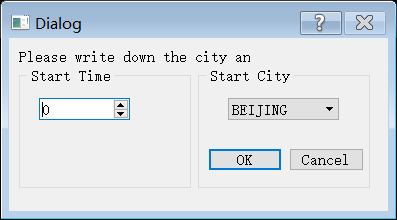
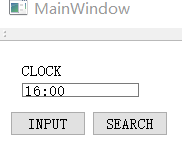
**各模块设计说明**

1. **各模块逻辑关系**
2. **初始化模块**

本模块由firstinput.cpp ，graph.cpp和mainwindow.cpp的部分功能组成。

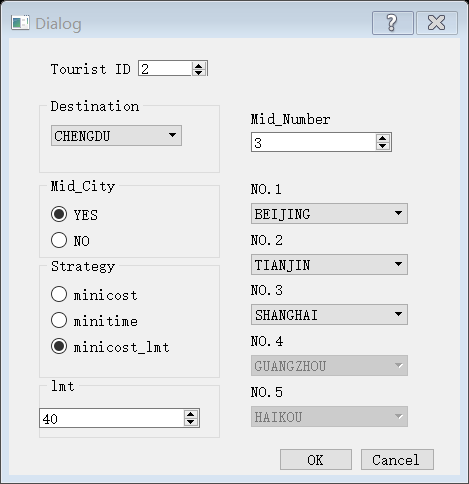
 firstinput负责初始化整个系统的初始时间和初始城市（默认情况下，所有乘客都在同一城市出发）。过程sendCity和sendTime负责向主窗口传递这两个参数，当然，如果用户拒绝初始化，直接选择close按键，那么程序会自动退出。

对于图结构，全局变量tg的初始化，这个由类Graph中的构造函数完成，即tg一旦被申请，它就完成了相应的初始化。图的初始化有两种方式，Graph();为无参构造函数，这个函数提供了早期的测试方法，通过固定的点和边进行简单的小范围的测试，适合在程序开发初期验证算法的正确性。过程Graph(QString& file\_name);是图的主要初始化手段，通过传入一个文件名参数，从文件中获取图的一切信息，当然，这个过程是高度自动化的，避免任何信息被“写死”在程序中。

其他重要内容的初始化例如乘客信息数组TST arr\_tst[MAX\_TST];，这个数组在mainwindow中通过宏时间轮询宏\_\_MUTI\_TST\_LOOP\_\_ 完成初始化。

1. **输入输出模块**

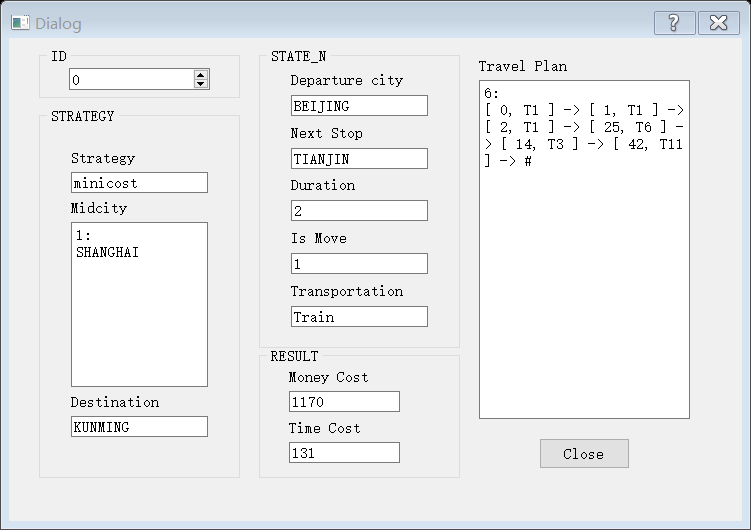
本模块主要由input.cpp负责参数的接受、传值，为算法正确执行提供参数保障。输出由mainwindow.cpp中的图形化输出和文件输出为主。search.cpp使得程序可以随时响应查询用户状态的SEARCH请求。

