# 数据结构与算法分析

## 2018211309 班 学号：2018211362

在《功能需求分析与总体方案设计》文档中我已经介绍了本软件的功能，满足的需求以及我的总体设计方案。在本文档中，我会针对本软件中的数据结构与算法实现进行详细分析。

## 目录

**一、地图信息的存储 - - - - - - - - - - - - - - - - - - - - - - - - P3**

1.1地图数据结构的设计思想 **- - - - - - - - - - - - - - - - - - - -** P3

1.2地图数据结构的实现- - - - - - - - - - - - - - - - - - - - - - P4

**二、旅客信息的存储 - - - - - - - - - - - - - - - - - - - - - - - - P6**

2.1旅客类的设计思想 **- - - - - - - - - - - - - - - - - - - - - -** P6

2.1.1 构造函数分析 - - - - - - - - - - - - - - - - - - - - - - P7

2.1.2 bestplan变量 - - - - - - - - - - - - - - - - - - - - - - P7

2.1.3 chosenplan变量 - - - - - - - - - - - - - - - - - - - - - P8

2.1.4 Path变量 - - - - - - - - - - - - - - - - - - - - - - - - P8

**三、核心算法分析 - - - - - - - - - - - - - - - - - - - - - - - - - P9**

3.1 FindPath类与DFS **- - - - - - - - - - - - - - - - - - - - - - -** P9

3.2旅客路径规划：贪心算法 —— person::BestPath()函数 - - - - - - P12

3.3 旅客出发时间算法：depart()函数 **- - - - - - - - - - - - - - - -** P13

**四、图形界面算法 - - - - - - - - - - - - - - - - - - - - - - - - - P15**

4.1 时钟与动画的同步 **- - - - - - - - - - - - - - - - - - - - - - -** P15

4.2旅客状态更新 - - - - - - - - - - - - - - - - - - - - - - - - - P16

4.3 动画显示 - - - - - - - - - - - - - - - - - - - - - - - - - - - P16

## 地图信息的存储

##### 1.1地图数据结构的设计思想

我在《功能需求分析与总体方案设计》4.2.2节中提到过：

“首先对于存储地图数据用邻接矩阵还是邻接表的问题，我认为，如果想支持真实时刻标，最终我们会得到一个有项的稠密图，对于稠密的图，使用邻接表是不划算的。但是我的软件又支持手动输入，也就是说，在我的这个程序中，城市数不是固定的，可以是从2至31的任意数量，那么城市数很少的时候，使用邻接矩阵就会有很大的浪费。那么，是不是可以在城市数量少的时候使用邻接表，多的时候采用邻接矩阵呢？我认为没有必要，原因是我给了用户增加城市、增加路径、增加时刻表的权力，即使当前城市数很少，我不能保证用户此后不添加数据。如果用户把一个稀疏的图硬生生添加成一个非常稠密的图，并且时刻表非常非常多的时候，如果采用这种方法，在转换的时候就会有时间，空间上的开销。此外，我认为数据被我使用一些特殊设计的数据结构被压缩的很好，无论城市数目多少，直接使用邻接矩阵在空间上的开销对于程序，对于计算机都是微不足道的。”

基于以上思想，我将地图与时刻表都存在了邻接矩阵中。

在最初实现此处的数据结构的时候，我的初步想法是把两个城市之间的旅行时间当作图中的边的权值。经过思考，我否定了这个想法，原因有二：

1、在此软件中，我们需要用到三种交通工具，两个城市之间采用不同的交通工具所花费的时间显然是不同的，按照这个思路实现数据结构的话我们就需要有三张邻接矩阵表。但是这样会增加大量重复无用的代码，因为对每一张邻接矩阵的操作都是一样的，即使用函数封装也得调用三次。

