正交路由选路算法

算法思路

该算法核心部分来自OCR算法,但是为了更好的用户体验和更好地满足用户需求,需要对OCR算法进行修改.

> OCR算法通过建立图形连接点之间的正交网络寻找两点之间可以避开障碍物并且代价最小的路径.OCR算法规定了正交网络的生成以及拐点数计算的规则，寻路算法采用A＊算法
>
> > **代价** = 两点的曼哈顿距离 + 拐点数
> > **曼哈顿距离** : 起点(x1,y1) 终点(x2,y2) H=|x2-x1|+|y2-y1|
> > **A*算法** : 一种启发式静态路由选路算法 公式表示为：**f(n)=g(n)+h(n)** 其中，f(n) 是从初始状态经由状态n到目标状态的代价估计，g(n)是在状态空 间中从初始状态到状态n的实际代价，h(n)是从状态n到目标状态的最佳路径的估计代价.

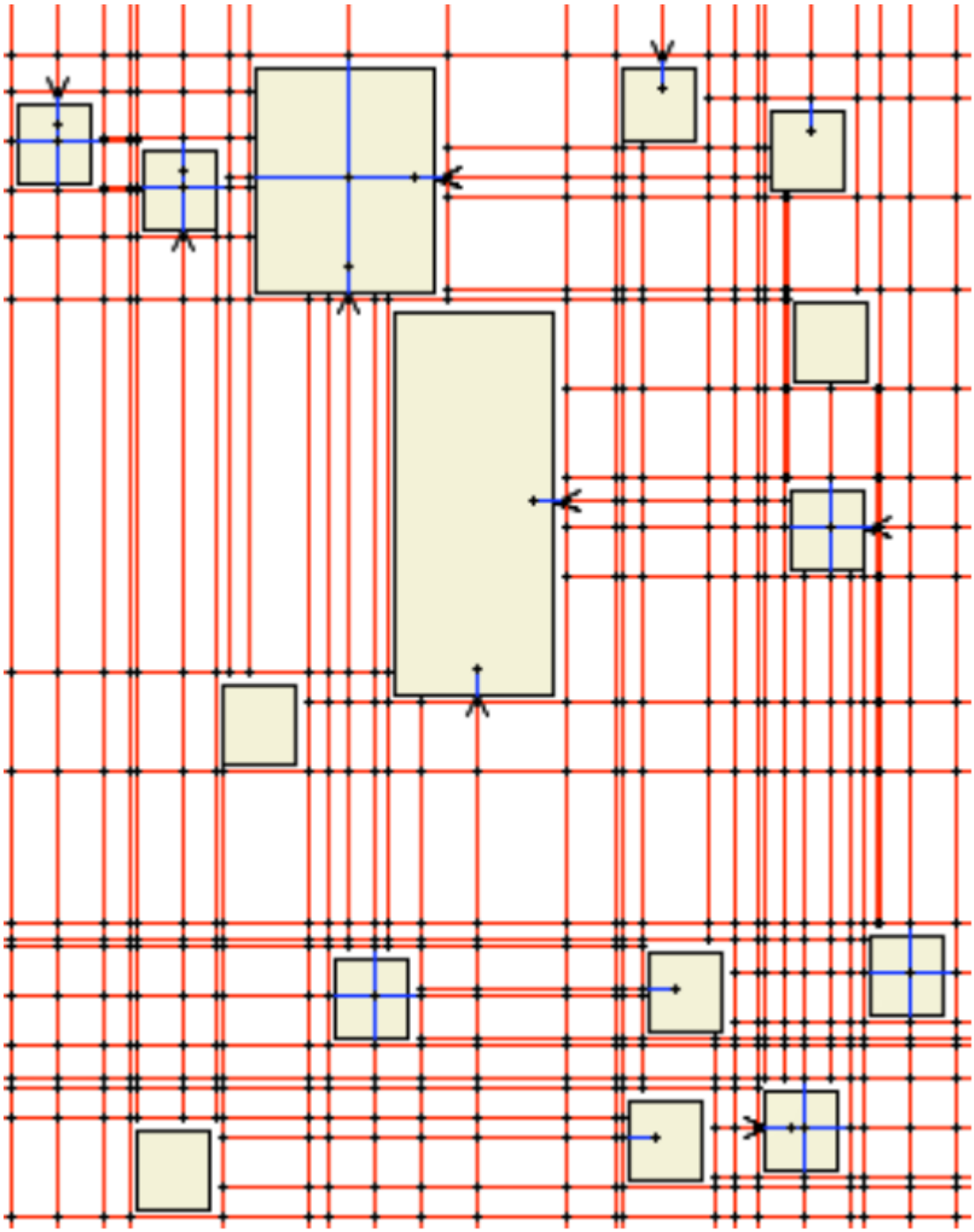
- 初始化正交路由网络

1. OCR算法生成正交路由时，将图形对应在Barrier的位置区间标志为不可通过（unwalkable），然后将四个顶点的坐标的X值和Y值分别加入到XI，YI中．需要注意的是,取四个顶点和中点时并不是直接取图形的坐标点,而需要向外扩展10个像素点后得到对应的点,避免路径和图形重合.
