OO\_11说明文档

1. 程序功能功能说明：

程序模拟出租车的乘客呼叫与应答系统，采用多线程的设计，并结合了相应的GUI即时显示结果。

1. 程序运行环境与运行指令规范

程序运行环境为Java(TM) SE Runtime Environment (build 1.8.0\_73-b02)，运行IDE为Intellij Idea 2017.1.2。如果用命令行编译并运行，编译指令为javac -encoding utf-8 \*.java，运行指令为java TaxiSys。最后输出结果保存在Log.txt文件中。

1. 程序注意事项

由于官方GUI的性能问题，在同时输入过多请求的时候可能产生程序运行的迟缓现象，又因为本次作业不强制要求使用GUI，故程序取消了部分GUI效果，包括红绿灯变化以及用户请求范围的显示。

程序提供了一些Taxi类中的接口函数来查询单个具体的出租车的当前状态，它们是：getId():返回出租车id，getX():返回出租车当前所在行，getY()返回出租车当前所在列（注意这里的行列都是以1开始，以80结束的数字），getState()获取出租车当前运行状态，getCredit()获取出租车当前信用值。程序还在TaxiManager中提供接口查询多个出租车状态，queryTaxi(int \_state):查询状态为\_state的所有出租车，返回一个ArrayList，而其中的getState(int i)返回编号为i的出租车的状态。至于出租车当前时刻，则调用getCurTime()即可返回当前毫秒数。

对于红绿灯状态的查询，在TrafficLight类中提供了getEastWest(int index)与getSouthNorth(int index)两个方法来查询点index对应的位置的红绿灯状况，如果输入的index非法则只能返回2，如果输入的index合法则返回0、1或者-1，分别表示该点没有红绿灯、此时该方向上红灯亮、此时该方向上绿灯亮。

程序规定所有的红绿灯变化的周期统一为TrafficLight类初始化时随机产生的一个介于[200,500]的值。

对于各个类的不变式，程序将它们都放在了repOK()的JSF中。

1. 程序输入说明

程序可以接受 [CR,(srcX,srcY),(dstX,dstY)]形式的输入，输入之间的空格都会被替换，因此接受含有空格的输入。其中的坐标值(srcX,srcY,dstX与dstY)都是以1开始，以80结束的形式表示，这与以后的路径显示和车辆位置记录中所采用的是同一种表示形式。此外，程序还接受一行不超过10个请求输入，例如[CR,(srcX,srcY),(dstX,dstY)]; [CR,(srcX,srcY),(dstX,dstY)]...其中要求请求之间用分号隔开，而且注意程序采取的分隔方式是严格按照分号分隔，两个分号之间、第一个分号与输入行首位之间、最后一个分号与输入行末尾之间都被认为是一个请求。此外，程序接受输入”end”作为整个程序结束的标志。

对于开闭路请求，程序支持用[CP, (srcX,srcY),(dstX,dstY)]表示关闭点(srcX,srcY)到(dstX,dstY)的路径，而[OP, (srcX,srcY),(dstX,dstY)]表示打开点(srcX,srcY)到(dstX,dstY)的路径。其中关闭路径不能是关闭本来就不存在的路径，起止点也不能相同。同理打开路径也不能打开本来就存在的路径或者根本不相邻的两点之间或者一开始根本不存在的路径。在关闭路径的时候，程序是等路径上的车都通过了之后才会真正切断路径，而程序是在200ms末将车瞬移到目的地的，因此可能存在在GUI上看似切断了路径但是车还是移动了过去的情况，因为切断路径的时刻处于200ms的中间，而此时从理论上讲车在路上，所以依然会移动过去。此外，程序对于增删路改变了点的路口数的情况，不再对红绿灯进行调整， 比如说，原先一个丁字路口被删掉一条边变成了只有两条边，那么此时该点的红绿灯还在，这可以解释为，我们的开闭路请求只是临时开闭路，因此无需改变最开始设置的红绿灯。最后，程序规定在服务中的出租车的行为是在开闭路指令发出后立即检测是否应该改变路径，而不是等到了相应的边的时候再改变。

所有关于坐标的输入（形如”(x,y)”）均支持前导零和正号。

输入的地图文件中允许有空行，地图的连通性需要由测试者保证。开闭路之后地图的连通性也需要由测试者保证。输入的交通信号灯文件的正确性也应该由测试者保证。如果地图文件或者信号灯文件不正确，那么程序会在报错之后自动退出。

1. 程序输出说明

程序对于格式错误的输入会报错，并且对于和以前的请求重复或者起始点重复的输入也会报错。程序会调用本次官方提供的GUI显示车辆与乘客状态。程序还会产生一个总的记录文件，记录文件中记录了每个用户刚刚发出请求时，周围车辆的状态，信用(Customer#i First Logging: )。以及抢单时间窗口内所有抢过单的车辆(Taxis tried to grap Customer#i's order:)，最后被选中的车辆(Taxi chosen by Customer#i : Taxi#j)以及车辆将乘客从起点送到终点的路径(Path found for Customer#i)。不同的乘客之间用一串短横线隔开，由于出租车完成每个乘客的请求的顺序不一定和乘客请求产生顺序一致，因此输出的乘客编号不一定是从小到大排列的。注意乘客假如乘不到车那么这些记录就是不完全的。建议用Sublime Text等编辑器打开输出的Log.txt文件，那样显示效果会更好。

此外，如果是在测试线程中向特定出租车加入的乘客请求，不会输出到Log.txt而只会输出到屏幕。因为没有经过调度类的调度，也就不会有正常的从控制台输入的请求所具有的3s窗口内抢单出租车之类的东西，因此程序规定对于这种强行加入出租车的乘客请求，不输出任何东西到文件。

1. OO11补充说明

本次作业还需要有以下说明：

首先，在Taxi.java中的TrackableTaxi类就是本次支持的可追踪出租车，测试者在编写测试线程的时候可以通过语句

deIterator**<*TrackableTaxi*.*Record*>** recorddeIterator **= ((*TrackableTaxi*)*taxis*[*99*]).getRecords()**

获取编号为99的出租车（前提是它是一辆可追踪出租车）的历次服务记录迭代器。recorddeIterator拥有前后迭代车辆服务记录的能力，它的next方法返回的是游标当前所在位置的Record，而它的previous方法则返回的是游标之前的位置的Record，测试者可以通过它的hasPrevious和hasNext()来判断是否存在前后元素。

上面的recorddeIterator提供的是迭代车辆历次服务的能力，可以通过它来获取该次服务的用户请求时间，用户请求起点，用户请求终点，详情请见TrackableTaxi中的内部类Record。

而要迭代访问某一次服务中出租车行驶的总路径（包括从接单位置到用户起点的路径与从用户起点到用户终点的路径），测试者可以使用语句

deIterator**<*Point*> *pointdeIterator* = *recorddeIterator*.next().getPoint();**

得到的迭代器pointdeIterator具有双向迭代访问可追踪车辆在某次服务过程中行驶的全部路径点。

注意，在使用迭代器的时候，需要在车辆hasNext或者hasPrevious的情况下才能调用recorddeIerator的next()方法或者previous返回一个record，否则得到的结果是null，则在获取pointdeIterator的时候会出错。

在TaxiSys.java中提供了一个几乎为空的init\_taxi方法，其中有对输入的判断以及相应的规格要求，并且返回一个Taxi[]，即地图中跑的所有出租车的数组。测试者可以在遵循相应规格的前提下，自己完成该函数并注释掉main方法里面的相应代码（已经指出），实现随机产生70辆正常出租车和30辆可追踪出租车的作用。