2、除此之外，两城市之间行程开销信息分开存储的话，由于图形界面的实现有很多个线程，在运行过程中如果用户修改了地图信息，三张表之间的同步很麻烦。

##### 1.2 地图数据结构的实现

综上所述，我决定把三张表合成一张表，该数据结构原理如下：

开辟一个城市数平方大小的邻接矩阵，矩阵中每一个点存储的不再是单一的交通工具通勤所花时间，而是一个我自定义的**path**类型的结构体，这个结构体封装了三个int变量：分别表示在两城市通勤时乘坐汽车，火车，飞机所花的时间，结构体定义如下：

typedef struct **pathnode**//路径

{

int BusWeight;

int TrainWeight;

int PlaneWeight;

}path;

这样只需要一张表，就可以下对应于各种交通工具的边权值。用同样的思想，在本软件中，两城市之间三种交通工具的时刻表信息同样被存储在一个邻接矩阵内，只不过实现要复杂的多。

在存储时刻表的邻接矩阵内，每个点同样是一个结构体，叫做elem（取这个名字是因为这个结构体内部还要封装别的结构体，使得数据结构很复杂，让我想起了元素周期表里的镧系和锕系，所以取名为element。），其定义如下：

typedef struct **tablenode**//时刻表

{

node\* BusNext;

node\* TrainNext;

node\* PlaneNext;

}elem;

可以看到，每个节点内部有三个**node**类型的指针，分别表示了从起始城市到目的城市每日的汽车，火车，飞机时刻表的链表头。之所以要用链表来表示时刻表，原因有三：

1. 本软件中时刻表是可以修改，增加的，用链表实现起来方便
2. 用链表实现时刻表可以在输入数据的时候不用考虑输入时刻的顺序。用户只需负责输入信息，时刻表的排序工作都会由内部算法实现。这样既方便查询时刻表的时候，呈现的信息有序化，也方便软件执行路径规划算法时查询特定的时刻表。
3. 使用链表的最大的好处就是节省空间，所有的空间被充分利用，也算是弥补一些我决定使用邻接矩阵带来的开销吧。

**node**节点的定义如下所示：

typedef struct **Tnode**

{

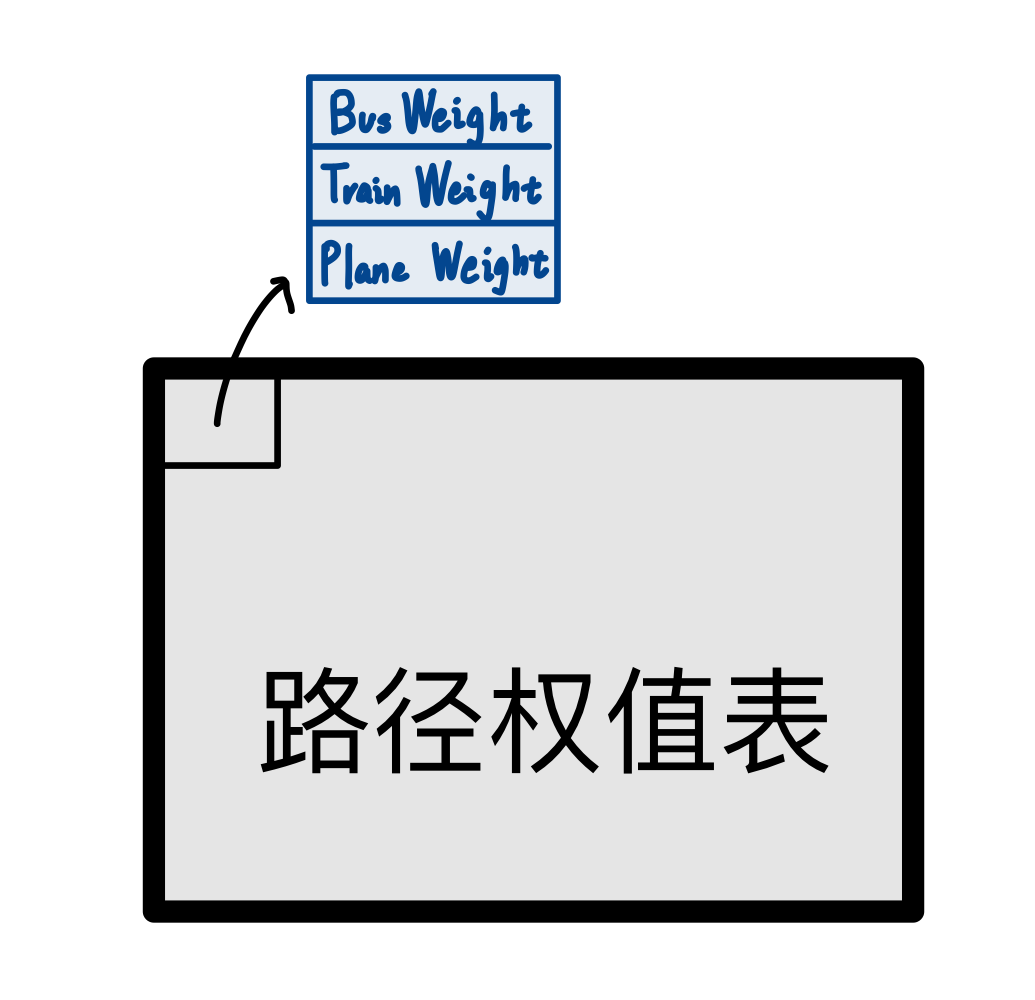
int start;