2. 但是为了适应更多的用户需求，在一些特定的情况下，有些图形是可以穿过的，这取决于起点和终点的位置．因此需要在每一次寻路时都创建一次正交路由网络.
3. OCR默认连接点为每条边的中点,但是实际需求为图上的任一点,所以XI，YI也需要在每一次寻路时重新赋值
4. 对应代码段`prepareLine()`,`initNullBarrier()`,`init()`,`initialBarrier()`等

- 初始化连接点

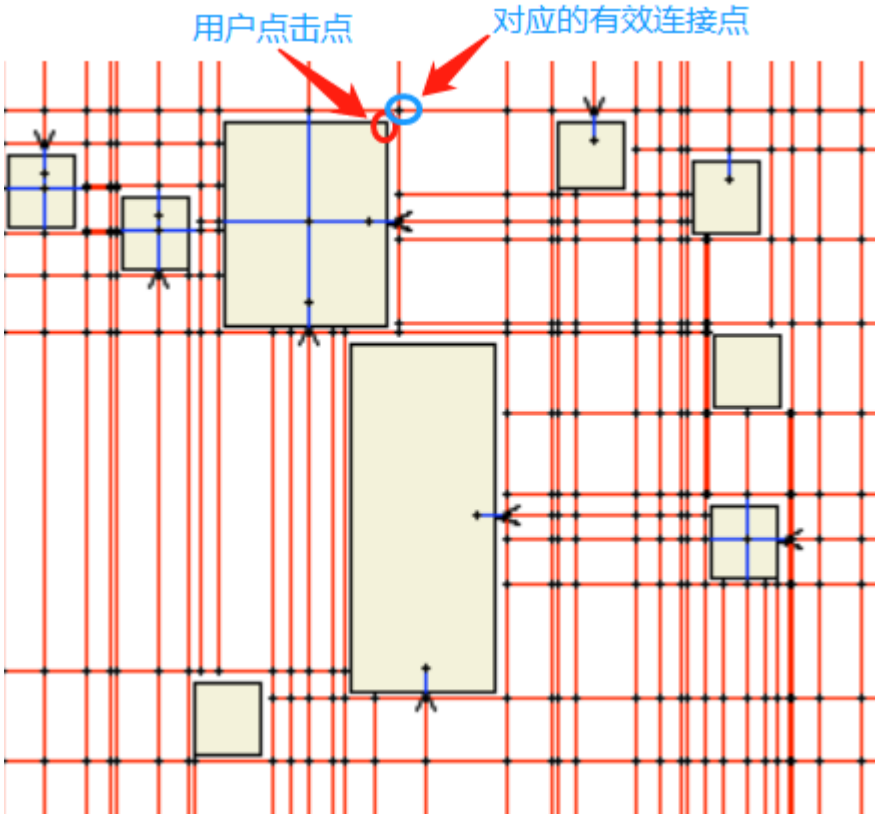
1. 计算XI和YI笛卡儿积的结果，若该点不落在不可穿越的图形内(walkable)，则设为有效连接点，加入到VectorPoint中,对应代码段`GenerateV()`

2. 实现后的效果如图,图中的黑色小点表示保存在VectorPoint的有效连接点,红色部分为有效连接点组成的正交网络(并不需要一个数据结构来保存).黄色部分在barrier里标记为0,白色部分标记为1



3. 从图中我们可以想到,用户点击的是图形的上的点,但是图形上的点并不是连接点,所以需要将用户点击的起点(或终点)转换为对应的最近的有效连接点,实现代码段`getStartLinkPoint()`,`getEndLinkPoint()`.

如图所示



- **寻路-A*算法** 有了起点和终点的有效连接点和正交网络后，就可以开始真正的寻路了，这一部分也是整个算法最核心的部分.

