COVID-19疫情环境下低风险旅行模拟系统的设计

摘要：

使用 Python 爬虫从网上获取三种交通工具的航班表，建立了 14 个城市的交通网。

算法方面，第一个策略使用了模拟退火算法删除大量的点，随后使用斐波那契堆优化的 Dijkstra 算法求解最短路径；第二个策略与 k 短路问题十分相近，因此考虑使用 A* 算法求解，使用 SLF+LLL 优化的 SPFA 求出结点到终点的距离作为 A* 的启发式函数。

UI 用 Qt 设计，分为主界面、命令行界面和动画演示共三个界面，功能和样式简洁明了，其中命令行用于直接查询规划方案，动画演示界面用于实时显示旅客状态。

最后对算法的正确性、健壮性和时间效率进行了分析与验证。

关键字：斐波那契堆优化的 Dijkstra 算法；模拟退火算法；A* 算法；SLF+LLL 优化的 SPFA；算法效率分析；Qt 界面

开发环境：windows 10 系统+Qt Creator+Qt 5.12

一、任务描述

城市间有三种交通工具（汽车、火车和飞机）相连，有些城市之间无法直达，需途径中转城市。某旅客于某一时刻向系统提出旅行要求。考虑在当前 COVID-19 疫情环境下，城市的风险程度分为低风险、中风险和高风险三种。系统根据风险评估，为该旅客设计一条符合旅行策略的旅行线路并输出；系统能查询当前时刻旅客所处的地点和状态（停留城市/所在交通工具），并做日志记录。

乘客的旅行计划包括起点、终点和旅行策略，旅行策略共有两种：

1.最少风险策略：无时间限制，风险最少即可
2.限时最少风险策略：在规定的时间内风险最少

二、需求分析

需求分析：明确任务定义是什么，限制条件是什么。例如：输入/输出数据的类型、值的范围及形式。

1. 基本要求
    - 城市总数不少于 10 个，三种不同风险城市个数均不得小于 3 个，城市风险分别为 0.9、0.5、0.2；
    - 旅客在城市停留风险 = 该城市单位时间风险值 × 停留时间；
    - 城市之间不能总只是 1 班车次，整个系统中航班数 < 10，火车数 < 30，汽车无限制；
    - 建立汽车、火车和飞机的时刻表；

- 根据旅客选择的起点、终点和旅行策略进行规划；
- 旅行模拟系统以时间为轴向前推移，每 10 秒左右向前推进 1 个小时；
- 建立日志文件，对旅客状态变化和键入等信息进行记录。

2. 附加功能

- 绘制地图，并在地图上实时反映出旅客的旅行过程
- 增加乘坐交通工具的风险值，旅客交通风险 = 交通工具单位时间风险值 × 该班次起点城市的单位风险值 × 乘坐时间
- 对模拟旅行风险计划时间轴进行调控，可以加快系统推进速度
- 动态显示旅客行程完成进度及已用时间等信息

三、建立模型

1. 模型假设

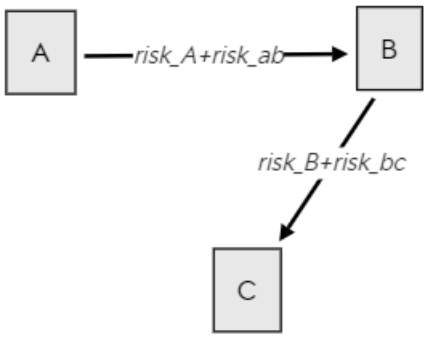
- 假设交通工具每日的时刻表固定。例如每天从北京开往西宁的“8837”号列车都会在 10:50 出发，在 23:55 到达西宁；
- 城市风险值固定，不会短时间内出现新增病例的激增或骤降，从而影响城市的风险值；
- 假设各种交通工具均为起点到终点的直达，中途无经停；
- 不考虑城市内换乘交通工具所需时间；
- 转乘时过夜无需额外的风险。

2. 建立模型

模型有两种建立方式，一种是以城市为节点进行建图，另一种是以交通为节点建图，下面分析两种方法：

- 以城市为节点建图

  如果以城市为节点建图，我们需要将城市的风险值放到边上，则此时边上的权值包括乘坐交通工具的等待风险和在交通工具上的风险值，如下图：



此时 $B \to C$ 的风险值计算公式如下：

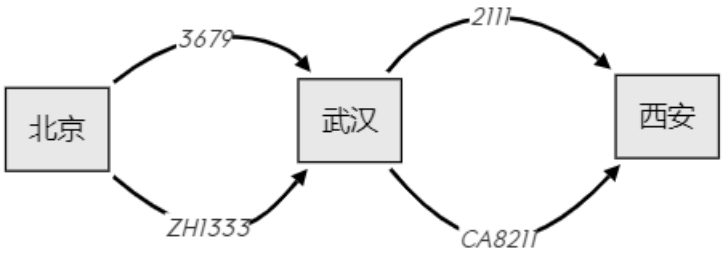
$$w = risk_B + risk_{bc}$$

$$risk_B = cityRisk[B] \times (begin_{bc} - end_{ab})$$

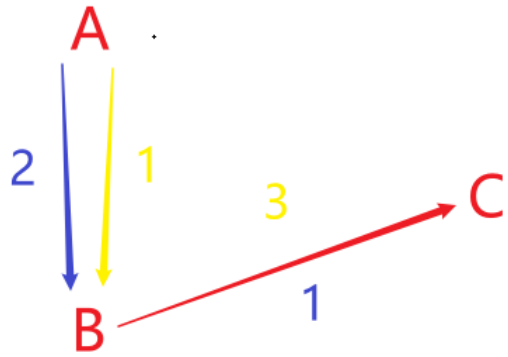
$$risk_{bc} = cityRisk[B] \times vehicleRisk[vehicle] \times (end_{bc} - begin_{ab})$$