 INPUT按钮被置于主窗口的右上角，是一个点击即可发生的事件按钮。INPUT按钮使得在任何时候可以呼出input对话框并同时阻塞主线程的运行。

上图中给出了一个示例：程序在某时间被发出INPUT请求，这个请求的详细含义包括：需要为ID为2的用户改变旅行计划（状态），目的地置为程度，需要经过三个中间城市，按顺序分别为北京，天津，上海，旅行策略为限制时间内最少金钱，这个时间被限制在40小时。

该对话框通过void input::getValue(Graph& tg) 过程把一切收集到的信息传递给主窗口，为后续的算法和状态转移提供参数。用户可以随时改变旅行状态，发出旅行请求，这样的灵活性是状态转移逻辑模块所保证的。

对于SEARCH请求，同样该按钮呼出search对话框用来查询任何用户的状态，同时阻塞掉主窗口。请注意，search对话框不向任何窗口传递参数，它只是一个显示状态的输出对话框。对话框被设计为动态的，即可以根据输入ID的不同查看不同的用户，而不必要调用多次。



给出示例是在某一状态下发出SEARCH请求。可以看到此时ID为0号的用户的即时信息： Edge显示了用户目前在哪条边上，strategy显示用户当前旅行策略，midcity显示了需要中途途径的城市，这个例子中，empty表示没有任何需要中途途径的城市。STATE\_N是真正的状态指针，它表明：该用户目前的出发地是北京，下一个状态是天津，他目前是移动状态（从北京去往天津的路上），这个状态一共会持续2小时。RESULT给出了COST代价，是与策略选择匹配的代价输出，一个旅游策略会同时输出该路线的时间开销和金钱开销。travelpplan是边的集合，它反映了用户当前输入下需要经历的所有边，当些边都被经历过后，本次旅行过程结束。QString compCity(Graph& tg , int city);和QString getTransportation(Graph& tg, int number);过程为这个输出过程提供了必要的格式转换，参数获取等机制。

1. **核心算法模块**

本程序的核心算法被嵌入到图类的公有方法中，所有过程均在graph.cpp中。

图类中的私有方法均为公有方法服务，而公有方法均是算法核心，调用算法时，只需要调用该图的公有方法即可。

int DIJ\_MiniTime(STATE\_NOW state, int dest\_city, STML &r\_stml); // MINITIME strategy

int DIJ\_MiniCost(STATE\_NOW state, int dest\_city, STML &r\_stml); // MINICOST strategy

int MiniCost\_TimeLimt(STATE\_NOW state, int dest\_city, unsigned int limtTime, STML &r\_stml); // LIMIT stategy

int DIJ\_MiniTime(STATE\_NOW state, int dest\_city,int \* arr\_MidCity,int arr\_size, STML &r\_stml); // MIDWAY mode

int DIJ\_MiniCost(STATE\_NOW state, int dest\_city,int \* arr\_MidCity,int arr\_size, STML &r\_stml);

int MiniCost\_TimeLimt(STATE\_NOW state, int dest\_city, unsigned int limtTime, int \* arr\_MidCity, int arr\_size, STML &r\_stml);

如上面的六个过程所示，这个六个过程是核心算法接口，每个算法均需要提供必要的参数。如下给出一个在getvalue过程中调用的一个算法

arr\_tst[target].result=tg.DIJ\_MiniCost(arr\_tst[target].state\_now,arr\_tst[target].dest,arr\_tst[target].\_stml);

对于策略一即最少金钱策略，算法核心是简单图的DIJ算法。给定发出请求时的状态、目的城市即可返回一个结果和边的集合（stml）。由于旅行图可能出现重边（在两个结点之间连有若干条有向边），DIJ不可以直接使用，但是我们发现，从A点到B点边的选择，只需要选择金钱开销最小的一条边即可，这是基于这样的事实：结点序固定，那么从其中一点到另一点，开销最小的边才可能作为最后输出路线的候选。对于某一顶点D，无需给定当前状态即可判断哪条边能成为候选。可以形式化地表示为待选边e = f（c），其中c是e连接的城市组，f是算法映射。

