数据结构课程设计

战疫旅人

技术文档

凌国瀚-2018213344

2020-7-1

目录

文档说明

本文档为项目 **《战疫旅人——COVID-19 疫情环境下低风险旅行模拟系统的设计》** 的技术说明文档，主要说明了整个项目的设计思想、数据结构、核心算法等技术问题，以便于老师和技术开发人员快速了解该项目并帮助后期的项目维护与功能删减。若欲了解该软件如何使用、如何配置软件运行环境以及错误说明，请查看*./document/用户文档.pdf*。

设计任务描述

城市之间有各种交通工具（汽车、火车和飞机）相连，有些城市之间无法直达，需要途径中转城市。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，各个城市的风险程度不一样，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），具体功能需求为：

a. 城市总数不少于 10 个，为不同城市设置不同的单位时间风险值：低风险城市为 0.2；中风险城市为 0.5；高风险城市为 0.9。各种不同的风险城市分布要比较均匀，个数均不得小于 3 个。旅客在某城市停留风险计算公式为：旅客在某城市停留的风险=该城市单位时间风险值*停留时间。
b. 建立汽车、火车和飞机的时刻表（航班表），假设各种交通工具均为起点到终点的直达，中途无经停。
c. 不能太简单，城市之间不能总只是 1 班车次；
d. 整个系统中航班数不得超过 10 个，火车不得超过 30 列次；汽车班次无限制；
e. 旅客的要求包括：起点、终点和选择的低风险旅行策略。其中，低风险旅行策略包括：
    a) 最少风险策略：无时间限制，风险最少即可
    b) 限时最少风险策略：在规定的时间内风险最少
f. 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(非查询状态的请求不计时，即：有鼠标和键盘输入时系统不计时)；
g. 不考虑城市内换乘交通工具所需时间
h. 系统时间精确到小时
i. 建立日志文件，对旅客状态变化和键入等信息进行记录
j. 用图形绘制地图，并在地图上实时反映出旅客的旅行过程。
k. 为不同交通工具设置不同单位时间风险值，交通工具单位时间风险值分别为：汽车=2；火车=5；飞机=9。**旅客乘坐某班次交通工具的风险 = 该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。**将乘坐交通工具的风险考虑进来，实现前述最少风险策略和限时风险最少策略。

功能需求说明及其分析

城市

首先需要确定城市的数量及其属性。关于城市数量，本项目直接设置为 ***10*** 个 **（北京 上海 深圳 成都 昆明 杭州 重庆 西安 大连 武汉）**。仅仅设置 10 个城市的原因并不是因为无法支持更多的城市数量，而是城市数量的增多将会使得使用更复杂并且没有实际意义。举例来说，当城市数量为 1000 个的时候，可视化地图界面对用户来说就太过繁杂了，并且由于飞机和火车的数量限制，汽车将在很多路径大幅重复使用，此时低风险旅行策略算法带来的优势将会被一定程度上稀释。为了体现低风险旅行策略的可行性，本项目最后仍然选择了 10 座城市的数量。然而开发人员如果仍然想要添加更多可达城市，请参考城市的添加部分。

显然，利用面向对象编程思想，可以封装一个“城市”抽象类应用于所有城市，将功能需求中各种相关变量封装为属性。每个城市的需要的属性有风险值、交通工具信息、时间信息等。详细说明请参见数据结构部分。

交通工具

时刻表

对于交通工具的时刻表，本项目没有设置复杂的随机时刻表，而是拥有一定的规则，但是也能保证程序正常运行完成目标要求。实际上，现实中的交通工具时刻表也是按照复杂且确定的规则设定的。时刻表的具体设置如下：

```
所有飞机：每 8 小时一班
所有火车：每 4 小时一班
所有汽车：每 2 小时一班

一天从 0 时开始计算，系统时间从 2020 年 1 月 1 日 0 时开始
```

乘坐用时

本项目模拟中国国内实际的 10 个城市间的旅行过程，因此城市间距离均为实际地理距离（精确到百位）。那么乘坐某个交通工具的用时是使用 **（路程/交通工具速度），结果向上取整** 这一公式计算得出的。设定三种交通工具的速度如下：

```
飞机速度：800km/s
火车速度：300km/s
汽车速度：100km/s
```

城市间的可达性及其可用交通工具

哪些城市之间可达？如果可达，可以使用哪些交通工具？下表展示了在本项目中城市之间的可达性和可用的交通工具。在本项目中，各个城市组成了一个联通图，即从任意一个城市出发，各个城市都能保证可达。

表例：

1：代表飞机，共10 班次

2：代表火车，共17 班次

3：代表汽车，共20 班次

	序号	0	1	2	3	4	5	6	7	8	9
序号	城市	北京	上海	深圳	成都	昆明	杭州	重庆	西安	大连	武汉
0	北京	-	1, 3	1, 2		2	2, 3	1	2, 3	2, 3	3
1	上海	1, 3	-	1		1, 2, 3	2, 3		3		2
2	深圳	1, 2	1	-	1	3	3		1	1	2, 3
3	成都			1	-	2, 3			2, 3		3
4	昆明	2	1, 2, 3	3	2, 3	-		2, 3		1	2
5	杭州	2, 3	2, 3	3			-	3		1	2, 3
6	重庆	1				2, 3	3	-	2, 3		3
7	西安	2, 3	3	1	2, 3			2, 3	-		3
8	大连	2, 3		1		1	1			-	2
9	武汉	3	2	2, 3	3	2	2, 3	3	3	2	-