其中，$risk_B, risk_{bc}$ 分别为城市 B 的等待风险和 $b \to c$ 的交通风险。$cityRisk[i]$ 为城市 i 的单位风险值，$vehicleRisk[j]$ 为交通工具 j 的单位风险值，$begin_{uv}, end_{uv}$ 分别为城市节点 $u \to v$ 的交通出发和到达时间。

但是这种方式图中会出现大量的重边，如下图：



并且由于问题涉及交通工具的出发和到达时间，前一步交通工具的选择会影响后一步中转城市的风险值，即影响下一步的边权。例如上图中，“2111”边上的风险值会因“3679”和“ZH1333”的不同选择而不同，因此我们在求解最短路时，无法保证得到最优解。再用一张图进行解释：



A→B 航班的选择，会影响到 B→C 同一航班的风险值，这样求得的 C 的最短路为黄色的路 $1+3=4$，而实际上的最短路为蓝色的路 $2+1=3$。因此得到的解并非最优解。

- 以交通工具为结点建图

  在经过了错误的尝试后，想到了使用交通工具为结点。假设有从交通节点 $u \to v$ 的有向边，则边权的公式为：

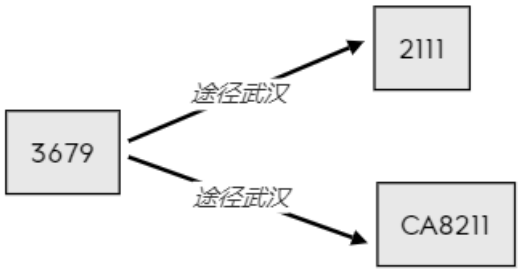
$$w = risk_u + risk_{u.to}$$

$$risk_u = cityRisk[u.to] \times vehicleRisk[u.vehicle] \times (u.end - u.begin)$$

$$risk_{u.to} = cityRisk[u.to] \times (v.begin - u.begin)$$

  其中 $risk_u, risk_{y.to}$ 分别为节点 u 的风险值（即交通工具风险）、中转城市风险值。$cityRisk[i]$ 为城市 i 的单位风险值，$vehicleRisk[j]$ 为交通工具 j 的单位风险值，$j.from, j.to, j.begin, j.end$ 分别为交通工具 j 的起点、终点、出发和到达时间。

  我们通过观察公式发现，两节点间的边权在读入交通工具时刻表时已经确定，并且点与点之间都是单边，如下图：



在这种情况下，需要一个起始**虚拟结点**，该结点的交通风险值为空，总风险值=起点城市等待风险值。

因为可能存在多个交通可以到达终点的情况，所以在求得最短路后，还需要在以到达城市为终点的交通中选一个风险最小的方案作为最终计划。

至此，就可以像处理正常的单源最短路问题一样求解本题了。

以上两种建模方法均可解决这个问题，我们采用第二种方法（以交通为节点建图）编写程序。

四、数据结构说明

- main.cpp

```
void myMessageOutput(QtMsgType type, const QMessageLogContext&
context, const QString& msg); /* 消息传递函数重定向log.txt */
void outputMessage(QtMsgType type, const QMessageLogContext&
context, const QString& msg); /* 日志重定向 */
```

- FibonacciHeap 类

```
class FibonacciHeap {
public:
    FibonacciHeap();              /* 初始化变量 */
    void initNode(int, double); /* 初始化结点 */
    void merge(int);              /* 插入结点u */
    void link(int, int);          /* 连接两结点 */
    void consolidate();           /* 确保没有根相同的度 */
    int extractMin();             /* 提取最小值 */
    void cut(int);                /* 切断关系 */
    void cutRec(int);             /* 确定关系是否应该被切断 */
    void decreaseDanger(int, int);  /* decrease key[u]到val */
private:
    std::vector<int> pa, child, left, right, deg;/* 分别为父亲、儿
子、左右子树、度 */
    std::vector<double> danger;      /* 结点的风险值 */
    std::vector<bool> vis;           /* 结点是否已被访问 */
    int minNode;                     /* 最小danger结点的索引 */
    int rootNum;                     /* 根结点数量 */
```

```
};
```

- Traveler类

```
class Traveler {        /* 乘客信息及规划路径 */
public:
    Traveler(int, QDateTime, QDateTime, int, int, int); /* 构造函数赋初值 */
    std::vector<TimeTable> getPlan();    /* 获取旅客计划 */
    QDateTime getCityArvTime(int);       /* 获取城市到达时间 */
    QDateTime getCityDptTime(int);       /* 获取城市离开时间 */
    double getMinDanger();               /* 获取最小风险值 */

    int id, strategy, source, destination;      /* 分别记录旅客序号、策略、起点城市序号、终点城市序号 */
    QDateTime startTime, deadlineTime, usedTime, totalTime; /* 分别记录旅客起始时间、限制时间、已用时间、计划总时间 */
private:
    double calDanger(std::vector<TimeTable>);   /* 计算规划路线的总风险值 */
    void Simulated_Annealing();                 /* 模拟退火算法 */
    std::vector<TimeTable> Dijkstra(std::vector<QDateTime>&, std::multimap<int, TimeTable>);
/* 斐波那契堆优化的Dijkstra算法 */
    void Spfa(int);                             /* SLF+LLL优化的SPFA算法 */
    std::vector<TimeTable> AStar(int, int);     /* 求解k短路问题的A*算法 */
    void path2Plan(int, const std::vector<TimeTable>&, std::vector<TimeTable>&);    /* 逆向求解计划 */
    void Swap(TimeTable&, TimeTable&);      /* 交换航班信息（模拟退火中使用） */
    QDateTime TotalDateTime();                  /* 计算totalTime */
    void updateTime(std::vector<QDateTime>&, std::multimap<int, TimeTable>::iterator, int, int);                /* 更新time */

    std::vector<double> danger;     /* 从起点到达结点的风险值 */
    std::vector<double> dangerIvs;  /* 航班到destination的风险值 */
    std::vector<TimeTable> plan;    /* 规划的路线 */
    std::vector<QDateTime> time;    /* 到达每个结点的时间 */
    double minDanger;               /* 最小风险值 */
};
```