对于第二个策略即最少时间策略，算法核心同样是简单图的DIJ算法。同策略一不同的是，不再可以用离线的DIJ算法计算，对边的筛选同样需要一些技巧。可以看到，对于某一顶点D，与它连接的边集合E的实际开销是不可被顶点D唯一确定的，还需要另外一个参数：到达D时的目前状态。也就是说，实际在这条边时间开销可以分为两个部分：等待开销和边开销。边开销显然是个只与这条边有关的定值，但等待开销需要由边出发时间和顶点到达时间联合确定。我们可以这样表示他们的关系：

t\_w = f（e.dur,state）

t\_c = e.dur

e = f（t\_w , t\_c）

其中tc表示边开销，tw是等待开销，显然地，边的实际开销是二者之和。在知道边的实际开销后，后续过程与算法一类似。

对于第三个策略，我们选择用BFS+剪枝的方法搜索解空间。显然这样的路径需要被剪掉：到达不了目的城市的，到达了目的城市但权重过大的，在搜索过程中总时间开销已经超过限制时间的。后两者可以用代码表述为

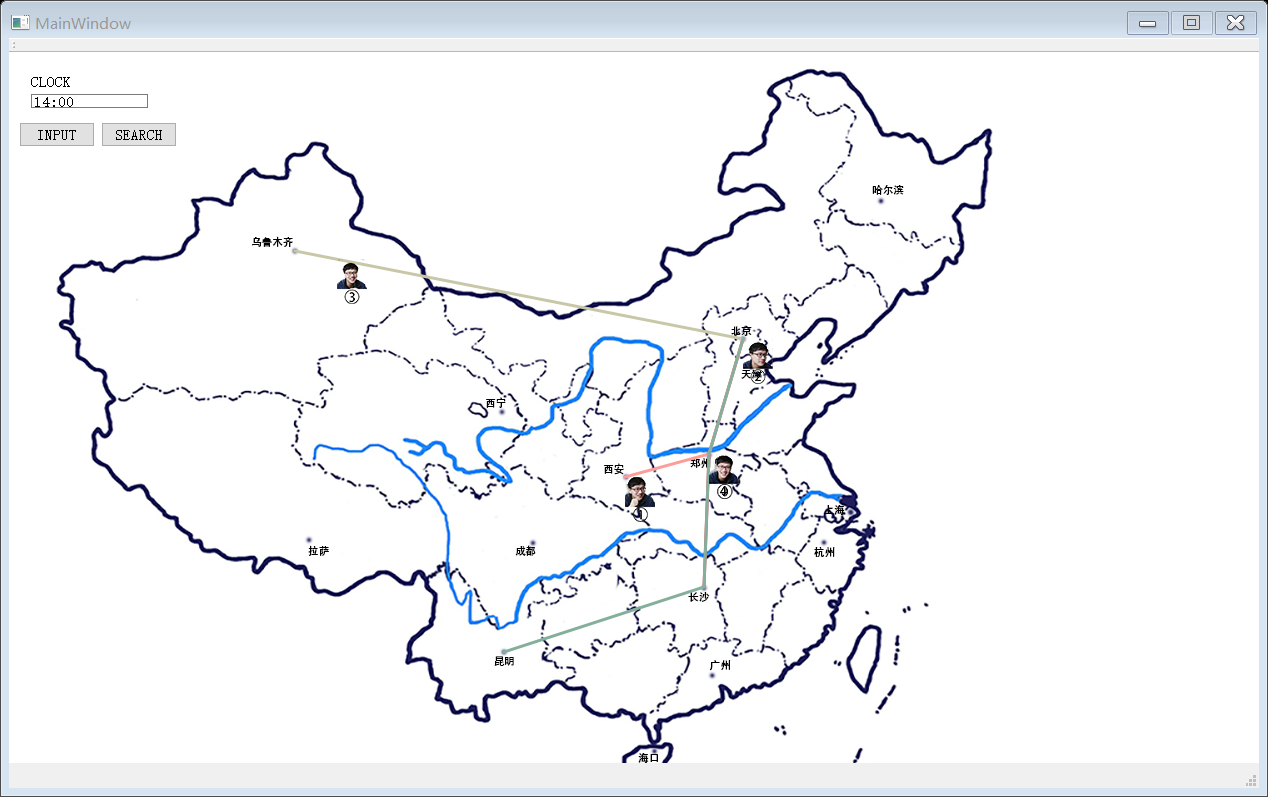
if(tempNode.total\_time > limtTime || tempNode.total\_cost > miniMoney)