交通工具的风险

额外需求中要求添加交通工具造成的风险以实现更加完备的旅行策略。本项目将在每一个城市对象实例中使用 K_MinRisk()方法实现该要求，该方法将会综合停留时间、城市风险、交通工具风险等因素计算出出发城市到目的城市的风险（详见数据结构）。

GUI（图形用户界面）

为了更好的实现图形界面和部分多线程操作，本项目采用基于 .Net Framework 4.6.1 平台的 C#语言编写，利用 WinForm 技术实现 GUI 和时间轴推移等功能。

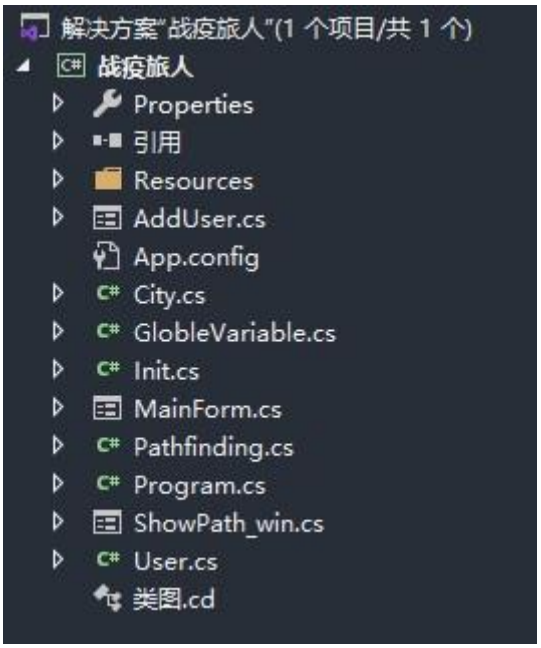
日志功能

通过全局变量确定了日志的默认保存地址，与可执行文件在同一目录；同时 GUI 界面也提供了更改日志文件保存路径的功能。本项目没有设置专用日志模块，而是在关键过程中利用 *using* 代码块功能插入日志处理程序。

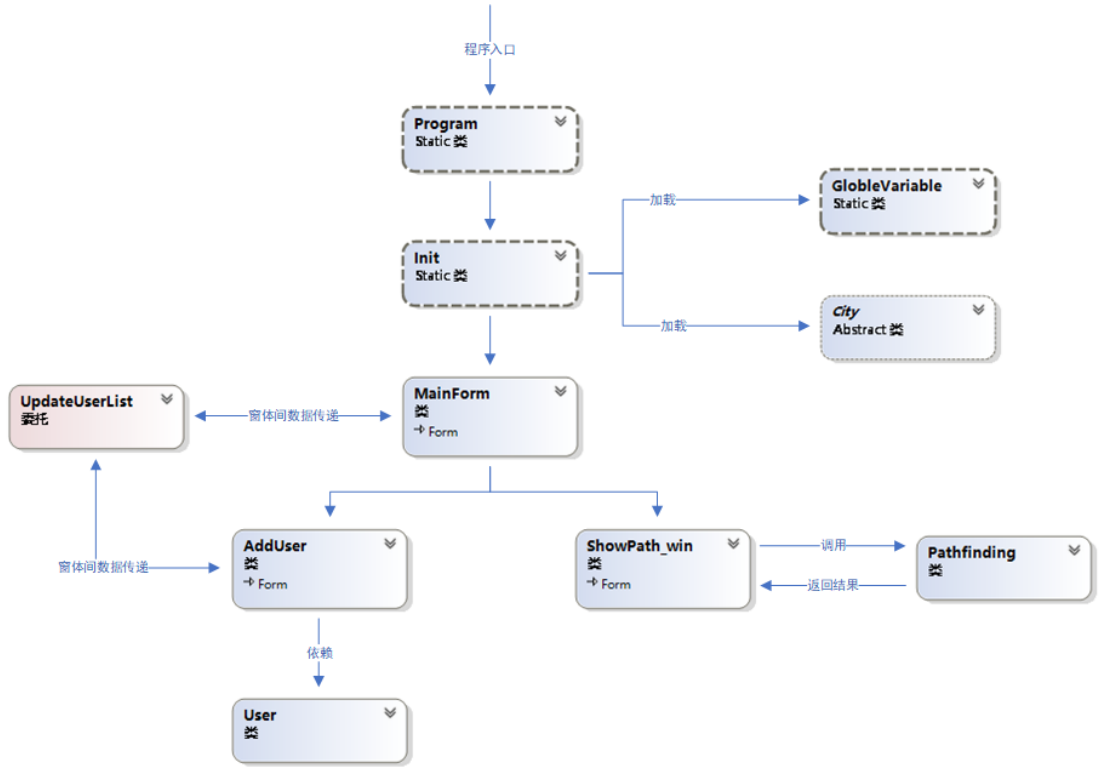
软件开发环境

请注意，本项目基于 Windows 10 的.Net 平台开发，使用 C#编程语言，无法在 Liunx 和 Mac OS 等操作系统上运行。在运行本项目时，请确保计算机已经装有 **.Net Framework 4.6.1** 环境，否则，请依照*./document/用户文档.pdf* 中的说明进行安装。本项目开发 IDE 为 **Microsoft Visual Studio Community 2019**，请于该 IDE 中打开本项目的解决方案文件*./战疫旅人.sln* 以查看和调试代码。

