



**数据结构课程设计**

**<模块设计报告>**

**兰学超 2019211564**

**李培阳 2019211580**

**熊瑞东 2018211612**

**模块设计**

## 1 Login模块

登录模块，提供用户登录窗口，并检测账号密码是否存在于本地文件且是否对应。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void on\_pbt\_login\_clicked() | 登录按钮对应的槽函数，检查用户输入的账号密码与文件是否对应，若对应则关闭窗口返回accepted，不对应则弹出提示框。 |
| bool checkLogin(const QString&id, const QString &pwd) | 参数分别为用户输入的账号和密码，检查账号是否存在于文件中，账号密码是否对应。 |
| void on\_pbt\_exit\_clicked() | 退出按钮槽函数，关闭窗口，返回rejected。 |
| void readAccounts(); | 读取文件中的账号密码及课程表信息。 |

## 2 Account模块

账户模块，用于存储用户的账号密码信息与课程时间表信息。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| Course gotoclass(Clock clock) | 参数为当前时间，结合课表信息判断当前是否有课，若有课返回当前在上的课，若没课返回-1。 |

## 3 Mainwindow模块

程序主窗口模块，串联所有核心算法模块，提供用户与系统的交互界面，并以图形化的方式输出。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void initAction() | 初始化工具栏按钮 |
| void initToolBar() | 初始化工具栏 |
| void initTimer() | 初始化定时器 |
| void readAdj() | 读入地图，存入邻接表 |
| void readVer() | 读入节点信息表 |
| void readL2P() | 读入逻辑名到物理名映射表 |
| void readShuttleSchedule() | 读入班车时刻表 |
| void currentOutput() | 输出当前状态信息 |
| bool inShahe(int spot) | 判断传入的点是否在沙河校区 |
| bool inBenbu(int spot) | 判断传入的点是否在本部校区 |
| bool inBetween(int spot) | 判断传入的点是否在两校区之间 |
| void searchNear() | 搜索附近一定范围的建筑并提供行进路线、距离和时间 |
| void searchSpot() | 搜索单个建筑并提供行进路线、距离和时间 |
| void findPath() | 两点寻路函数 |
| void antFind() | 多点寻路函数 |
| void switchShahe() | 切换至沙河地图 |
| void switchBetween() | 切换至校外地图 |
| void switchBenbu() | 切换至本部地图 |
| void setTime() | 设置时钟加速的结束时间 |
| void drawPath(const QVector<Vertex\*>  &path, bool currentIn) | 绘制整条路径，参数分别为要绘制的路径、是否将当前位置绘入路线的标志 |
| void move() | 更新当前状态并画出行进路线，若跨校区则进行地图切换 |
| void clear() | 擦除地图上的所有路线 |
| void setCurrent(const QVector<Vertex\*>&path, qreal dis, qreal time, Priority pr) | 设置当前状态，传入的参数分别为导航路径、距离、时间和优先度 |
| void setCurrentToVer(int no) | 将当前状态移动至指定的顶点 |
| void updateTime() | 系统时间+1s并显示在文本框中 |
| int getFirstShuttleTime(qreal t) | 参数为到达班车上车点所需的秒数，返回到达上车点后还需要的等车秒数 |
| int getFirstBusTime(qreal t) | 参数为到达班车上车点所需的秒数，返回到达上车点后还需要的等车秒数 |
| void fuzzySearch(const QString &str) | 参数为输入的字符串，执行模糊搜索，将搜索结果记录在数组中，并依此动态创建多个选择按钮 |
| void mouseDoubleClickEvent  (QMouseEvent \*e) | 鼠标双击事件，在界面中显示当前点击顶点的相关信息 |
| void setVelocity(int x) | 设置速度函数 |
| void clock\_slot() | 定时器超时执行的槽函数，包括currentOutput、updateTime、move等。 |
| void cancel() | 清除右侧创建的按钮 |
| void reenter() | 重新输入函数，清除右侧按钮，清除文本输入框 |
| void choose\_findpath\_time() | 选择最短时间路线 |
| void choose\_findpath\_dis() | 选择最短距离路线 |
| void choose\_searchnear(int i) | 从搜索附近的所有结果中选择一个作为终点 |
| void choose\_search\_fzs(int i) | 从搜索地点文本框的模糊搜索的所有结果中选择一个作为文本 |
| void choose\_fp\_begin\_fzs(int i) | 从两点寻路起点文本框的模糊搜索的所有结果中选择一个作为文本 |
| void choose\_fp\_end\_fzs(int i) | 从两点寻路终点文本框的模糊搜索的所有结果中选择一个作为文本 |
| void choose\_ant\_fzs(int i) | 从多点寻路文本框的模糊搜索的所有结果中选择一个作为文本 |
| void antfindNext() | 继续输入多点寻路的下一个途经地点 |
| void antfindStart() | 多点寻路开始导航 |
| void antfindStartDis() | 选择多点寻路的最短距离路线 |
| void antfindStartTime() | 选择多点寻路的最短时间路线 |
| void stopTime() | 暂停系统时钟 |
| void logSearch() | 打开日志搜索窗口 |
| void gotoClass() | 根据当前时间和用户课程表，导航去教室 |
| void search\_fzs(const QString &str) | 对搜索地点文本框中的内容进行模糊搜索 |
| void findpath\_begin\_fzs(const QString &str) | 对两点寻路起点文本框中的内容进行模糊搜索 |
| void findpath\_end\_fzs(const QString  &str) | 对两点寻路终点文本框中的内容进行模糊搜索 |
| void antfind\_fzs(const QString &str) | 对多点寻路文本框中的内容进行模糊搜索 |