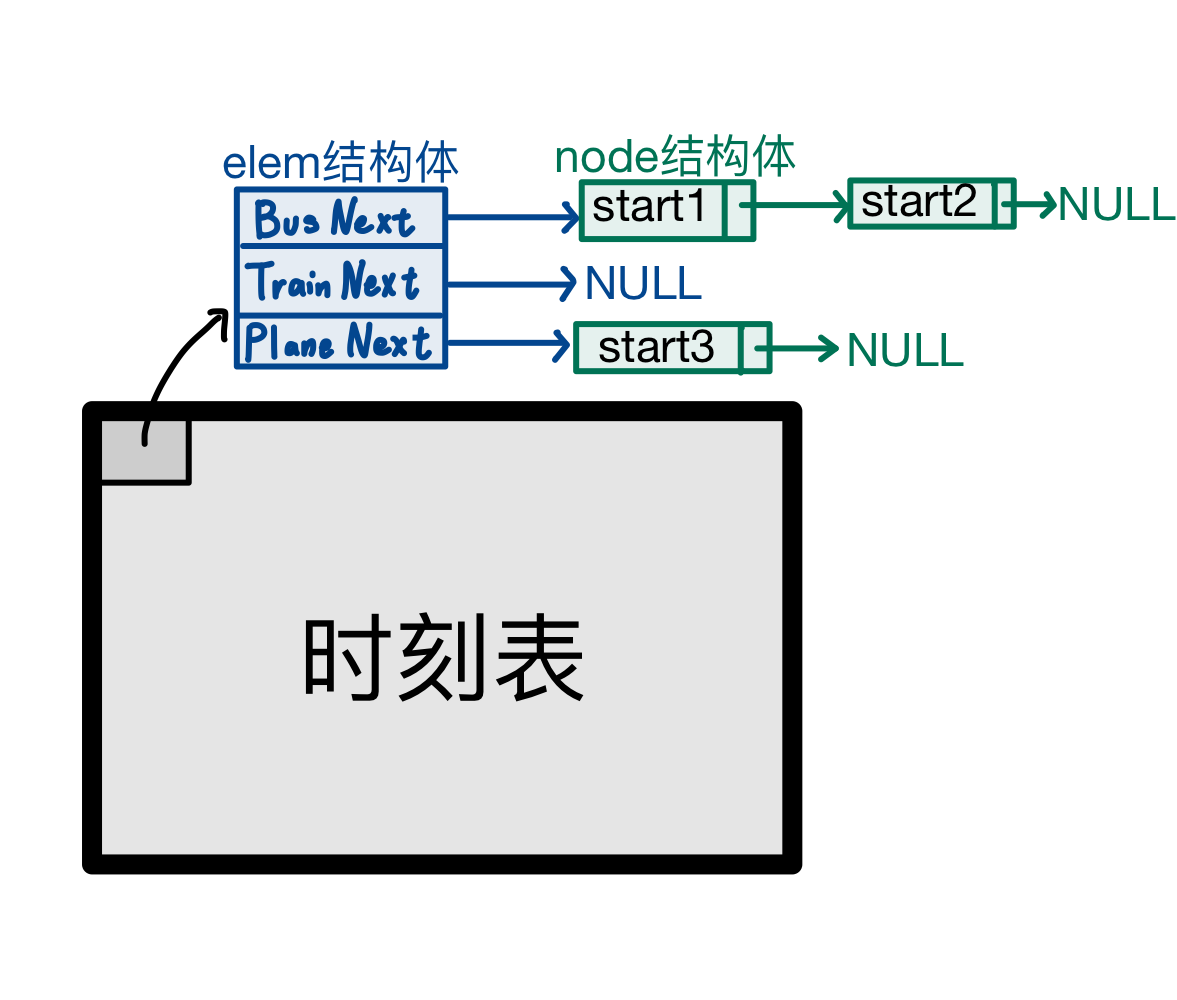
Tnode\* next;