程序结构与模块划分



解决方案中文件如上图，它们的关系如下流程图：



委托（delegate）是在 JAVA、C#等完全面向对象的语言中重要的可复用编程方法。在本项目中，委托的用法较为简单，其主要目的为将一个函数作为参数通过委托在窗体之间传递，从而实现不同窗体之间的数据交换。

接下来将详细介绍各个模块，但若要查看模块内函数和参数的具体功能，请参见源代码内注释，或参考数据结构。

静态类

Program 类

该类作用为 WinForm 环境和初始化变量的加载，是程序的主入口，即 main 函数所在的模块。

Init 类

该类为程序初始化静态类，初始化了一些必要的变量和参数。

GlobleVariable 类

该类为公共静态类，即该类的成员作用均类似于全局变量。该类定义了本项目全部重要

的变量参数，并提供两个重载的 **CreatCity()** 方法。该类将贯穿本项目所有过程，其内容的详细介绍详见数据结构。

WinForm 窗体类

这些类涉及到 GUI 的使用，关于如何使用 GUI，请参考*./document/用户文档.pdf*。

.Net 平台的 WinForm 窗体类的特点是将设计器自动生成的代码和程序员编写的代码通过 *partial* 关键字组成两个部分类。在 Visual Studio 2019 中的解决方案资源管理器只能看到该窗体的演示界面和设计器自动生成的代码文件如 *MainForm.Designer.cs*。**若要查看具体事件方法和窗体属性配置，请通过*单击.cs 文件 - 右键 - 查看代码（F7）*的方式。**

MainForm 类

该类为本项目 WinForm 的 GUI 部分的主窗口。主要实现主窗口的用户交互功能、地图路径绘画功能、全局时钟周期功能、委托的定义等。每个事件函数都添加了必要的描述注释。

AddUser 类

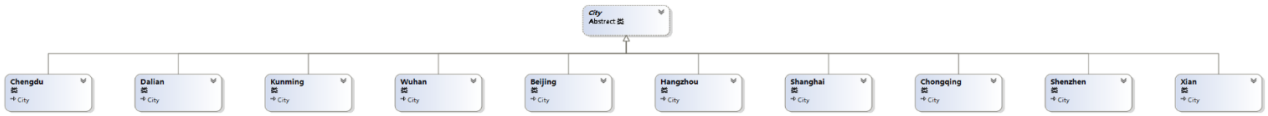
顾名思义，该窗体为添加用户时的交互界面。用户需要在该界面输入始发城市、目的城市、旅行策略、旅行时间限制等参数。在该窗体中，本项目额外添加了始发交通工具选择的功能，即在始发站用户可以指定特定的交通工具出发。若无此需求，选择系统默认即可。

ShowPath_win 类

该类为本项目较为复杂的窗体类。在 ShowPath_win 类中，首先调用 PathFinding 类的寻路算法找到给用户的推荐路径，再将路径数据整合到 User 对象实例中并在 GUI 显示。与此同时，该界面也可以用于在旅行时查看用户的详细状态，并提供删除用户信息的功能。

其他的类

City 类家族



City 类是一个抽象类，将城市概念抽象化，其 10 个子类即本项目设置的 10 个城市。City 类提供了一个城市对象全部的属性参数，包括但不限于风险值计算、等待交通工具发车时间、可达性向量、最短距离计算、最小风险城市估计等。

User 类

用户类抽象化了用户对象，封装了用户对象的旅程起止位置、出发到达时间、路径选择和交通工具选择等属性。

PathFinding 类

该类为本项目的核心算法类。对于最小风险策略，采用了以风险值为权重的 Dijkstra 算法计算用户旅行路径；对于限时最小风险策略，在 Dijkstra 算法基础上使用了 Yen 的 K 条最短路径算法（KSP），计算风险第 k 小路径，若不符合时间需求，则计算第 k+1 小路径，直到时间符合要求为止。若要求的限制时间过短，没有一条路径能够在限定时间内完成旅行，则返回 null。具体操作原理和分析见核心算法及其具体分析。

数据结构

本部分将通过源文件内注释配合相关部分代码，具体解释变量和函数的数据结构，以不同*.cs* 文件为划分，其中，*Program.cs* 文件将略过。

Init.cs

```
/// <summary>
/// 初始化调用函数
/// </summary>
    public static void Init_def()
```

该函数是主程序完成初始化的调用入口。

```
/// <summary>
/// 初始化城市映射字典
/// </summary>
public static void InitCityMapping()

/// <summary>
/// 初始化交通工具映射字典
/// </summary>
public static void InitVehicleMapping()

/// <summary>
/// 初始化城市风险表，低风险0.2，中风险0.5，高风险0.9
/// </summary>
public static void InitRisks()
```

GlobleVariable.cs

```
/// <summary>
/// 用于窗体互联的委托
/// </summary>
/// <param name="index">删除的用户ID</param>
public delegate void UpdateUserList(int index);
```

该委托作为参数在各个窗体的构造函数中传递，主要目的是在增加/删除用户后保证主窗口的用户列表能够同步正确显示。

```
/// <summary>
/// 系统时间
/// </summary>
    public static DateTime dt;
```