- Schedule类

```
static const int cityAmount = 12;// 城市总数量
enum Vehicle { car = 1, train = 2, flight = 3 };//  枚举交通工具变量值
```

```
static std::vector<double> cityRisk = { 0.9, 0.9, 0.2, 0.9, 0.5
,0.2, 0.5, 0.2, 0.2, 0.2, 0.2, 0.5, 0.9, 0.9, 0.5 };// 城市风险值
static std::vector<double> trafficRisk = { 2.0, 5.0, 9.0 };// 交
通工具风险值

struct TimeTable {// 每个航班信息的结构体
    int from, to, vehicle;       /* 记录交通起点、终点、类型 */
    double danger;               /* 乘坐交通工具的风险 */
    QString num;                 /* 交通编号 */
    QTime begin, end;            /* 交通起始、结束时间 */
    TimeTable() { from = -1; }       /* 默认构造函数 */
    TimeTable(int _from, int _to, QString _num, QTime _begin,
QTime _end, int _vehicle) :
        from(_from), to(_to), num(_num), begin(_begin),
end(_end), vehicle(_vehicle) {// 初始化列表
        // 计算乘坐交通工具的风险
        double duration = (_end.hour() + 24 - _begin.hour()) %
24;
        danger = cityRisk[_from] * trafficRisk[_vehicle] *
duration;
    }
};

class Schedule {
public:
    Schedule();                         /* 读取文件并将数据存入
trafficMap, databaseRev, vehicleSelect */
    static int cityToNum(QString);  /* 城市字符转换为编号 */

    static std::multimap<int, TimeTable> trafficMap;// 存航班表
    static std::multimap<int, TimeTable> databaseRev;// 存逆航班表
    static std::vector<std::vector<TimeTable> > vehicleSelect;//
存每条路可能的航班

private:
    static bool isInited;    /* 保证仅读取一次文件 */
};
```

- Widget 类

```
class Widget : public QWidget {
    Q_OBJECT
public:
    explicit Widget(QWidget* parent = nullptr); /* 创建按钮、连接
信号和槽 */
    void paintEvent(QPaintEvent*);       /* 绘制背景 */
    FigWidget* figWidget = NULL;         /* 下一级页面 */
    CmdWidget* cmdWidget = NULL;         /* 下一级页面 */
    ~Widget();

private:
    std::vector<Traveler*> travelerList;     /* 旅客列表 */
    Ui::Widget* ui;                          /* ui */
};
```

- CmdWidget 类

```
class CmdWidget : public QWidget {
    Q_OBJECT
public:
    explicit CmdWidget(QWidget* parent = nullptr);  /* 创建按钮、
连接信号和槽 */
    void paintEvent(QPaintEvent*);               /* 绘制背景 */
    std::vector<QString> handleInput(QString);  /* 处理输入信息 */
    QString num2City(int);  /* 序号转换为城市字符 */
    ~CmdWidget();           /* 析构函数 */

    int addNum;          /* 添加乘客次数 */
    bool allowInput;         /* 是否可以输入 */
    std::vector<Traveler> travelerList; /* 旅客列表 */
    Schedule schedule;                  /* 读取文件 */
signals:
    void cmdWidgetBack();   /* 命令行界面返回信号 */
    void setBtnZoomOver();  /* 处理信息信号 */
private:
    Ui::CmdWidget* ui;      /* ui */
};
```

- FigWidget 类

```
class FigWidget : public QWidget {
    Q_OBJECT
signals:
    void DoStartTimer();    /* 开始计时的信号 */

public:
    explicit FigWidget(QWidget* parent = nullptr);  /* 初始化ui、
建立时间线程及连接信号和槽 */
    void initFigureUI();         /* 初始化ui界面 */
    void initFigureConnect();    /* 连接信号和槽 */
    void initTimeThread();       /* 建立时间线程 */
```

```
    QDateTime getStartTime();    /* 获取开始时间 */
    QDateTime getSpentTime();    /* 获取已用时间 */
    QDateTime getDeadline();     /* 获取截止时间 */
    int getStrategy();           /* 获取用户所选策略 */
    int getStart();              /* 获取用户所选始发地 */
    int getDestination();        /* 获取用户所选目的地 */
    void showPlan(int);          /* 传入id，打印旅客的计划 */

    void displayTotalTime();         /* 显示方案所需总时间 */
    void displayFare(std::vector<TimeTable> path);/* 显示方案所需
经费 */
    void displayPath(std::vector<TimeTable> path);/* 在pathlist窗
口中显示路径 */
    void displayCurTime();           /* 显示旅客当前时间 */
    QString num2City(int index);     /* 将城市编号转为城市名称 */
    void startButtonClicked();       /* 开始按钮按下，开始计算路径图
形输出 */
    void addTravelerButtonClicked();/* 添加乘客，按钮初始化 */
    void timeStart();                /* 计时信号 */
    void displaySpentTime();         /* 显示已经花费的时间 */

    ~FigWidget();

    Schedule schedule;                   /* 读取文件 */
    int currentTraveler;                 /* 当前旅客序号 */
    std::vector<Traveler> travelerList; /* 旅客列表 */

signals:
    void figWidgetBack();        /* 页面返回信号 */

public slots:
    void travelerChanged(); /* 切换旅客时更改界面显示 */

private:
    Ui::FigWidget* ui;  /* ui */

    int strategy, start, destination;    /* 策略、出发地、目的地 */
    int addtravelertimes;                /* 添加旅客次数，即旅客最大
编号 */
    int startclickedtimes;               /* "开始"按钮点击次数 */
    int strategyBtnIdx;                  /* 策略按钮索引 */
    std::vector<bool> startclicked;      /* "开始"按钮第一次按下 */

    //参与时间进程的变量
    QTimer* mstimer;         /* 计时器 */
    QThread* timethread;     /* 计时器线程 */
};
```