> 一般的A*算法会考虑一个节点与之相邻的8的点,但是在正交路由网络中,我们只考虑和该节点进入方向一致,右边以及左边的连接点,考虑这三个相邻的连接点时要按照优先级进行考虑,优先级:进入方向>右边>左边

1. 从起点 A 开始,并把它就加入到一个由有效连接点组成的 openlist中.现在openlist里只有一项,即起点A.
2. 根据优先级查找与起点A相邻的连接点,如果有则计算该点的代价加入到openlist中.把起点 A 设置为这些连接点的父亲.
3. 把 A 从openlist中移除，加入到 closelist中,closelist 中的每个连接点都是现在不需要再关注的。
4. 从openlist中得到代价最小的点,把该点从openlist里取出,放到closelist中;

> f = g + h

5. 根据优先级检查所有与它相邻的连接点，忽略已经加入closelist的连接点,如果点不在openlsit中，则把它们加入到 openlist中。
6. 如果某个相邻的方格已经在openlist中，则检查这条路径是否更优，也就是说经由当前连接点到达该连接点(已经在openlist中的邻接点)是否具有更小的g值。如果没有，不做任何操作。相反,如果g值更小，则把该连接点的父亲设为当前连接点,然后重新计算该连接点的f值和g值。
7. 不断重复4到6步,直到将终点加入openlist
8. 对应代码段

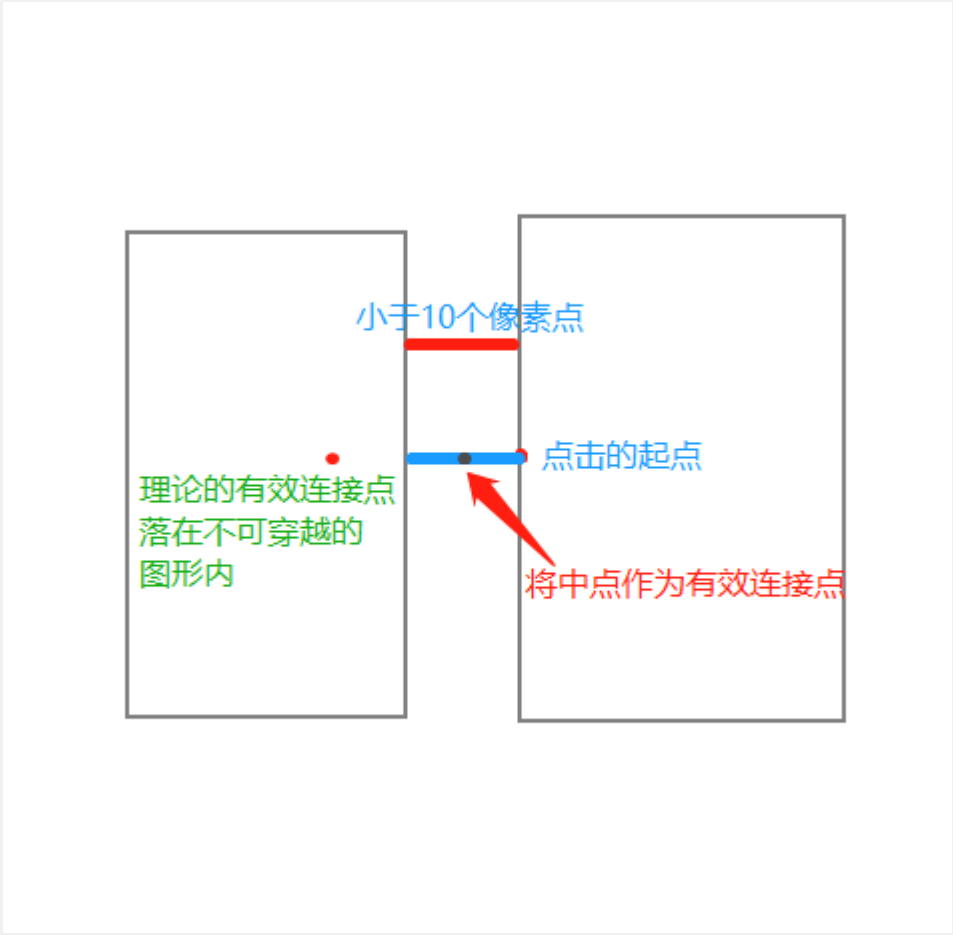
实现步骤

> 实际实现过程中,我们要考虑以下几个问题:

- 路径太贴近图形;
- 起点和终点因为紧挨着物体,所以没有有效连接点(有效连接点在10个像素点之外);
- 什么时候图形可以穿过,什么时候不可以;

解决方案:

1. 在代价一样的情况下,我们应该优先选择更远离图形的路径,所以当连接点靠近图形时,将其G值加上某个权重,降低该点的优先级
2. 当起点或终点找不到有效连接点时,将起点和图形之间的间隙的中点作为有效连接点,并且要加入到XI或YI中.