}node;

每个**node**节点代表了一班车次（航班），其封装了一个int型变量 start表示该车次（航班）的发车时刻。

最后，我用两张图来呈现上述提到的数据结构之间的关系，相信可以更直观地呈现我的设计思想。





## 旅客信息的存储

##### 2.1旅客类的设计思想

下面给出源码中旅客类的实现。其中与数据结构相关的成员我会仔细讲述；与数据结构相关不大的成员我在此不赘述，其功能我已经在注释中写明。

class **person**

{

public:

**person**();

**person**(int num,int start,int mid,int end ,int kind,int departTime,int limit,int vexnum,double \* risklist,path\*\* Map,elem\*\* Table);

int vex; //城市数

elem\*\* table; //时刻表

path\*\* map; //地图

double\* risklist; //城市风险

QLabel\* pic; //用图片显示当前位置

QSequentialAnimationGroup \*seqGroup; //动画组

QVector<QVector<bestpath>> bestplan; //若有多个最优解

QVector<bestpath> chosenplan; //选出来的最优解

QVector<QVector<int>> Path; //初步DFS结果

int departTime; //出发时间

int pathexist = 0; //是否存在符合要求的路径

bool moreplan; //是否有多条符合要求的路

bool still = true; //旅客是否在某城市滞留

bool done = false; //旅客是否旅行结束

int loc;

private:

int end; //旅客目的城市

int mid; //旅客必经城市

int start; //旅客起始城市

int kind; //策略：1不限时，2限时

int limittime; //旅行限时

int PersonNum; //旅客编号

void **show**(int time); //限时旅客任意时刻的状态信息

int **depart**(node\* first, int now); //计算旅客在某城市何时出发

void **InitPath**(); //把地图信息传给旅客

void **BestPath**(); //旅客规划路径

};

###### 2.1.1构造函数分析

首先两个**person**构造函数。第一个构造函数传入的参数含义分别是：num：旅客序号；start：起始城市；mid：必经城市；end：目的城市；kind：限时策略/不限时策略；departTime：旅客动身时间；limit：旅行限时；vexnum：地图上城市数；risklist：存储各城市风险值的数组；Map：边权值邻接矩阵；Table:全部时刻表。第二个构造函数是给QT中的QVector初始化使用的。

###### 2.1.2 QVector<QVector<bestpath>> bestplan

这是一个存储bestpath类型结构体的二维容器，其功能是存储实现旅客旅行计划的所有符合要求的路线，下面给出bestpath结构体的定义：

typedef struct **Tbestpath**

{

int curcity;

int nextcity;

int staytime; //在cur滞留的时间

int arrivetime; //到达next时间

int nextcost; //下一次出发所需的行程

char transport; //离开cur的交通工具

double risk; //在当前城市的风险累计值

}bestpath;