- Gif 类

```
class Gif : public QWidget
{
```

```
public:
    explicit Gif(QWidget* parent = nullptr); /* 绘制背景、建立时间线程、连接信号和槽 */
private:
    void update();                    /* 刷新画面 */
    void paintEvent(QPaintEvent*);    /* 绘图 */
    QPixmap setPointGraph();          /* 设置图标 */
    QPointF setPointPos();            /* 设置图标位置 */
    QDateTime getSplitTime(QDateTime, QDateTime);    /* 获取两时间点时间间隔 */
    QPointF getCityCor(int city);                    /* 获得城市对应坐标 */
    double getTimeDifference(QDateTime, QDateTime); /* 获得两时间间隔时间差 */
    QPointF getMoveDistance(QDateTime, QDateTime, QDateTime, int, int);/* 获得坐标增量 */
    int state;                /* 旅客状态 */
    QTimer* paintmstimer;     /* 刷新计时器 */
};
```

- MyPushButton 类

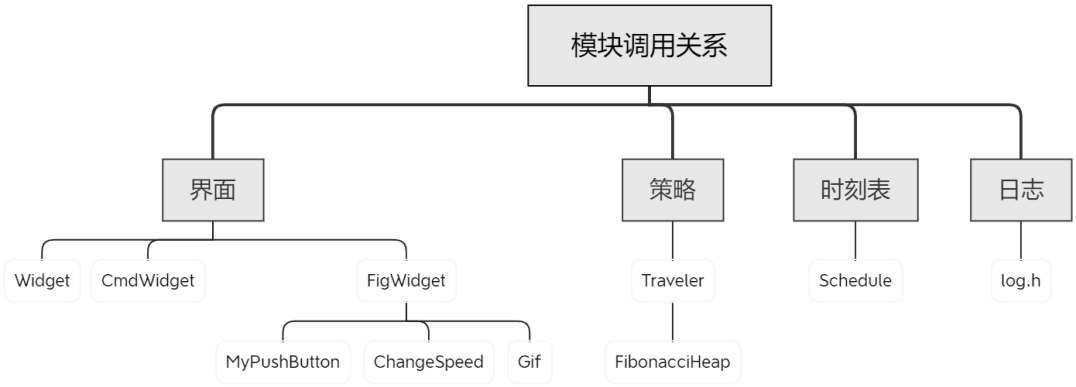
```
class MyPushButton : public QPushButton {
    Q_OBJECT
public:
    MyPushButton(QString normalImg, QString pressImg = ""); /* 加载图片 */
    void zoom1(); /* 向下跳 */
    void zoom2(); /* 向上跳 */

private:
    QString normalImgPath;  /* 默认显示图片路径 */
    QString pressedImgPath; /* 按下后显示图片路径 */
};
```

- ChangeSpeed 类

```
class ChangeSpeed : public QWidget {
    Q_OBJECT
public:
    explicit ChangeSpeed(QWidget* parent = nullptr);     /* 连接slider和spin */
    QSpinBox* spin;      /* 建立spin控件 */
    QSlider* slider;     /* 建立slider控件 */
signals:
    void speedValChange();  /* 速度值改变信号 */
};
```

- 以上各模块的调用关系如下图：



五、各模块设计说明

- `main.cpp`

  功能：使用日志文件，将程序中的 `Debug`, `Error` 等调试信息全部输出到 `log.txt` 文件中方便调试。

  实例化 `Widget` 对象，并显示主窗口。

- `FibonacciHeap` 类

  对斐波那契堆进行操作，用斐波那契堆优化 Dijkstra 算法。方法如下：

```
namespace fibonacciHeap
{
    void initNode(int u, int val) {/* 初始化结点 */ }
    void merge(int u) {/* 插入结点u */ }
    void link(int u, int v) {/* 将v设为u的孩子 */
    void consolidate() {/* 确保没有根有相同的度 */ }
    int extractMin() {/* 提取最小值并进行consolidate操作 */ }
    void cut(int u) {/* 切断父亲/孩子的关系 */ }
    void cutRec(int u) {/* 检查标记，确定u是否应该被cut */ }
    void decreaseKey(int u, int val) {/* decrease key[u]到val */
}
};
```

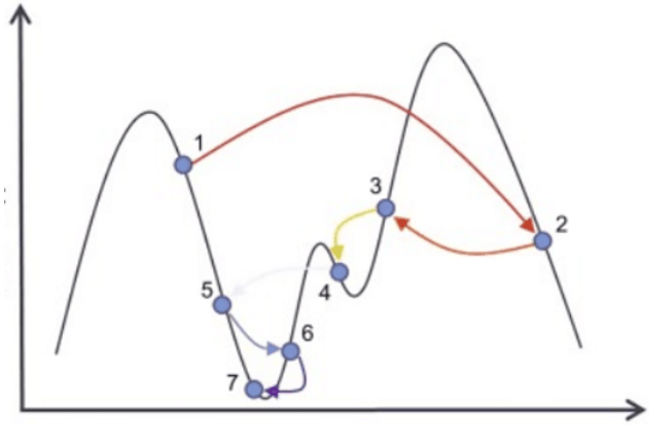
- `Traveler` 类

  1. 模拟退火算法

     （1）基本算法思想

     模拟退火算法实际上是一种贪心算法，但是在搜索的过程中引入了随机因素。模拟退火算法以一定概率接受一个比当前差的解，因此可能跳出当前的的局部最优解，从而达到全局最优解。