当然BFS是基于队列的， 结点入队时仍有一些技巧：它需要记录是依靠哪条边来的，已经在这条边之前的全部边的集合（部分路径）。

对于中间城市的可选情况，算法一和算法二选择统一的处理方式，重写函数，并增加反复调用机制。我们只需要构建一个temp\_city[MAX\_CITIES]数组，这个数组是包括出发城市、途经城市、到达城市的有序城市集合，在这个数组上对每相邻两个元素调用算法一和算法二即可。对于算法三，既然已经是BFS搜索了解空间，不妨加设一条剪枝条件：isMatch(arr\_MidCity,arr\_size,tempNode.route\_stml)。该条件检查了匹配情况，即当前输出路径如果不包含正确顺序的途经城市子序列，那么该路径也要被剪除。

1. **主窗口显示模块**

本模块由mainwindow.cpp构成。



如上图为程序的主窗口显示界面，即图形化显示界面。该界面的地图为旅行图，当一个旅客发出请求并返回一条非空路径时，线路标识会同时出现在地图上，人头像即代表旅客。左上角包括响应按钮部分（上面已有所介绍）和系统时钟。每个用户的旅行过程是相互独立且同时进行的。

基于这样的设计，可以很方便的看出用户预期的旅行路线，当这个路线被更改时，程序也会立即更新这个新的路劲。如果算法返回了一个空路径，那么在窗口上不会有任何路径显示，用户在到达最近目的地后停止，进入等待状态。

1. 时序模拟和状态转移逻辑模块

本模块由simulation.cpp和mainwindow.cpp构成，负责进程模拟和状态转移。

由于需要随时接收改变状态的响应，在INPUT返回时，我们就需要对当前状态进行刷新，以便衔接接下来的状态序列，这个功能由过程void init\_state(int target,Graph &travel\_graph) 实现。这是一个从STATE\_NOW向STATE\_N转换的功能函数。当然我们也需要从当然前的STATE\_N中预测出算法需要的STATE\_NOW，这个过程由input.cpp中307行至323行代码完成。

状态转移的核心逻辑由过程S\_STATE state\_trans(Graph &travel\_graph,int target)完成，它需要一个用户指针，考察这个用户的当前STATE\_N，然后返回下一个STATE\_N。调用这个过程，可以构成如下状态转移方法：

void \_simulation(Graph &travel\_graph)

{

printState();

//cout<<"MOVE FLAG : "<<moveflag<<endl;

\_\_MUTI\_TST\_LOOP\_\_

{

arr\_tst[i].timepass++;

if(arr\_tst[i].timepass == arr\_tst[i].state\_n.state\_dur)

{

arr\_tst[i].state\_n = state\_trans(travel\_graph,i);

arr\_tst[i].timepass = 0;

}

}

}

printState是日志和命令行输出过程，轮询宏向每一个用户发出状态更新要求，每一个用户维持这一个状态流逝变量timepass，它衡量当前状态已经经历了多久，当与STATE\_N结构中的STATE\_DUR相等时说明这个状态的生命周期已到，需要进行状态转移，此时我们调用state\_trans过程，同时将timepass置0。