我从这个结构体的定义开始，说明一下**QVector<QVector<bestpath>>** **bestplan**为什么要定义成一个二维的容器。

在**bestpath**中，**curcity**意为current city，即当前城市；**nextcity**表示旅行计划中下一个城市; **staytime**表示在当前城市等待交通工具停留的时间; **arrivetime**表示到达下一个城市的时刻; **nextcost**表示从当前城市出发到下一城市所需时间; **transport**表示到下一个城市乘坐的交通工具; **risk**表示当前累计风险值。

说到这里，bestpath的功能已经很明显了，每一个bestpath类型的数据表示旅客整个旅行计划中的一个子计划，可以存储从某城市到下一城市的所有数据。所以，一条符合旅客要求的规划路径就是由许多个bestpath节点组成的，即**QVector<bestpath>**，由于可能有多条路径都符合旅客的要求，那么就需要将多个路径封装在一起，便于操作。至此我们弄清楚了**QVector<QVector<bestpath>> bestplan**的作用。

之所以选择嵌套两层容器是因为每条路径不定长，每个旅客有几条可选路径不确定，使用vector充分利用空间，操作简便。而且，这些我使用了容器的地方并不是核心数据结构，只是暂存一下数据，我就无需自己造轮子了。

###### 2.1.3 QVector<bestpath>>chosenplan

基于上面的讨论，用户会选出多条符合要求的路径，程序会把用户所选的路径储存为chosenplan。

###### 2.1.4 QVector < QVector <int>> Path

首先给出得到Path的实现，它是通过person类中的InitPath()得到的：

void person::**InitPath**()

{

QVector<QVector<int>> path(vex);

if(this->map[start - 1][end - 1].BusWeight != -1 || this->map[start - 1][end - 1].TrainWeight != -1 || this->map[start - 1][end - 1].PlaneWeight != -1)

{

path[start - 1].push\_back(end - 1);//如果有直达，直接输出直达路径。

}

else//没有直达再搜索。

{

for (int i = 0; i < vex; i++)

{

for (int j = 0; j < vex; j++)

{

if (this->map[i][j].BusWeight != -1 || this->map[i][j].TrainWeight != -1 || this->map[i][j].PlaneWeight != -1)

{

path[i].push\_back(j);

}

}

}

}

Path = FindPath().allPathsSource(*path*,start,end);//所有能到达终点的路径

}

在这里需要说明的是，我在这里将起始城市与终点城市的直达路径直接当作了解输出。我承认这样在某些极端情况下（主要指手动输入的非真实数据）会造成解不是最优，但是我使用的数据是全国31个城市的110条互联路径，共有四百多个交通工具班次。在这样的情况下，经过我的测试，两城市之间的路径规划的解平均高达7000多个，这样的计算我认为开销有些大，不如将两点之间的直连路径直接设为解。况且，在使用真实时刻时，两地之间的直达线路就是最优的，不会出现我所说的极端情况。

同样地，这也是一个二维QVector，作用同样是存储多条路路径，与**bestplan**不同的是，它只存储路径的序号。这是因为在我的路径规划算法中，首先得到的就是这个**Path**，**Path**只是存储了所有能从源节点到目的节点的通路，可以认为是最初级的**bestplan**，将**Path**放入person类中的**BestPlan**函数，才能分析出旅客在任意时刻任意城市的状态，得到**bestplan**。

上面的代码中，**FindPath**是一个找路的class，我会在算法部分详述。

至此，与数据结构联系密切的函数与成员变量我已经大致讲述结束。下面进入核心算法部分进行详细讲解。

## 核心算法分析

##### 3.1 FindPath类与DFS

下面介绍本程序算法中的基础：给出起始城市与目的城市，找出表中所有可能的通路。

我最初的打算是直接使用Djikstra算法，但是仔细思考后，Djikstra并不能很好地满足本程序的需求。试想，给用户限定一个必经城市之后，如果必经城市不在Djikstra算法给出的路径上怎么办？把这条路排除掉然后重新执行Djikstra算法。再试想，Djikstra算法可以求出权值最小的通路，那么在此程序中权值应该是什么？如果设为滞留在城市所增加的风险值，那么在加上了考虑交通工具的乘坐风险时，风险就不只与城市风险有关了，还与两城市之间的距离，乘坐的交通工具，交通工具的发车时刻有关，这样的情况下，我认为Djistra实现起来过于复杂。