**算法分析：**

1. **导航线路推进：**

用求得的路径数组、距离、时间等信息设置当前状态，表示开始导航。在clock\_slot中调用的move函数中，程序计算经过系统的1s后前进的距离，更改当前状态中的current\_pass及上一顶点、下一顶点等信息。同时将Qt的painter\_path连线到当前状态的坐标。具体推进伪代码如下：

Double t = 1；

double v = 当前速度\*所在边的拥挤度；

Double d = t\*v；

While（当前状态加上d足以到达下一顶点） {

If(下一顶点是当前路径终点) {

更改当前状态到终点（坐标、已走距离/时间、上一/下一节点、速度=0等）

}

Else {

t -= 走到下一节点所需时间；

更改当前状态到下一顶点（坐标、已走距离/时间、上一/下一节点、当前边距离等）；

v = 所在新边的拥挤度\*当前速度；

d = v \* t；

绘制路线（连线到当前顶点）；

}

}//while

If(没到终点) {

更改当前状态到下一顶点（坐标、已走距离/时间、上一/下一节点、当前边距离等）；

}

绘制路线（连线到当前坐标）；

1. **两点寻路跨校区导航：**

由于跨校区校外路段及班车、公交的特殊性，采取分段导航的策略。如果在寻路时发现起止点不在同一校区，即设置跨校区cross标志为1，并进行3段寻路（以公交为例）：第一段从起点到当前校区公交上车点；第二段为设定好的公交路线；第三段从目的校区公交上车点到终点。

开始导航后若当前状态到达第一段路线终点，系统自动用第二段路线设置当前状态，到达第二段终点时同理。

1. **途经多点导航：**

由于途经多点导航与传统的旅行商问题略有区别：并非遍历所有顶点也并非需要回到起点，故蚁群算法有所改动。

不能以原始邻接表adj\_map作为蚁群算法的邻接表，会出现两顶点不连通的情况。故应二重循环遍历途经顶点数组，两两之间使用dijkstra算法生成有向最短路径，并存于抽象完全图QMap<int,QMap<int,Dijkstra>> abmap中（abmap[i][j]存储从顶点i到顶点j的最短路径、距离、时间）。通过将蚁群算法使用的邻接图设为此abmap，问题就解决了。但是生成该abmap的时间复杂度是O(n^4)，效率略低，不过总体相比旅行商问题的指数级是十分优秀的，并且准确度较高。