     接下来举例说明：



模拟算法在搜索到局部最优解 4 时，会以一定的概率接受到 5 的移动。经过几次移动后就有可能到达最优，于是便跳出了局部最大值。以下是伪代码描述：

```
delta, T0, Tk; // 指定参数
void Simulated_Annealing() { //模拟退火
    t = T0;
    while (t > Tk) {
        /* 产生新解 */
        /* 计算当前解的结果 */
        /* 计算当前解与最优解的差 */

        /* 此解更优，更新ans及新解 */
        if (deltaAns < 0) /* 更新ans及新解 */

        /* 已一定概率接受新解，不更新ans */
        else if (P * RAND_MAX > rand()) /* 更新新解 */

        else {/* 拒绝新解 */}

        t *= delta;//降温
    }
}
```

（2）具体实现

首先对用 vehicleSelect 对两城市间的交通工具进行选择，随后通过 Dijkstra 算法对已选交通工具计算最短路。以下是算法实现的伪代码：

```
void Traveler::Simulated_Annealing()
{
 double t = T0;// 初始温度
    while (t > Tk) {
        // 产生新解。随机产生下标进行交换，从而产生新序列
        Swap(Schedule::vehicleSelect[i][0],
Schedule::vehicleSelect[i][random() % items]);
        // 将vehicleSelect转换为multimap<int, TimeTable>
pathSelect
        // 计算新解风险值
        std::vector<int> nowPlan = Dijkstra(time,
pathSelect);
        double nowDanger = calDanger(nowPlan);
        //deltaDanger < 0,
```

```
            if (nowDanger < minDanger) {/* 接受新解更新minDanger
*/ }
            // 以一定概率接受新解
            else if (exp(-deltaDanger / t) * RAND_MAX > rand())
{/* 不更新minDanger */ }
            // 拒绝新解
            else {/* 恢复原序列 */ }
            t *= delta;// 降温
        }
    }
```

（3）算法效率

退火算法每次随机产生新解时，删除了很多交通工具结点，大大降低了后续 Dijkstra 求解的复杂度，提高算法效率。模拟退火算法的时间复杂度无法精确计算，在数据量非常大的时候，模拟退火+ Dijkstra 算法的结合会大大提高运行速度。

2. 斐波那契堆优化的 Dijkstra 算法

（1）基本算法思想

Dijkstra 算法使用了广度优先搜索解决单源最短路径问题，基于贪心的思想，不断选择离源点最近的点进行标记，最终得到源点到每个点的最短路径。伪代码如下：

```
  for (v = 0; v < G.vexnum; ++v) {
     min = MAX_NUM;
   for (i = 0; i < G.vexnum; ++i)
        if (!visited[i] && d[i] < min) {// 找到最近的未访问结
点
            /* 更新min和v */
        }
    visited[v] = true;// 标记已访问
    for (i = 0; i < G.vexnum; ++i)
        if (!visited[i] && min + map[v][i] < d[i])
            /* 更新d[i] */
}
```

普通 Dijkstra 算法的时间复杂度为 $O(n^2)$，当图中结点增多时，时间复杂度会呈平方式增长，这时程序效率会变得很差。为解决这种情况，采用了斐波那契堆来优化 Dijkstra 算法。普通 Dijkstra 算法在内层遍历找最近的结点时，我们可以将结点存入小根堆中，这样无需遍历内层循环便可以直接取出最近的结点，但调整堆需要一定的时间复杂度。斐波那契堆不涉及删除元素的操作有 $O(1)$ 的时间复杂度，`extract_min` 和 `delete` 的复杂度和其他堆相比，在 n 较小时效率更佳。稠密图每次 `decrease_key` 只需要 $O(1)$ 的时间复杂度，和二项堆的 $O(logn)$ 相比是巨大的改进。

斐波那契堆的关键思想在于尽量延迟堆堆的维护。当向堆中插入新结点或合并两个堆时，并不去合并树，而是将这个操作留给 `extract_min`。斐波那契堆优化的 Dijkstra 算法伪代码如下：

```
 namespace fibonacciHeap {/* 方法见FibonacciHeap类 */ };
void Dijkstra() {
    fibonacciHeap::initNode(src, dist[src]);// 初始化结点
```

```
    fibonacciHeap::merge(src);// 插入结点
    for (int i = 0; i < n - 1; ++i) {
        int u = fibonacciHeap::extractMin(); // 找到堆顶最小
结点
        for (int j = head[u]; j; j = nxt[j]) {// 松弛每条出边
            if (dist[u] + weigh[j] < dist[v]) {
                // 如果从未插入过堆
                if (dist[v] == INF) {/* 更新变量并将v插入堆 */
}
                else {/* 进行decrease_key操作 */ }
            }
        }
    }
}
```

（2）具体实现

执行算法时，先检查当前结点是否被退火算法删除。如果没删除，则对边进行松弛。通过查找 databaseRev 找出能到达终点的航班视为图中的中终点，找到风险最小的一个终点制定计划。

```
std::vector<int> Traveler::Dijkstra(std::vector<QDateTime>&
time, std::multimap<int, TimeTable> pathSelect) {
    fp.initNode(0, danger[0]);  /* 初始化新结点 */
  fp.merge(0);              /* 插入虚结点0 */
    for (int i = 0; i < trafficAmount - 1; ++i) {
      int u = fp.extractMin();
        for (unsigned j = 0; j < travelerMap[u].size(); ++i)
{
            int v = travelerMap[u][j].first;

            /* 如果v已被退火删除则直接continue */

            TimeTable e = travelerMap[u][j].second;
            if (danger[v] > danger[u] + e.danger) {
                if (danger[v] == __DBL_MAX__) {/* 初始化结点
v、插入结点v */ }
                else {/* 进行decreaseKey操作 */ }
                // 更新path, time
                path[v] = u;
                updateTime(time, it, u, v);
            }
        }
    }
    std::vector<int> plan;

    /* 找终点是destination的航班tmpMinTfc */

    path2Plan(tmpMinTfc, path, plan);// 传入终点航班
    return plan;
}
```