我在《功能需求分析与总体方案设计》中已经提到了我的整体设计思想之一是用空间换时间，使程序使用流畅，用Djistra算法选出一条不符合要求的路再抛弃重新选取所花费的时间太多，与我的思想相违背。所以我放弃了Djistra。

还是那个原则：“用空间换时间”，我使用了DFS算法。与其等待Djistra一次次地尝试，不如一次性把所有的通路都找出来，再把不符合的路径再删去，再按照选路策略选出最小风险或限制时间内的最小风险路径。

虽然这个算法听起来有些暴力，但是它的空间开销只是占时的，包括在第二部分提到的bestplan，一旦用户选出了一条路径，这些空间就都被释放掉了。经过我的测试，在我的电脑上，此程序最多能够支持700条直接通路。注意，不是时刻表，而是两点之间的通路，相当于把中国省会城市两两互联（汽车，火车，飞机都可以连接）。

我寻找符合要求的通路的算法为：

1. 初始化：在旅客类的**InitPath**()函数中，用旅客的起始点，终止点去查询MAP，只要两点之间有交通工具（权值不为-1）就将其压入容器中，等待处理。
2. 使用一个匿名对象（为了减少开销，用完就释放）来处理初始化过的容器，并将返回值返给旅客类的**Path**。
3. 重点来看**FindPath**的实现，它从起始点开始深度优先搜索到终止点，并每前进一个节点就将该节点入栈，达到终点时，栈中元素全部出栈压入**Path**，这样就顺序得到了一条可能的路径。
4. 为了防止程序陷入环路导致崩溃，在**dfs**(QVector<QVector<int>>& graph, int start)函数中的循环体内，我做了相应处理，每次遍历到一个节点就做相应标记，如果再次遍历到了同样的节点就说明走了一个环路，这个方向走不通，放弃。开始继续搜索其他的路。

下面贴出源代码：

void person::**InitPath**()

{

QVector<QVector<int>> path(vex);

for (int i = 0; i < vex; i++)

{

for (int j = 0; j < vex; j++)

{

if (this->map[i][j].BusWeight != -1 || this->map[i][j].TrainWeight != -1 || this->map[i][j].PlaneWeight != -1)

{

path[i].push\_back(j);

}

}

}

Path = FindPath().allPathsSource(*path*,start,end);//所有能到达终点的路径

}

class **FindPath**

{

public:

QVector<QVector<int>> **allPathsSource**(QVector<QVector<int>>& graph, int start, int dest)

{

end = dest - 1;

dfs(*graph*, start - 1);

return res;

}

private:

int end; //终点

QVector<int> way;

QVector<QVector<int>> res;

void **dfs**(QVector<QVector<int>>& graph, int start)

{

way.push\_back(start); //将当前节点加入路径

if (start == end) //是终点则将当前路径加入结果

{

res.push\_back(way);

way.pop\_back(); //记得要把刚加入的节点删掉

return;

}

for(int next : graph[start]) //遍历下一可达节点(未出现过path中的)

{

if (std::find(way.begin(), way.end(), next) != way.end())

{

continue;

}

else

dfs(*graph*, next);

}

way.pop\_back(); //完成一条路径后要把它删除

}

};

3.2旅客路径规划：贪心算法 —— **person::BestPath()**函数

前一节的person::InitPath()函数完成了所有通路的初始筛选，接下来需要用person::BestPath()来筛选出符合要求的路，大体的思想是使用了贪心算法，由于这一函数代码较长，我在此不贴出代码，直接介绍其工作流程与我的设计思想。