1. **途经多点跨校区：**

类似的，多点寻路也会面临跨校区的问题。解决方法也是分段寻路，以下以公交为例。

首先将途经顶点数组分为两个：沙河顶点数组和本部顶点数组。然后将沙河公交上车点加入沙河顶点数组（如果原本没有），本部同理。然后对应两个顶点数组分别生成两个abmap。

随后遍历沙河顶点数组和本部顶点数组，将不同校区的所有途经顶点构建为一个abmap（沙河顶点a到本部顶点b的路径由3段合成：a到沙河公交点路径+跨校区路径+本部公交点到b的路径）。以该abmap进行蚁群算法。

接着将蚁群后得到的路径拆分，由于最短路径的必然性，该路径中必然可分为三段，第一段为沙河校区路径（终点为沙河公交点），第二段为跨校区路径，最后一段为本部路径（起点为本部公交点）。

开始导航后若当前状态到达第一段路线终点，系统自动用第二段路线设置当前状态，到达第二段终点时同理。

1. **离开/进入校园自动切换地图的显示：**

当前状态校园出口并即将跨校区时，系统自动切换校区是合理的需求。为了易于实现该操作，我们多设置了两个逻辑上的顶点，称之为沙河和本部切地图点，以沙河切地图点为例。

沙河切地图点物理上与沙河校门点重合，即从校门到该点的距离为0。利用边距离为0这一特殊性，我们在move函数中加入判断：若当前上一顶点为沙河校门点且当前边距离为0且跨校区标志cross==true，将自动切换地图。本部同理。

1. **中文模糊搜索：**

在C++中英文一个字符占一个字节，而中文一个字符占3个字节，故不能采用单纯的字符串匹配函数。我们在使用string::find的基础上增加模3（%3）操作，判断搜索到的子串第一次出现的下标是否为3的整数倍，如果是才可认为匹配成功。

该算法能解决绝大部分中文匹配异常的问题，但仍有样例会造成错误，如“人类”和“死”将被认定为相同字符串。不过由于我们的程序中不涉及错误字段，故可以信任该算法。

该算法应提前筛除匹配串为空串的情况，否则一定认为匹配成功：空串是任意串的子串，出现下标为0。

1. **模糊搜索结果选择**

通过动态创建多个QLabel和QPushButton，并绑定同一槽函数，使用Lambda表达式传值以确定所点击按钮的编号，实现将多个搜索结果展示于用户界面供用户选择。用一个指针数组记录动态创建的组件，以便在选择后全部删除释放空间。

1. **路线显示问题**

只有在当前画面显示的地图与当前导航所处地图相匹配的情况下才显示导航的线路。因此通过设置当前显示地图的标志，将之与当前状态所在地图进行匹配，若成功匹配则将当前位置的小圆点加入到QGraphicsScene中，并且将QPathItem的path设置为当前地图对应的QPainterPath。如此可实现即使地图不匹配，图形化的移动仍会进行，只不过没有显示出来。

## 4 Clock模块

时钟模块，提供模拟导航系统的时间。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| QString get\_clock\_str() | 将时钟转化为格式化字符串并返回 |
| void update\_clock() | 系统时钟更新，+1s |
| int difSec(const Clock &clock) | 求两时钟相差的秒数 |
| bool operator>=(const Clock &clock) | 重载运算符>=：用于判断时间的大于等于关系 |
| Clock operator-(int h) | 重载运算符-：用于计算时间减去小时数得到的时间 |

## 5 Current模块

当前状态模块，提供用户当前状态信息及周边信息。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| bool inShahe() | 判断当前状态是否在沙河校区 |
| bool inBenbu() | 判断当前状态是否在本部校区 |
| bool inBetween() | 判断当前状态是否在校外地图 |