（3）算法复杂度

- 朴素 Dijkstra 若利用数组（或链表）存储所有顶点，时间的复杂度是 $O(n^2)$;
- 利用二项堆实现的 Dijkstra 算法可将时间复杂度优化至 $O((m+n)logn)$
- 斐波那契堆则可将时间复杂度进一步优化到 $O(m+nlogn)$

这里 m 表示边的数量，n 表示顶点数量。

3. A* 算法

（1）基本算法思想

A* 算法是一种很常用的路径查找和图形遍历算法。它有较好的性能和准确度，可以被认为是Dijkstra算法的扩展，由于借助启发函数的引导，A*算法通常拥有更好的性能。

A*算法通过下面这个函数来计算每个节点的优先级：

$$f(n) = g(n) + h(n)$$

其中：

- $f(n)$ 是结点 n 的综合优先级。当选择下一个要遍历的结点时，选取综合优先级最高的结点；
- $g(n)$ 是结点 n 距离起点的代价；
- $h(n)$ 是结点 n 距离终点的预计代价。这也是 A* 算法的启发函数。

启发函数：

- 极端情况，当启发函数 $h(n)$ 始终为 0，则 $g(n)$ 决定节点优先级，此时算法就退化成了 Dijkstra 算法。
- 如果 $h(n)$ 始终小于等于节点 n 到终点的代价，则 A* 算法保证一定能够找到最短路径。但当$h(n)$ 的值越小，算法将遍历越多的节点，导致算法越慢。
- 如果 $h(n)$ 完全等于节点 n 到终点的代价，则 A* 算法将找到最佳路径，并且速度很快。
- 如果 $h(n)$ 的值比节点 n 到终点的代价要大，则 A* 算法不能保证找到最短路径，不过此时会很快。
    - 在另外一个极端情况下，如果 $h(n)$ 相较于 $g(n)$ 大很多，则此时只有 $h(n)$ 产生效果，这也就变成了最佳优先搜索。

A* 算法对于 k 短路问题是一种很好的解决方案，A* 算法目标状态第一次被取出时即为最优解，根据这个特点，我们只需要取出第 k 个目标状态就是答案。

因为 A* 算法是 Dijkstra 算法的扩展，可以使用堆对算法进行优化。A* 算法解决 k 短路问题的伪代码如下：

```
void Astar() {
    while (!q.empty()) {
        /* 取出队首结点 */
        if (u.now == t) {    // 到达终点
            k--;
            if (!k) {/* 得到第k短路 */ }
        }
        else
            for (int i = 0; i < E2[u.now].size(); ++i) {//
松弛操作
                /* 松弛 */
```

```
                if (u.route.find(tmp) == -1)
                    /* 结点node入队列 */
            }
        }
        /* 无路径 */
    }
```

（2）具体实现

我们的策略 2（限时最小风险）可以被抽象成一个 k 短路问题。根据算法求出风险最小的方案，然后计算该方案的 `time[destination]` 是否已经超出 `deadlineTime`，若已经超出，则寻找下一个最短路。具体实现代码如下，其中 `dangerIvs` 为 SPFA 算法计算出的启发式函数（见 SPFA）：

```
std::vector<TimeTable> Traveler::AStar(int source, int
destination) {
    std::priority_queue<node> q;
 bool canArrive = false;
    q.push(node(source, 0, dangerIvs[source]));
 while (!q.empty()) {
        node u = q.top(); q.pop();
      /* 若u已被访问，则continue */
        if (u.now != destination) {
          for (unsigned i = 0; i < items; ++i, ++it) {// 对
边进行松弛
                int v = it->second.to;
              if (danger[v] > danger[u.now] + it-
>second.danger) {
                    danger[v] = danger[u.now] + it-
>second.danger;// 松弛
                    /* 结点入队列 */
                    // 更新path,time
                    path[v] = it->second;
                    updateTime(time, it, u.now, v);
                }
            }
        }
        else {/* 到达终点，判断时间是否符合要求 */ }
    }
    // 求出路径
    std::vector<TimeTable> plan;
    if (canArrive) path2Plan(destination, path, plan);
    return plan;
}
```

（3）算法复杂度

因为 A* 算法是 Dijkstra 算法的拓展，其最差情况即退化为 Dijkstra 算法，因此时间复杂度与 Dijkstra 相同。

4. SLF+LLL 优化的 SPFA 算法

（1）基本算法思想

在介绍 SPFA 算法思想之前，首先介绍 Bellman-ford 的算法思想：从 i 到 j 点最多经过 n-1 条边，因此对这 n-1 条边进行遍历，对于每条边找能更新的，即每次对当前的 m 条边进行松弛。因为该算法搜寻边过于盲目，我们可以做一些小的优化：若某次遍历没有更新数组 d，则后面也不会更新 d（因为是用当前的 d 更新 d 的其他部分），直接退出循环。以下是伪代码：

```
for (i = 1; i <= n - 1; ++i) {
    flag = 0;
    for (j = 1; j <= m; ++j)// 对m条边进行松弛
        if (d[v[j]] > d[u[j]] + w[j]) {
            flag = 1; // 小优化
            d[v[j]] = d[u[j]] + w[j];
        }
    if (!flag) break;
}
```

SPFA 算法是对 Bellman-ford 算法的改进，Bellman-ford 搜寻边时过于盲目。实际上，只有上一次数组 d 改变的点才会造成下一次 d 的改变，因此使用队列仅将队首相邻的结点入队列，防止盲目搜索。SPFA 与 BFS 十分相似，区别是 SPFA 可以重复入队列，在出队列后 vis 数组置 false，表明可以重新入队列。

同样，使用 SLF（Small Label First）思想对队列进行优化，将队列改为双端队列，对要加入队列的点 now，如果 d[now] < d[q.front()]，则将其插入到队头，否则插入到队尾。