1. 逐一处理之前通过person::InitPath()函数已经初始化的Path中的每一条通路。
2. 对一条通路中的每一步旅行执行以下操作（以X城市到Y城市为例）：
3. 检查X到Y的乘坐各种交通工具开销的权值BusWeight，TrainWeight，PlaneWeight是否为-1（-1即不存在交通工具通勤），若不为-1，则将该权值**distance**（即乘坐时间）加上从刚到达X城市至乘坐交通工具离开的时间：**staytime**（这段时间如何计算我在下一小节阐述），即得到从刚到达X至刚到达Y的全部时间TimeTotal，如果把每一小段的TimeTotal相加，就会得到从起始城市到终点城市的总时间开销，这对于限时旅行策略是必要的。
4. 由于我实现了选做部分的“计算交通工具风险”，所以在每一小段的行程中，旅客增加的风险值公式如下，其中**risklist**是旅客类的成员，存储各城市风险值，而**TransportRisk**由交通工具决定，汽车为2，火车为5，飞机为9：

**risk = (distance \* TransportRisk + staytime) \* risklist[X]；**

通过此公式别计算出从X乘坐可选的交通工具到Y增加的风险，使用贪心算法，选择风险增加最小的那一个交通工具。至此，这一小段行程规划完毕。此外还需要做的一件事是检查XY中是否有必经城市，如果有，进行标记。如果Path已经遍历完毕，执行步骤（5），否则回到步骤（1）继续执行。

1. 释放Path的空间，检测标记，筛掉所有没有标记的规划，在剩余的

路径中根据策略选择出最优解。如果只有一个最优解，保存，释放bestplan。如果有多个最优解，给用户弹出输入框，让用户输入决定选择哪一个解，之后再释放bestplan。至此最优路径已经被规划了出来。

3.3 旅客出发时间算法：**depart()函数**

在上一节中，我提到了一个**staytime**的变量，决定旅客在旅途中从一个城市到另一个城市如何确定乘坐哪班车出发。我使用person类内部的depart()函数来计算。下面我先给出代码并加以注释，再对应着代码详细介绍我的计算出发时间算法。

int person::**depart**(node\* first, int now)

{

now = now % 24;

node\* p = first->next; //时刻表上第一个车次，不用判断p是否为空

node\* head = p; //运行到这个函数里的话，p至少是有一个车次的

if (p->start < now)//当前时间比第一个始发时间大，只能等今天下一班

{

if (p->next)//如果时刻表上有下一班车

{

p = p->next;//从下一班车开始查表

while (p)

{

if (p->start < now )

{

p = p->next;

}

else if (p->start >= now)//有一班车比现在时刻晚或相等

{

return p->start - now;

}

}

//运行到这还没return，说明当前时刻比今天所有车次发车时间

//都晚只能等明天第一班车了

return head->start + 24 - now ;

}

else//如果今天没有下一班车了

{

return p->start + 24 - now;//坐明天第一班车的等待时间

}

}

else if (p->start == now)//现在刚好有车出发

{

return 0;

}

else if (p->start > now)//第一班车比现在时刻晚

{

return p->start - now;//等车时间

}

return p->start + 24 - now;//等待比较稳妥

}

1. 该函数的返回值是旅客在X城市预计的出发时间与当前时间的差，传入的是当前模拟系统内的时间与交通工具类型。
2. 在计算时，遍历table[X][Y]中对应的交通工具类型的时刻表。该算法能顺序遍历时刻表能给出最优解是基于时刻表由链表实现，创建时可以顺序插入，插入算法很基础没什么难度，在此不介绍。
3. 从时刻表的链表头开始遍历，比较当前系统时间与交通工具始发时间，有以下三种情况：
4. 当前时刻与表头记录的交通工具始发时间相等，直接搭乘这一班离开当前城市。
5. 如果当前时刻比表头记录的交通工具首发时间晚，那么从表头开始顺序遍历，一直找到始发时间大等于当前时刻的记录节点，选中这一节点所记录的班次，计算需等待的时间差，作为返回值返回。如果一直遍历到表尾，当前时间比各节点记录的时间都晚，说明今天已经没有通勤的交通工具了，只能等到明天乘坐第一班，选中头节点，计算时间差（注意要跨天）作为返回值返回。
6. 如果当前时刻比表头记录的交通工具首发时间早，那么直接乘坐第一班交通工具（也就是表头节点所记录的那一班）出发就可以了。