DateTime 是.Net 标准库中封装的时间类

```
/// <summary>
/// 日志保存路径
/// </summary>
public static string logPath;

/// <summary>
/// 城市数量
/// </summary>
public const int cityNumber = 10;

/// <summary>
/// 定义无限为double/int 数据类型的最大值
/// </summary>
public const double INF = double.MaxValue;
public const int INF_int = int.MaxValue;

/// <summary>
/// 各个城市控件在主窗体的坐标
/// </summary>
public static int[,] cityPos = { { 951, 240 }, { 1027, 360 }, { 938, 48
5 }, { 816, 373 }, { 794, 447 },{ 1005, 373 }, { 838, 389 }, { 874, 322
 }, { 1027, 221 }, { 938, 373 } };

/// <summary>
/// 城市映射字典
/// </summary>
public static string[] cityMapping = new string[cityNumber];

/// <summary>
```

```
/// 交通工具映射字典
/// </summary>
public static string[] vehicleMapping = new string[4];

/// <summary>
/// 城市风险值表
/// </summary>
public static double[] risks = new double[cityNumber];

/// <summary>
/// 飞机的速度
/// </summary>
public const double planeSpeed = 800.0;

/// <summary>
/// 火车的速度
/// </summary>
public const double trainSpeed = 300.0;

/// <summary>
/// 汽车的速度
/// </summary>
public const double carSpeed = 100.0;

/// <summary>
/// 飞机的风险参数
/// </summary>
public const int planeRisk = 9;

/// <summary>
/// 火车的风险参数
/// </summary>
public const int trainRisk = 5;

/// <summary>
/// 汽车的风险参数
/// </summary>
public const int carRisk = 2;

/// <summary>
/// 在一天中，飞机每8小时起飞一班
/// </summary>
public const int planeCycleTime = 8;
```

```
/// <summary>
/// 在一天中，火车每4小时发车一班
/// </summary>
public const int trainCycleTime = 4;

/// <summary>
/// 在一天中，汽车每2小时发车一班
/// </summary>
public const int carCycleTime = 2;

/// <summary>
/// 利用泛型存储用户数据
/// </summary>
    public static List<User> users = new List<User>();
```

```
/// <summary>
/// 通过城市代码创建城市对象实例
/// </summary>
/// <param name="cityCode">城市代码</param>
/// <param name="curTime">当前时间</param>
/// <param name="originVehicle">始发城市时所在的交通工具<para>1：飞机；2：列车；3：汽车；0：非始发站</para></param>
/// <returns>创建的城市对象</returns>
public static City CreatCity(int cityCode, int curTime, int originVehicle)

/// <summary>
/// 通过城市代码创建城市对象实例，Yen算法专用重载
/// </summary>
/// <param name="cityCode">城市代码</param>
/// <param name="curTime">当前时间</param>
/// <param name="originVehicle">始发城市时所在的交通工具<para>1：飞机；2：列车；3：汽车；0：非始发站</para></param>
/// <param name="forbidCity">一个n行2列的矩阵，表示不可达路径，第一列为不可达路径的起始城市，第二列为不可达路径的终止城市</param>
/// <returns>创建的城市对象</returns>
public static City CreatCity(int cityCode, int curTime, int originVehicle, int[][] forbidCity)
```

在这里，创建城市函数有两个重载，第一个重载是通过城市序号调用相应城市的构造函数正常创建一个新的城市；第二个重载中包含的参数 *int[][] forbidCity* 主要用于 Yen's KSP 算法的计算，即当不是求最短路径，而是求第 K 短路径时，总有 K-1 条路径在第 K 短路径的计算中必须忽略。因此在求第 K 短路径创建城市的时候，必须根据不可达信息 *int[][] forbidCity* 更新该城市的可达性属性。

MainForm.cs

```
Graphics g; // 绘制地图上的路径的GDI+对象
private int _selectedIndex = -1;     // 当前选中的用户ID

/// <summary>
/// 构造函数
/// </summary>
public MainForm()

/// <summary>
/// 委托：UpdateUserList 的函数，作用为删除UsersList列表上的用户
/// </summary>
/// <param name="index">用户ID</param>
    private void DelUserList(int index)
/// <summary>
/// 地图加载完成事件，将把地图置于背景之上
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void PictureBox1_LoadCompleted(object sender, AsyncCompletedEve
ntArgs e)

/// <summary>
/// 单击添加用户按键事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void AddUser_Click(object sender, EventArgs e)

/// <summary>
/// 每10秒一个周期，更新主视窗
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void UpdateMainForm_Tick_1(object sender, EventArgs e)

/// <summary>
/// 主视窗为焦点时开始计时
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void MainForm_Activated(object sender, EventArgs e)
```

```
/// <summary>
/// 主视窗失焦时停止计时
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void MainForm_Deactivate(object sender, EventArgs e)

/// <summary>
/// 主视窗第一次加载的初始化
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void MainForm_Load(object sender, EventArgs e)

/// <summary>
/// 双击用户列表事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void UsersList_MouseDoubleClick(object sender, MouseEventArgs e
)

/// <summary>
/// 单击用户列事件，主要内容是路径的绘制
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void UsersList_SelectedIndexChanged(object sender, EventArgs e)

/// <summary>
/// 系统时钟周期修改
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void ChangeInterval_Click(object sender, EventArgs e)

/// <summary>
/// 系统时钟周期输入框事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void NewIntervalNumber_Click(object sender, EventArgs e)
```

```
/// <summary>
/// 改变日志文件路径
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void ChangeLogPath_Click(object sender, EventArgs e)

/*  以下均为显示城市具体信息事件 */

private void BeijingButton_Click(object sender, EventArgs e)

private void ShanghaiButton_Click(object sender, EventArgs e)

private void ShenzhenButton_Click(object sender, EventArgs e)

private void ChengduButton_Click(object sender, EventArgs e)

private void KunmingButton_Click(object sender, EventArgs e)

private void HangzhouButton_Click(object sender, EventArgs e)

private void ChongqingButton_Click(object sender, EventArgs e)

private void XianButton_Click(object sender, EventArgs e)

private void DalianButton_Click(object sender, EventArgs e)

private void WuhanButton_Click(object sender, EventArgs e)
```