我们还可以用 LLL（Large Label Last）继续进行优化，对每个要出队的队首元素 u，比较 d[u] 和队列中点的 d 的平均值，如果 d[u] 更大，将其弹出放到队尾，再取队首元素进行相同操作，直到队首元素的 d<= 平均值。

SPFA 算法伪代码如下：

```
deque<int>q; q.push_back(s);
sum = d[s];
 while (!q.empty()) {
    /* 取出队首结点 */
    //LLL优化
    while (q.size() * d[now] > sum) {/* 队首元素的d高于平均
值，将其放到队尾 */}
    for (int i = 0; i < E[now].size(); ++i) {
        int v = E[now][i].first;
        if (d[v] > d[now] + E[now][i].second) {// 需要松弛
            /* 松弛 */
            /* SLF优化 */
        }
    }
}
```

（2）具体实现

在本问题中，因为 A* 算法的估值尽量与正确值相似且必须 < 正确值的要求（见 A* 算法）。这里使用 SLF+LLL 优化的 SPFA 算法计算结点到终点的风险值 dangerIvs 作为后面 A* 算法的启发式函数。具体实现如下：

```
void Traveler::Spfa(int source) {
    std::deque<int> q;
    q.push_back(source);
    dangerSum += dangerIvs[source];
    while (!q.empty()) {
     int now = q.front();
        //LLL优化
     while (dangerIvs[now] * q.size() > dangerSum) {
            q.pop_front(); q.push_back(now);
         now = q.front();
        }
        q.pop_front();
        vis[now] = false;
        dangerSum -= dangerIvs[now];
        // 对边进行松弛
        for (unsigned i = 0; i < items; ++i, ++it) {
            int v = it->second.from;
          if (dangerIvs[v] > dangerIvs[now] + it-
>second.danger) {
                /* 松弛 */
              /* 若v已经遍历，则continue */
                //SLF优化
              if (!q.empty() && dangerIvs[v] <
dangerIvs[q.front()]) q.push_front(v);
                else q.push_back(v);
            }
        }
    }
}
```

（3）算法复杂度

- Bellman-ford：适用于权值有负值的图的单源最短路径，时间复杂度O(mn)；
- SPFA：时间复杂度 $O(km)$，k 为每个节点进入队列的次数，且一般 $k \leq 2$【1】。

这里 m 表示边的数量，n 表示顶点数量。

- `Schedule` 类

  功能：读取文件中的航班表，并将其以不同形式存储在 `trafficMap`, `databaseRev`, `vehicleSelect` 中，便于后续使用。

- `Widget` 类

  功能：绘制主界面。

- `CmdWidget` 类

  功能：绘制命令行界面。

- `FigWidget` 类

  功能：绘制动画演示界面。

- `Gif` 类

  功能：绘制动画延时界面中旅客在地图上的移动。

- `MyPushButton` 类

功能：绘制不规则按钮。

- `ChangeSpeed` 类

  功能：绘制加速控件并实现相应加速功能。

六、样例测试

1. 样例的设计

   通过 Python 爬虫爬取“铁友网”上的航班信息，并根据题目要求及算法需求对爬取信息进行了筛选及格式更改。随后根据城市的地理位置和疫情风险值设置了 14 个城市，其中高、中、低风险城市数量各为 4、4、6，下图为交通线路分布示意图（样例见文档 test.txt）：

2. 算法正确性及效率测试

   - 正确性测试

     首先我们通过暴力搜索的方法求出正确答案，然后将我们的策略与暴力搜索进行对比，验证方案的正确性。

     - 策略一（风险最小）

       模拟退火算法在数据量很小的情况下是正确的，Dijkstra 算法证明在上学期详细进行过学习，在此不做证明。

       测试数据集：城市数：30，交通工具数：580

       随机选择 5 个起点终点，用暴力搜索得到答案 1，随后再用策略一的算法测试得到答案 2。我们得到的答案相同，证明了算法的正确性。

     - 策略二（限时风险最小）

SPFA 算法的证明见【1】。A* 算法的证明与 Dijkstra 算法相近，我们可以进行不严格的证明。设估价皆<=实际，若先搜到一个点，其目前经过+估值最小，而实际上并非如此。我们可以看到，在 A* 的搜索中，越往后搜越接近实际情况。所以，预估小的点先进队后，会拓展其邻近节点，然后一会儿之后，队列中大部分都是这个点的拓展点了。但是，很久之后，当那个并非最短路经过的点渐渐真实（该点实际与预估相差较大），其预估+目前就会被其他点超过，而变得不再优先，排在队尾等。所以一个点如果非最优解，就不会一直拓展下去，所以先到终点的必为最优解。