3. 图形是否可以穿过的规则如下

- 直接忽略的图形

1. 既包含起点也包含终点
2. 只包含起点且终点不在图形上;或者只包含终点且起点不再边上

- 不可以忽略的图形

1. 即不包含起点也不包含终点

- 特殊处理的图形:不需要在Barrier标记为不可走,但需要将顶点信息加入XI,YI

1. 包含起点且终点在该图形上；或包含终点且起点在该图形上

- 注: 包含指点在图形内部,不在边上;在图形上指

1. 根据图形是否可以穿过初始化Barrier和XI,YI

```
 if (startInfo.y > x &&
        startInfo.x > y &&
        startInfo.y < x + w &&
        startInfo.x < y + h)
 {
        flags = false; //包含起点
 }
 if (endInfo.y > x &&  endInfo.x > y &&
        endInfo.y < x + w &&  endInfo.x < y + h)
 {
        flage = false; //包含终点
 }
 if (flage && flags) {
        init(...rects[i]); //将图形对应在barrier数组相应位置的point.value全部赋值为0，表示不可穿越
        initialXIYI(x, y, w, h); //将图形的四个顶点加入到XI，YI
 }
 else if ((flage && !flags) || (!flage && flags)) {
        flags = true;
        flage = true;

        /*起点是否在图形上*/
        if (
        (startInfo.y == x &&
                (startInfo.x >= y && startInfo.x <= y + h)) ||
        (startInfo.y == x + w &&
                (startInfo.x >= y && startInfo.x <= y + h))||
        (startInfo.x == y &&
                (startInfo.y >= x && startInfo.y <= x + w)) ||
        (startInfo.x == y + h &&
                (startInfo.y >= x && startInfo.y <= x + w)))
        {
                flags = false; // 起点在图形上
        }
        /*终点是否在图形上*/
        if (
        (endInfo.y == x && (endInfo.x >= y && endInfo.x <= (y + h))) ||
        (endInfo.y == (x+w) && (endInfo.x >= y && endInfo.x <= (y + h))) ||
        (endInfo.x == y && (endInfo.y >= x && endInfo.y <= (x + w))) ||
        (endInfo.x == (y+h) && (endInfo.y >= x && endInfo.y <= (x + w))))
        {
                flage = false; // 终点在图形上
        }

        /*是否:包含起点且终点在该图形上；或包含终点且起点在该图形上*/
        if (!flage || !flags)
        {
                initialXIYI(x, y, w, h); //将图形的四个顶点加入XI,YI,但是图形可以穿越
        }
```

```
            /* 如果只是包含了其中一个点,则忽略该图形 */
    }
        /*如果包含了起点和终点，则忽略该图形*/
```

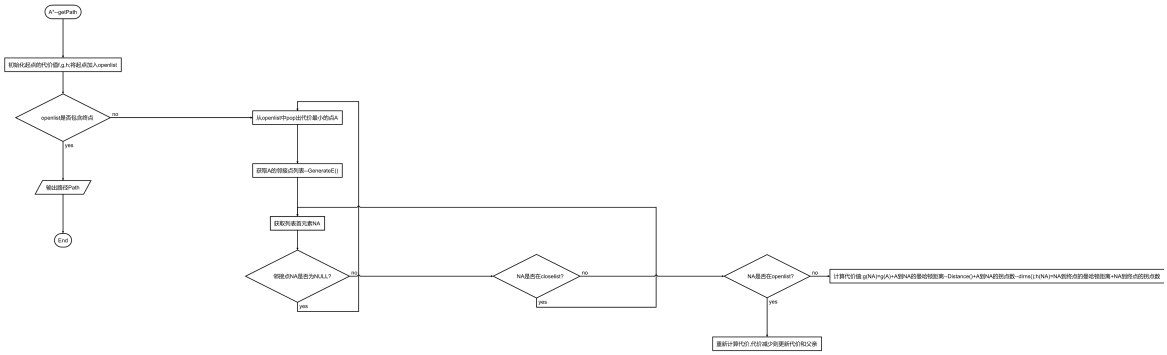
2. 将起点终点的坐标加入到XI,YI

```
            switch (startInfo.dir) {
                    case "N": // 若起点的前进方向为N,则将起点所在的行加入XI；其他同理
                            XI.push(startInfo.y);
                            break;
                    case "S":
                            XI.push(startInfo.y);
                            break;
                    case "E":
                            YI.push(startInfo.x);
                            break;
                    case "W":
                            YI.push(startInfo.x);
                            break;
            }
            switch (endInfo.dir) {
                    case "N":
                            XI.push(endInfo.y);
                            break;
                    case "S":
                            XI.push(endInfo.y);
                            break;
                    case "E":
                            YI.push(endInfo.x);
                            break;
                    case "W":
                            YI.push(endInfo.x);
                            break;
            }
```

3. 生成VectorPoint,即正交路由网络--`GenerateV()`

4. 将起点和终点转换为相应的有效连接点--`getStartLinkPoint()`,`getEndLinkPoint()`

5. 执行A*算法--`getPath()`



实现效果