AddUser.cs

```
User newOne;    // 新添加的用户对象实例
private readonly UpdateUserList _updateUserList;    // 委托在该类中传递所搭载的中间变量

/// <summary>
/// 构造函数
/// </summary>
/// <param name="uul">从调用 AddUser 窗体的父窗体传递来的委托</param>
public AddUser(UpdateUserList uul)

/// <summary>
/// 按下确认键后的事件
```

```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Accept_Click(object sender, EventArgs e)

/// <summary>
/// 按下取消键后的事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Cancel_Click(object sender, EventArgs e)

/// <summary>
/// 窗体加载时事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void AddUser_Load(object sender, EventArgs e)

/// <summary>
/// 选择始发地
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void OriginCities_cob_SelectedIndexChanged(object sender, Event
Args e)

/// <summary>
/// 选择目的地
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void DestCities_cob_SelectedIndexChanged(object sender, EventAr
gs e)

/// <summary>
/// 选择初始交通工具
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Vehicle_cob_SelectedIndexChanged(object sender, EventArgs
e)
```

```
/// <summary>
/// 选择不同策略
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Limit_rad_CheckedChanged(object sender, EventArgs e)
```

ShowPath_win.cs

```
private readonly int _recommandTime;     // 建议旅程花费时间
private readonly bool _isNewUser = false;   // 判断是否是新用户
private readonly UpdateUserList _updateUserList;    // 委托在该类中传递所搭载的中间变量

/// <summary>
/// 属性：用户 ID
/// </summary>
private int UserId { get; set; }

/// <summary>
/// 属性：限制时间更改后的输入
/// </summary>
private string ChangedLimitTime { get; set; }

/// <summary>
/// 属性：该类所处理的目标用户
/// </summary>
public User TargetUser { get; set; }

/// <summary>
/// 构造函数，其中需要判断是添加新用户后的情况还是双击 UserList 的情况
/// </summary>
/// <param name="userId">所查询的用户 ID</param>
/// <param name="uul">从调用 ShowPath_win 窗体的父窗体传递来的委托</param>
public ShowPath_win(int userId, UpdateUserList uul)

/// <summary>
/// 调用算法计算用户最短路径。在限时策略中，若无法找到符合条件的最短路径，将返回 null
/// </summary>
/// <param name="recommandTime">建议用户设定的最小时间，通常用于返回 null 的情况</param>
/// <returns>用户最短路径</returns>
private City[] FindPath(out int recommandTime)
```

```
/// <summary>
/// 在窗体显示该旅程的详情
/// </summary>
private void ShowInfo()

/// <summary>
/// 判断是否输入新的限制时间
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void ChangedLimitTimeInput_TextChanged(object sender, EventArgs
 e)

/// <summary>
/// 点击确认的事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Accept_Click(object sender, EventArgs e)

/// <summary>
/// 点击取消的事件，同时也是删除用户的事件
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Cancel_Click(object sender, EventArgs e)
```

City.cs

该部分仅展示抽象基类 City 类，各个城市的子类请参考源代码。

```
/// <summary>
/// 地图上的X 坐标
/// </summary>
public int mapPos_x;

/// <summary>
/// 地图上的Y 坐标
/// </summary>
public int mapPos_y;

/// <summary>
/// 城市类构造函数
/// </summary>
```

```
/// <param name="curTime">当前时间</param>
/// <param name="originVehicle">始发城市时所在的交通工具<para>1：飞机；2：
列车；3：汽车；0：系统自动选择</para></param>
public City(int curTime, int originVehicle)

        /// <summary>
/// 城市编号
/// </summary>
public abstract int CityCode { get; set; }

/// <summary>
/// 与其它城市的距离
/// </summary>
public abstract int[] Distance { get; set; }

/// <summary>
/// 可达性向量，表示该城市与哪些城市可达。
/// <para>0 代表不可达；</para>
/// <para>1 代表航班可达；</para>
/// <para>2 代表列车可达；</para>
/// <para>3 代表汽车可达；</para>
/// <para>4 代表航班、列车可达；</para>
/// <para>5 代表航班、汽车可达；</para>
/// <para>6 代表汽车、列车可达；</para>
/// <para>7 代表航班、汽车、列车均可达</para>
/// </summary>
public abstract int[] Reachable { get; set; }

/// <summary>
/// 风险信息
/// </summary>
public abstract double[] RiskInfo { get; set; }

/// <summary>
/// 时间信息
/// </summary>
public abstract int[] TimeInfo { get; set; }

/// <summary>
/// 交通工具信息
/// </summary>
public abstract int[] VehicleInfo { get; set; }

/// <summary>
```

```
/// 城市自身的风险值
/// </summary>
public abstract double Risk { get; set; }

/// <summary>
/// 城市的名称
/// </summary>
public string Name { get; set; }

/// <summary>
/// 最快上汽车时间
/// </summary>
public int MinCarTime { get; set; }

/// <summary>
/// 最快上火车时间
/// </summary>
public int MinTrainTime { get; set; }

/// <summary>
/// 最快上飞机时间
/// </summary>
public int MinPlaneTime { get; set; }

/// <summary>
/// 始发城市所乘交通工具
/// </summary>
public int OriginVehicle { get; set; }

/// <summary>
/// 当前时间
/// </summary>
public int CurTime { get; set; }

/// <summary>
/// 更新Info系列属性，由子类负责重写，本质上是再次调用K_MinRisk()方法
/// </summary>
public abstract void UpdateInfo();


/// <summary>
/// 寻找数组最小值
/// <para>若没有最小值，返回-1</para>
/// </summary>
```

```
/// <param name="arr">目标数组</param>
/// <param name="minValue">最小值</param>
/// <returns>最小值下标</returns>
public int GetMin(double[] arr, out double minValue)
```