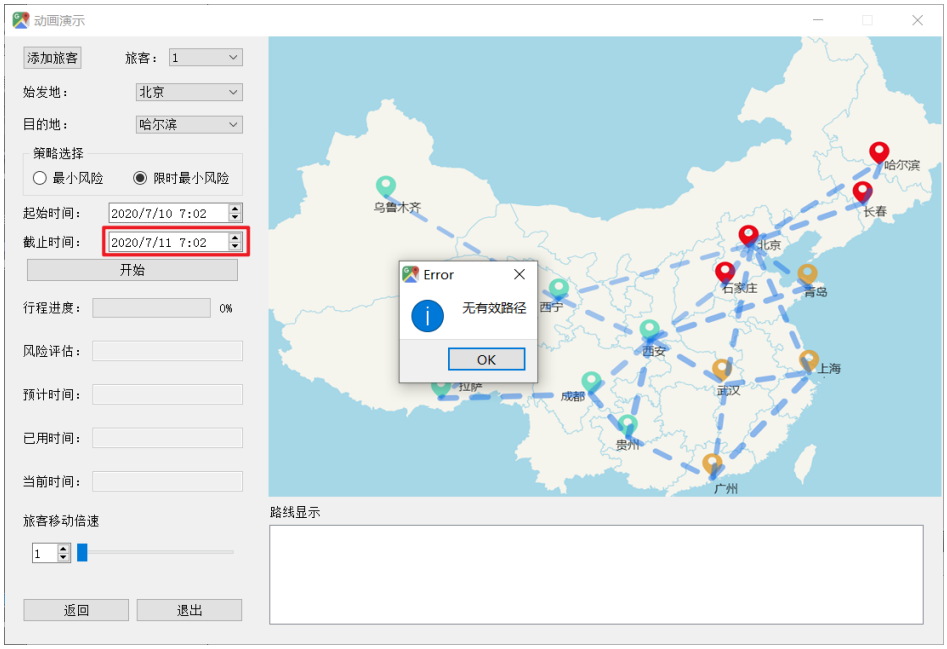
测试数据集：城市数：20，交通工具数：230

修改暴力搜索方法，如果当前路线已经超过了时间限制，直接剪枝。随机选择 5 个起点和终点。设置时间限制为 2 天，在当前时间限制内找到一个可行解，比较该方法与相同时间限制下的暴力搜索得到的结果，得到一致的答案。证明了 A* 算法在当前环境下的正确性。

- 健壮性测试：
  - 当输入城市名称不合法时，报错：

  - 当规划前没有清空时，报错：

- 当选择策略 2（有限时间最小风险）时，截止时间才会有效。此时若截止时间内无航班，则报错：



  - 时间性能测试
    - 关于时间复杂度的说明见各模块设计说明。

3. 界面测试

在命令行界面中按回显字符格式输入样例旅客信息后，右边打印旅客行程计划（详细使用方法见用户文档）：

测试样例：青岛→西宁 策略 1（最小风险） 起始时间 截止时间（选策略 1 时有效）

格式化输入：

```
Qingdao Xining 1 2020-07-09 14:58:00 2020-07-15 21:00:00
```

打印结果见下图：

在图形化界面添加乘客并选择信息后开始规划路线，左下方显示旅客实时信息，右上方实时显示旅客移动过程，右下方显示规划路线：

七、评价和改进

1. 建立模型：

- 在分析问题构思实现策略时，我起初下意识用城市当作结点建图，但在写算法的时候发现这样做有很多问题，比如图出现了重边的现象，需要进行处理，并且这样操作使得规划前因无法确定每一个城市的到达时间，前一步求解出的最短路会影响到下一个最短路的求解，只能求出近似最优解；
- 最后决定以交通为结点进行建图，从而使得交通到交通之间的风险值在求解前已知，将动态风险问题转换为已知风险进行最短路规划的问题。

2. 算法：

- SPFA 算法【1】

  SPFA 算法的复杂度为 $O(e)$，而 Dijkstra 的算法复杂度为 $O(n^2)$，当 $e << n^2$ 时，SPFA 算法体现出时间复杂度上的优越。文献【1】中已证明，对于单源最短路径算法，最小的时间复杂度是 $O(e)$，即要“查完”所有边，因此这个时间复杂度是期望的最好方法。

- 模拟退火算法【2】

  传统的模拟退火算法中，是否接受当前状态的解由 Metropolis 准则判定，公式如下：

$$p = min[1, \frac{1}{1 + exp(\Delta E/T)}]$$

  文章提出一种改进的模拟退火算法，通过改变接受当前状态解的准则，加快了其运算时间并增强了算法的收敛性。因时间原因无法编程实现，只能理论上对其进行研究学习。

- A* 算法

  A* 算法的最差时间复杂度为 $O(n^2)$，我们可以用可持久化可并堆优化来降低时间复杂度。使用可持久化可并堆优化合并一个结点与其在 T 上的祖先的信息，每次将一个结点与其在 T 上的父亲合并。这样在求出一个结点对应的堆时，无需复制结点且之后其父亲结点对应的堆仍然可以正常访问。因为时间原因，无法进行具体实现，只能进行理论的学习。算法的伪代码如下：

```
struct Edge {/* 链式前向星 */} e1, e2;
struct node {/* 定义节点 */} a;
priority_queue<node> Q;
void dfs(int x) {/* DFS搜索边集 */}
struct LeftistTree {// 左偏树结构
  LeftistTree() {/* 初始化操作 */}
  int newnode(node w) {/* 增加结点w */}
  int merge(int x, int y) {/* 合并结点x,y */}
} st;
void dfs2(int x) {/* DFS搜索左偏树 */}
void AStar(){
  Q.push({t, 0});
  while (!Q.empty()) {
    /* 取队首 */
    /* 判断是否访问 */
    /* 更新dis */
    /* 所有边集入队列 */
  }
  dfs(t);
  for (int i = 1; i <= n; i++)
    if (tf[i]) /* 遍历边集，插入结点 */
  dfs2(t);
  if (st.rt[s]) /* rt[s]入队列 */
```

```
    while (!Q.empty()) {
      /* 取队首 */
      if (st.lc[a.x]) /* lc入队列 */
      if (st.rc[a.x]) /* rc入队列 */
      /* x入队列 */
    }
  }
```

算法复杂度：

- 时间复杂度为 $O(nlogm)$，空间复杂度为 $O((n+k)logm)$。

3. 界面：
    - 界面制作精心，简洁明了，又不失生动活泼；
    - 根据旅客计划实时移动的逻辑处理较好，界面中旅客能够流畅地进行移动；
    - 能够完美地完成查询旅客计划，实时显示旅客状态等一切要求操作。

八、用户使用说明

- 见使用说明.pdf

九、开发过程记录

- 见软件开发过程.pdf

十、附录

[1]段凡丁.关于最短路径的SPFA快速算法[J].西南交通大学学报,1994(02):207-212.

[2]冯玉蓉,朱均燕.模拟退火算法收敛性的研究[J].福建电脑,2006(12):46-47.