## 图形界面算法

##### 4.1 时钟与动画的同步

在图形界面上，时钟显示与动画同步是很重要的。在最初的图形界面版本中，我使用了QT自带的QTimer计时器类，但是我发现这样存在一个问题：当点击查询键次数足够多时，时钟显示与动画的进度之间会有越来越大的误差。我举个例子来说明导致这一现象的原因：比如我设置系统中的时间每过一秒增加一小时，那么计时器超时时间应为1000毫秒，每超时一次系统时间加一。若当我点击按钮暂停了动画与计时器时，计时器已经记录了999毫秒，这999毫秒因为计时器被暂停的缘故会被清0，此时再当我返回运行界面，恢复计时器和动画时，动画不受影响，接着暂停时的状态继续运行，但是计时器就会比动画慢了999毫秒。这种差异累计下去就会造成非常严重的不同步现象。

对于这一问题我采用了重写运行界面的超时事件的方法，我自己通过调用系统时间来计时实现了一个计时器，该计时器被暂停后会保存暂停前的计数，解除暂停后，可以接着之前的数值继续计数，不存在计数被清零的情况，从而解决了时钟与动画同步的问题。

##### 4.2旅客状态的更新

现在已经搞定了时钟的问题，需要解决旅客状态随时间的更新。由于我重写了超时事件，每次计时器超时都会进入处理超时事件函数中，所以，水到渠成地，我可以把旅客状态的更新放在超时事件处理函数中，算法如下：

1. 每当发送超时事件时，遍历所有的旅客的chosenplan
2. 查询旅客的chosenplan中的每一个小节点的信息，如果出发时刻与现在相等，那么就更新旅客的状态，记录当前城市与所选的交通工具风险值，以便在接下来的几个单位时间内能实时计算旅客的累计风险值。
3. 在更新的同时，状态信息输出到日志，以便未来查询。

值得一提的是，在本程序中，状态更新与动画是分离的，即动画不受旅客状态变化的控制（除了更改旅客旅行计划时），因为在旅客选出路径的时候，该旅客的动画已经被确定了，所以无需与状态更新连接起来，提升了一些效率。

##### 4.3 动画显示

在上一节我提到，旅客的动画是与运行时状态更新分离的，因为在确定路径时就已经设置好了动画。动画是如何设置的与上一节的状态更新算法类似，在选定路线后就开始预演旅客未来的状态变化，记录下在某一时刻的状态信息设置成动画，使每一段动画在预定的时间播放，这样就可以达到与旅客状态变换同步，与时钟同步的效果了。

在2.1节中的person类定义中可以看到，我使旅客类维护了一个Label来展示旅客所乘坐的交通工具，还维护了一个串行动画组seqGroup，以此实现动画的显示，我主要想说明的是为何要将这二者加到旅客类的内部，我的理由如下：

1. 保证一个旅客始终只对应一个label，每次旅客的动画都对它的类内部维护的这个label做操作而不是再生成新的label，因为那样会造成内存泄漏，生成的多个label在生成后都无法被访问，在界面上还会留下之前动画的痕迹，无法清除。
2. 之所以要给每个旅客一个穿行动画组是因为方便在遍历的时候添加旅途中各小段的动画，不然旅客要维护的动画太多，代码稍不注意也会造成内存的泄露。此外，让所有的旅客都维护一个串行动画组的好处就是可以把他们压入运行界面维护的一个并行动画组中，每次执行暂停操作的时候，只需暂停运行界面的并行动画组就可以了，不用遍历每个旅客的动画，方便快捷。