除此之外，需要着重介绍 City 抽象类中的一个重要方法 *K_MinRisk()*，其接口和参数介绍如下：

```
/// <summary>
/// 计算在最 k 小风险内能到达的城市。当 k = 1 时，等价于求最小风险城市
/// <para>注意：由于时间是由 路程/速度 算出的，因此时间不满整小时时，向上取整
</para>
/// </summary>
/// <param name="k">参数值，即本函数求第 k 小风险城市，k 的最小值为 1。
</param>
/// <param name="riskInfo">所有可达城市的最小风险值</param>
/// <param name="vehicleInfo">所有可达城市最小风险值使用的交通工具</param>
/// /// <param name="timeInfo">所有可达城市最小风险值花费的时间</param>
/// <returns>目标城市</returns>
public int K_MinRisk(int k, out double[] riskInfo, out int[] vehicleInf
o, out int[] timeInfo)
```

该方法的目标是，给出给定参数 k，求出最 k 小风险能到达的城市，当 k = 1 时，等价于求最小风险城市。算法首先遍历所有城市，通过可达性向量判断其可达性和可用交通工具，分 7 种情况计算风险、时间、所用交通工具，风险计算公式如下：

*最小风险值 = 使用相应交通工具花费的最小时间 * 城市自身风险 + 交通工具风险 * 城市自身风险 * 向上取整函数(到目标城市的距离 / 交通工具速度);*

eg.

*minRiskTemp = MinPlaneTime * Risk + GlobleVariable.planeRisk * Risk * (int)Math.Ceiling(Distance[i] / GlobleVariable.planeSpeed);*

最后，筛选出最 k 小风险的目标城市，并将城市、风险值、时间、交通工具等参数通过 C#特有的 *out* 关键字执行多返回值返回。事实上，City 类中的风险、时间、交通工具等属性的初始化和更新也是依赖于这个方法。

User.cs

```
/// <summary>
/// 构造函数
/// </summary>
/// <param name="origin">始发城市</param>
/// <param name="dest">目标城市</param>
/// <param name="originTime">出发时间</param>
/// <param name="originVehicle">初始交通工具</param>
```

```
public User(int origin, int dest, DateTime originTime, int originVehicl
e)

/// <summary>
/// 判断用户是否到达
/// </summary>
public bool IsArrive { get; set; }

/// <summary>
/// 推荐用户的路径
/// </summary>
public City[] RecommandPath { get; set; }

/// <summary>
/// 起始城市
/// </summary>
public int Origin { get; set; }

/// <summary>
/// 目标城市
/// </summary>
public int Dest { get; set; }

/// <summary>
/// 用户出发时间
/// </summary>
public DateTime OriginTime { get; set; }

/// <summary>
/// 路径城市的到达时间
/// </summary>
public DateTime[] CityReachTime { get; set; }

/// <summary>
/// 下一个城市的到达时间
/// </summary>
public DateTime NextCityReachTime { get; set; }

/// <summary>
/// 停留所在城市
/// </summary>
public string Location { get; set; }

/// <summary>
```

```
/// 出发时乘坐的交通工具
/// </summary>
public int Vehicle { get; set; }

/// <summary>
/// 在城市逗留为状态0，在移动中为状态1
/// </summary>
public int State { get; set; } = 0;

/// <summary>
/// 是否采用限时风险最小策略
/// </summary>
public bool LimitTimeStrategy { get; set; }

/// <summary>
/// 限时风险最小策略中的时间限制
/// </summary>
public int LimitTime { get; set; }
```

PathFinding.cs

该模块数据结构详解可参考核心算法及其具体分析。

维护提示：*请注意，Dijkstra_Def()函数的接口和 Yen()函数的接口是不同的*

```
private readonly int _origin;   // 起始城市序号
private readonly int _curTime;  // 当前时间（小时）
private readonly int _originVehicle;    // 起始交通工具
private readonly int _destination;  // 终点城市序号

/// <summary>
/// Dijkstra 类构造函数
/// <para>Dijkstra 在本项目中仅计算风险最短路径</para>
/// </summary>
/// <param name="origin">起始站点</param>
/// /// <param name="destination">目标站点</param>
/// <param name="curTime">当前时间</param>
/// <param name="originVehicle">起始站点交通工具</param>
public PathFinding(int origin, int destination, int curTime, int origin
Vehicle)

/// <summary>
/// 改进版Dijkstra 算法求最小风险路径
```

```
/// </summary>
/// <param name="forbidCity">未使用Yen算法时，此参数为null即可，否则该参数表示不可到达的路径集合</param>
/// <param name="risks">到达各个城市的最大风险</param>
/// <returns>全路径矩阵</returns>
public City[][] Dijkstra_Def(int[][] forbidCity, out double[] risks)

/// <summary>
/// KSP, Yen算法求次k短风险路径
/// </summary>
/// <param name="limitTime">限制时间</param>
/// <returns>次k短风险路径的泛型集合</returns>
public List<City[]> Yen()

/// <summary>
/// 计算一个路径的风险值
/// </summary>
/// <param name="path">路径</param>
/// <returns>路径总风险值</returns>
public static double CountPathRisk(City[] path)

/// <summary>
/// 计算一个路径花费的时间
/// </summary>
/// <param name="path">路径</param>
/// <returns>路径总花费时间（小时）</returns>
public static int CountPathTime(City[] path)

/// <summary>
/// 计算路径代码，若两条路径的路径代码相同，说明是同一条路径
/// </summary>
/// <param name="shortestPath">目标路径</param>
/// <returns>路径代码</returns>
private static string PathToCode(City[] shortestPath)
```