## 6 Dijkstra模块

迪杰斯特拉算法模块，根据传入的优先度（距离/时间）提供两点间的最短距离（时间）路径的算法，并且结合BFS实现搜索一定范围内的所有顶点及其路径的功能

共实现四种功能：

1. 单源点的两点寻路：

起点和终点均为顶点的最短路径算法。

1. 双源点的两点寻路：

起点位于边的中间（不在顶点上），终点为顶点的最短路径算法。

1. 单源点的范围搜索：

起点为顶点的Dijkstra+BFS算法。

1. 双源点的范围搜索：

起点位于边的中间（不在顶点上）的Dijkstra+BFS算法。

**主要函数及功能如下：**

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void singleSearch(int begin, qreal r,  qreal v) | 搜索单源点一定距离范围内的建筑并计算到各建筑的行进路线 |
| void doubleSearch(int begin1, int begin2,   qreal r, qreal v, qreal ldis, qreal rdis) | 搜索双源点一定距离范围内的建筑并计算到各建筑的行进路线（当点位于某边而非节点上时，以边两端的节点作为源点） |
| void singleFind(int begin, int end,  Priority priority, qreal v) | 根据不同策略计算单源点到终点的行进路线 |
| void doubleFind(int begin1, int begin2,  int end, Priority priority,  qreal v, qreal ldis, qreal rdis) | 根据不同策略计算双源点到终点的行进路线（当点位于某边而非节点上时，以边两端的节点作为源点） |
| void start() | Dijkstra算法 |
| void search() | Dijkstra+BFS算法 |
| void insert\_to\_visit(int x) | 插入待遍历顶点，保证顶点集有序 |

**1.单源点两点寻路算法（Dijkstra算法）描述：**

1. **数据结构**

QVector<int> to\_visit;

待遍历顶点集，维持为一个距离（时间）递减的优先队列，每次遍历弹出其队头元素。

QMap<int,int> in\_visited;

已处理的顶点集，Key为顶点编号，Value为1表示在集合内，否则不在。由于in\_visited不需要遍历，而需要经常查询某个顶点是否已处理，故使用底层为红黑树的QMap容器，提高查询效率。

QMap<int,int> parent;

父顶点表，Key为顶点编号，Value为其对应的父顶点的编号。同样由于经常需要查询某顶点的父亲，故用QMap存储。该表用于逆向生成路径。

QVector<Vertex\*> dij\_path;

路径数组，值为顶点指针。

QMap<int,qreal> dis;

距离表，Key为顶点编号，Value为其到起点的距离。使用QMap原因如上。

QMap<int,qreal> time;

时间表，Key为顶点编号，Value为其到起点的时间。使用QMap原因如上。

**（2）操作步骤**

① 初始时，visited为空，to\_visit中只有起点s。

② 然后，从to\_visit中弹出队头顶点（距离（时间）最短的顶点）记为current，并将其加入到visited中。

③ 接着，遍历不在visited中的与current邻接的所有顶点，计算他们到current的距离d（时间t），如果比原先到该点的距离（时间）短，就更新它，并且将其parent设为current。

④ 重复②③操作，直到所有顶点都已加入visited。

最后通过parent表，从终点end逆向求出最短路径。

伪代码如下：

While（to\_visit非空）{

Current = to\_visit队头；

弹出to\_visit队头；

current加入visited；

For(所有current的邻接点) {

If(该点in\_visited) continue；

If((优先度为距离优先 && 该点到current的距离+current 到起点的距离 < 该点到顶点的距离) ||

(优先度为时间优先 && 该点到current的距离+current 到起点的时间 < 该点到顶点的时间)) {

更新dis；

更新time；

该点的parent = current；

}

If（current没有父亲） current的parent = begin；

}

}

int p = end;

while(p!=begin) {

dij\_path.push\_front(vertices[p]);

p = parent[p];

}

dij\_path.push\_front(vertices[begin]);

