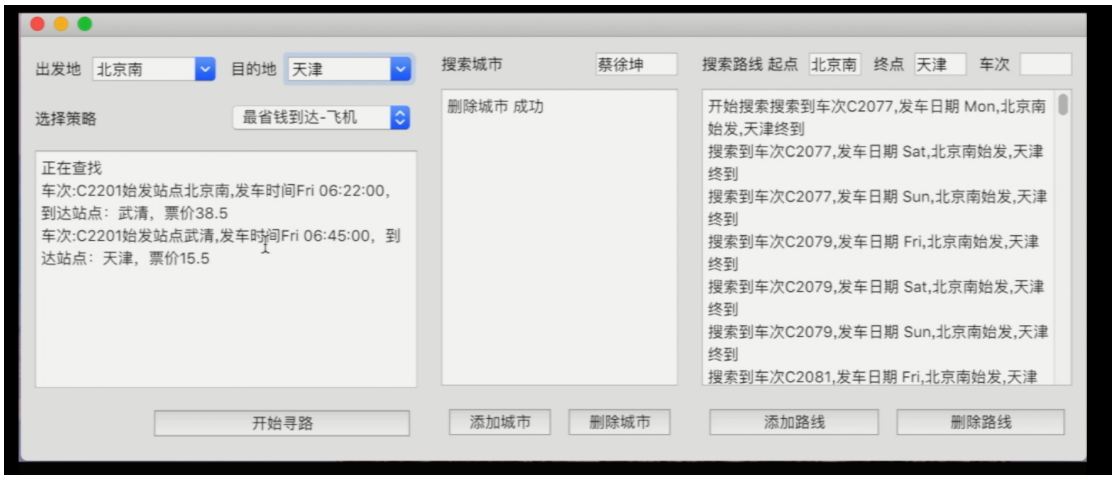
使用 16s 完成排序，内容存于 by_score.csv。

全国交通咨询模拟

添加城市：

路径（模糊）搜索

不同需求的方案提供：



完整功能展示可见于视频。

四．实验总结

本次实验中，我们实现了多关键字排序和全国交通咨询模拟程序。

多关键字排序带有美观的图形界面，实现了多关键字排序及结果的 csv 格式导出，功能完备，简洁高效。

全国交通咨询模拟程序带有方便直观的图形界面，方便易用；支持多种路径规划策略，支持城市与路径的手工增删。此外，为保证程序的真实性并检验程序效率，我们使用的数据来自真实的 12306 网站数据，规模较大，对程序的算法效率也提出了较高考验。而在实际测试中，路径的规划速度很快，按下寻路按钮后数据瞬间即可返回，说明本程序的时间复杂度控制良好。

五．分工

刘士琪：多关键字排序与 12306 爬虫实现
陈国凯：全国交通咨询模拟程序实现与其实验报告
李非冬：演讲 PPT 及实验报告多关键字排序部分

六．附录：文件说明

多关键字排序：
`src/file_select.ui` `src/keyword_select.ui` `src/sort_result.ui` 界面文件
`src/main.vala` 主文件
`src/window.vala` 核心代码文件

全国交通咨询模拟：
Mainwindow.cpp：图形界面与数据初始化
algorithm.cpp:核心算法