核心算法及其具体分析

本项目的最小风险路径采用了 Dijkstra 算法计算最小风险路径，其本质思想仍然是贪心算法。然而，由于每个路径、经过城市的风险值会由于经过时间的不同而发生改变，导致无法达成最优解，因此传统 Dijkstra 算法需要进行一些改变以适应这种条件。

对于限时最小风险策略，基本思想如下：首先判断最小风险路径是否符合时间限制，若符合，则输出路径；若不符合，寻找次小路径，判断是否符合限制……以此类推，直到找到所有从源地址到目的地址的路径，若仍然没有符合时间限制，则不得不告知用户修改其最小

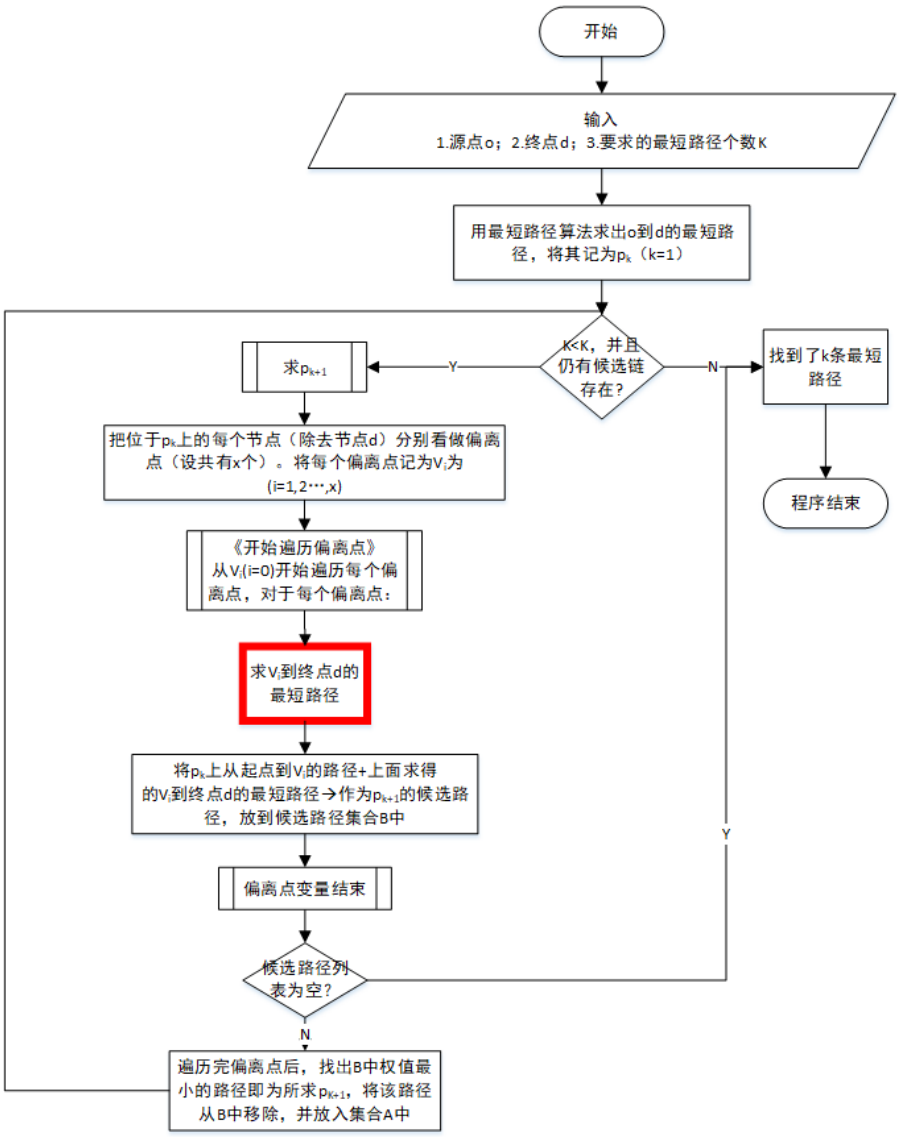
时间限制。这样以后即可成功输出限时的最小风险策略。为此，必须采用 K 条最短路径算法 KSP-Yen's Algorithm，找到次 k 小风险路径。

改进版 Dijkstra 算法

改进版本 Dijkstra 算法与未改进算法的主要区别在于，改进版 Dijkstra 算法的城市间风险值（即概念上的路径）是动态的，是基于时间变化的函数。

具体改进方式为，当寻找到路径上的目标节点（城市）的风险最小邻近城市后（此处采用 City 类中的 K_MinRisk()函数），需要更新原有路径的节点。而实际上每个路径节点就是一个 City 类实例，包含原路径中进入该城市的时间、原有风险等参数，因此在更新节点后，整个路径的风险值和时间值都会递归地发生改变。此时再次运用贪心算法原理，找到更新后的总体风险最小路径，并进入下一个循环周期。算法时间复杂度为$O(n^3)$，n为城市的总数量。