**2.双源点两点寻路算法（Dijkstra算法）描述：**

分别从当前所在边的左端点（记为lbegin）、右端点（rbegin）进行单源点寻路dij\_left和dij\_right。记当前位置到lbegin的距离为ldis、时间为ltime；到rbegin的距离为rdis、时间为rtime。

若dij\_left的距离+ldis<dij\_right的距离+rdis（时间同理），取dij\_left路径为最短路径，并将dij\_left的路径起点为current状态的下一顶点。

**3.单源点范围搜索算法（Dijkstra+BFS算法）描述：**

数据结构与算法1类似，仅用如下数据结构替换dij\_path：

QMap<int,QVector<Vertex\*>> paths：

路径表，Key值为顶点编号，Value为以该顶点为终点的最短路径。

伪代码如下：

While（to\_visit非空）{

Current = to\_visit队头；

弹出to\_visit队头；

if(dis[current]>搜索半径) break;

current加入visited；

For(所有current的邻接点) {

If(该点in\_visited) continue；

If((优先度为距离优先 && 该点到current的距离+current 到起点的距离 < 该点到顶点的距离) ||

(优先度为时间优先 && 该点到current的距离+current 到起点的时间 < 该点到顶点的时间)) {

更新dis；

更新time；

该点的parent = current；

}

If（current没有父亲） current的parent = begin；

}

逆向求出current的路径，加入路径表

QVector<Vertex\*> temp;

int p = current;

while(p != begin) {

temp.push\_front(vertices[p]);

p = parent[p];

}

temp.push\_front(vertices[p]);

paths[current] = temp;

}

## 7 Ant模块

蚁群算法模块，提供途径多点的最短路径算法。

由于途经多点算法与传统的旅行商问题略有不同，因此蚂蚁最终不必回到起点，蚂蚁行进使用的完全图也并非初始邻接表，而是根据途经顶点集合由Dijkstra算法生成的（该操作在主窗口内完成，故此模块中不涉及）。

**蚁群算法描述：**

**（1）基本原理**

① 根据具体问题设置多只蚂蚁，分头并行搜索。

② 每只蚂蚁完成一次周游后，在行进的路上释放信息素，信息素量与解的质量成正比。

③ 蚂蚁路径的选择根据信息素强度大小（初始信息素量设为相等），同时考虑两点之间的距离，采用随机的局部搜索策略。这使得距离较短的边，其上的信息素量较大，后来的蚂蚁选择该边的概率也较大。