KSP 算法



K-Shortest Pathes, Yen's Algorithm 本身也是基于 Dijkstra 算法改进而来，它很大程度上稀释了 Dijkstra 算法基于贪心策略的缺点。算法可分为两部分，算出第 1 条最短路径 P(1)，然后在此基础上依次依次算出其他的 K-1 条最短路径。在求 P(i+1) 时，将 P(i)上除了终止节点外的所有节点都视为偏离节点，并计算每个偏离节点到终止节点的最短路径，再与之前的 P(i)上起始节点到偏离节点的路径拼接，构成候选路径，进而求得最短偏离路径。

本项目算法中的变量 *tempPathSet* 即所谓偏离节点集合，函数主体为一个死循环，在偏离节点数量为 0（即没有更多能到达目的地的路径）的时候退出。算法实例和其它介绍可见 Wasdns, K 条最短路径算法(KSP, k-shortest pathes)：Yen's Algorithm、Yen 的 K 条最短路径算法（KSP）。

范例测试

./document/logs 中提供部分情况的范例测试过程与结果，供读者参考。范例输出来自于日志文件。

物理距离较近范例 和 *物理距离较远范例* 是在最小风险策略下，任意交通工具出发的两个例子，分别以较近的 上海 – 杭州、较远的 大连 – 深圳 为旅行目标进行测试。二者测试结果和推算均表明，默认配置下的最小风险策略往往花费的时间也较短。

限时最小风险策略范例 中采用限时最小风险策略，以 成都 – 上海 为旅行目标进行测试。在测试开始时，输入了一个过低的时间限制，这导致用户的添加失败，并向用户表明最低要求时限，在输入满足最低要求时限的时间后，程序成功找到对应路径并完成旅行。

出发交通工具限定范例 是设定了特定的出发处交通工具，在该例中被设置为火车。因此，客户在始发地乘坐的第一个交通工具必须是火车，然后才能最终到达目的地城市。

多用户综合范例 综合了以上所有情况，设置多名不同路径、不同条件的用户实现完整全面的旅行过程

评价与改进

本项目基本实现了设计任务的全部基本要求和额外要求，能够通过综合城市和交通工具的风险为用户寻找合适的最小风险路径或限时最小风险路径，并且支持多用户输入、运行和图形化显示。运行参数表现良好

然而，本项目仍然具有改进空间，首先，有些测试人员可能希望拥有更多城市才能体现更好的测试结果；其次，本项目算法仍然有改进空间，基于贪心策略的算法或许在某些情况下无法得到最优解。

城市的添加

如果测试人员想要添加一个城市应该怎么做？以下具体列出了如何正确地在本项目中添加一个城市：

1. 在 *City.cs* 中创建 City 抽象类的子类，完善城市参数，城市序号应为原最高城市序号+1

2. 令常量 *GlobleVariable.cityNumber* 自增 1
3. 在主窗体设计界面，拖动一个按钮控件至地图中相应城市位置，并双击该控件编写城市信息事件
4. 查看该按钮控件在主窗口的坐标属性，加入常量 *GlobleVariable .cityPos*
5. 在 *GlobleVariable.CreatCity()* 中调用新城市构造函数
6. 在 *Init.cs* 中更新城市映射的名称、城市的具体风险值

至此，一个新的城市已经被添加至整个系统了。事实上，完全可以在 GUI 中实现用户添加城市的功能。然而对于一个面向有明确需求的用户的一个寻路软件，该功能太过突兀，因此最终没有对用户开放相应接口。

算法的改进

在本项目要求中，“路径”（风险）是会随时间变化的，于此同时，花费的时间也是不确定的。利用基于贪心策略的改进算法计算最小风险路径或许是可靠的，然而对于限时最小风险路径可能并非如此。因为变量相较传统图论问题更多，因此应当思考基于暴力搜索的一些优化算法。除去 KSP-Yen 算法以外，模拟退火算法就是很好的例子。

我们知道，在设定相当大的迭代数量和一定的“降温”率后，模拟退火算法一定能够得出全局最优解。在本项目的问题中，可以尝试出发城市到目标城市尽可能多的路径，并且按照一定概率（如$e^{-\frac{\Delta t}{T}}$）对路径进行扰动（如删除、增加、替换路径中城市），在温度达到设定下限或迭代数量达到上限后停止迭代，这样以后可以得到风险最小路径的最优解。如果在迭代过程中加入时间的限制，那么也能很容易得到限时风险最小路径的最优解。

由以上的分析可以看出，模拟退火算法的参数设定是非常困难的一点，例如迭代数量，唯一确定的是它必须相对城市数量来说特别大。这时候我们可以设置一个非常巨大的迭代参数，在得出最优解后对其进行二分查找，最后总能找到合适的参数值。对其它参数也能执行类似的操作。

然而和基于贪心策略的 Dijkstra 算法不同，基于暴力搜索的算法由于其不确定性，其性能无法保证，尽管可能性非常小，但在最坏的情况下其时间复杂度仍可能达到阶乘的级别。因此尽管是一个好的选项，本项目仍然选用了较为稳定的 Dijkstra+KSP 组合，而没有使用模拟退火算法。