④ 每只蚂蚁只能走合法路线（经过每个地点1次且仅1次），为此设置禁忌表来控制。

⑤ 所有蚂蚁都搜索完一次就是迭代一次，每迭代一次就对所有的边做一次信息素更新，原来的蚂蚁死掉，新的蚂蚁进行新一轮搜索。

⑥ 更新信息素包括原有信息素的蒸发和经过的路径上信息素的增加。

⑦ 达到预定的迭代步数，或出现停滞现象（所有蚂蚁都选择同样的路径，解不再变化），则算法结束，以当前最优解作为问题的最优解。

**（2）信息素及转移概率的计算**

——本次迭代边（i，j）上的信息素增量；

——第k值蚂蚁在本次迭代留在边（i，j）上的信息素量；

——信息素增加强度系数：500

——蚂蚁个数：58

——最大迭代次数：100

——信息素蒸发系数：0.75

——持久性（残留）系数；

——时刻t蚂蚁k由城市i转移到城市j的概率（转移概率）；

——蚂蚁k的禁忌表。

——信息素浓度表

——能见度表

——信息素比例因子：0.8

——能见度比例因子：4.2

——计算公式：

**（3）算法步骤**

① 初始化参数:开始时每条边的信息素量都相等。

② 将各只蚂蚁放置各顶点，禁忌表为对应的顶点。

③ 取1只蚂蚁，根据公式计算出转移概率，按轮盘赌的方式选择下一个顶点，更新禁忌表，再计算概率，再选择顶点，再更新禁忌表，直至遍历所有顶点1次。

④ 计算该只蚂蚁留在各边的信息素量，该蚂蚁死去。

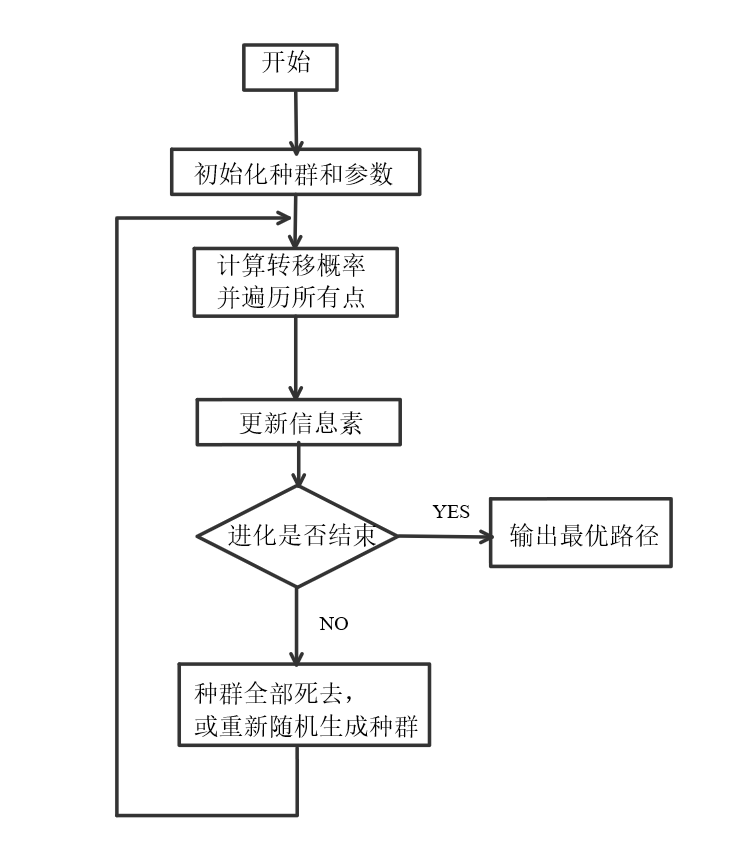
⑤ 重复③④，直至m只蚂蚁都周游完毕。

⑥ 计算各边的信息素增量，和信息素量。

⑦ 记录本次迭代的路径，更新当前的最优路径，清空禁忌表。

⑧ 判断是否达到预定的迭代步数，或者是否出现停滞现象。若是，算法结束，记录当前最优路径及距离、时间; 否，转②，进行下一次迭代。

**算法流程图如下：**

****

**主要函数及功能如下：**

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void start(const QVector<int> &spots,  Priority priority) | 蚁群算法开始迭代，参数分别为顶点集和优先度（距离/时间） |
| void clearAll() | 清空所有辅助表（ph、visibility等） |
| void shuffle() | 洗牌，随机取一个顶点作为蚂蚁的起点 |
| void move() | 蚂蚁根据概率，结合轮盘赌进行移动 |
| void record() | 记录单次迭代的最短路径、距离、时间等 |
| void record\_inall() | 记录所有迭代中的最短路径、距离、时间等 |
| void update() | 更新每条路径的信息素浓度 |
| void init\_ph() | 初始化信息素浓度表 |
| void init\_delta\_ph() | 初始化信息素增量表 |
| void init\_visibility() | 初始化能见度表 |
| void init\_tabu() | 初始化禁忌表 |
| void clear\_tabu() | 清空禁忌表 |
| qreal rnd(qreal lower, qreal upper) | 返回lower到upper中的一个随机数 |

## 8 LogSearch模块

日志查询模块，提供查询日志文件相关信息的功能。

主要函数及功能如下：

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void sendData(Clock cur) | 主窗口传入当前时间 |
| void on\_pushButton\_clicked() | 查询按钮的槽函数，根据操作和时间筛选日志记录并输出 |