**基于A\*算法的角色寻路软件V1.0说明书**

1需求分析

1.1开发背景

随着互联网和计算机的发展，游戏成为了一种有着悠久历史和广泛群众基础的智力活动项目。计算机游戏对设备要求相对简单、所需空间较小、游戏乐趣无穷，因而在其发展过程中吸引了越来越多的群众。适当地游戏丰富了人们的业余生活，锻炼了人们的智力，促进了朋友之间的交流，是一类健康有益的活动。

游戏类型，通常用于分类电子游戏，是一种分辨游戏之间区别的方法。[世界](https://baike.baidu.com/item/%E4%B8%96%E7%95%8C/24458)上并没有统一的分类方法。游戏类型主要分为6类：动作、冒险、模拟、角色扮演、休闲和其他，他们各有几十种分支，形成了庞大的“游戏类型树”。

当选择一种游戏类型进行开发时，经常遇到的一个需求就是让角色从某一个点A移动到另一个点B，常用的方法是两点之间直线距离最短，可以利用勾股定理求最短的直线距离。但是游戏中往往会有障碍物阻挡在直线路径上，这时就需要设计人员利用算法寻找出最优路径了。

寻路问题是游戏中经常遇到的，选择一种合适的寻路算法不仅可以提高用户的游戏体验，也可以改善程序的运行效率。

1.2开发目的

1.利用A\*算法实现一款角色寻路软件。

2.学习常用寻路算法的基本原理、算法，掌握路径问题的的基本建模方案。

3.深入学习Java编程与图形化界面的结合，将抽象的寻路问题转化为可视化的图形界面，实现抽象与现实的结合，数字到图形的转化，展示Java编程语言的魅力。

4.利用计算机强大的计算能力和准确的数据处理能力，将A\*算法的原理和思想充分展现出来，充分发挥计算机的优势。

5.通过开发寻路软件学会分析寻路过程中的各类影响因素，包含主观支配因素和客观决定因素，并且掌握如何进行大规模测试、进行算法改进和测试分析。

1.3开发环境

•操作系统：Windows 10

•开发语言：Java

•开发平台：Eclipse

2软件总体设计

2.1软件总体框架

软件主要分为以下6种模块：

1. 数据结构模块
2. 障碍设置模块
3. 移动规则模块
4. 估值量化模块
5. A\*算法模块
6. 数据存储模块
7. 图形界面模块

2.2软件总体框架说明

1. 数据结构模块

数据结构模块包含了该程序中使用到的角色，软件地图等数据集合，数据结构模块是整个程序的基础，它决定了其他模块的构建原理。比如，角色的颜色属性是直观上判断移动角色、目标角色、静态障碍物、动态障碍物的基础，角色的横纵坐标确定了角色在地图中的位置，角色的其他属性标示着每个角色的独一无二。数据结构模块的确立为其他模块的构建奠定了坚实的基础。

1. 障碍设置模块

障碍设置模块确定了地图中的两种障碍，静态障碍和动态障碍。静态障碍是固定的，只要设置好就不会移动；动态障碍不是固定的，可以移动，并且移动的方向速度都是不确定的，动态障碍每次移动的方向和速度由随机函数决定。障碍设置模块是为了提高A\*算法的准确性和高效性而设置的，同时可以观察影响因素变化时，模型的稳定性。

1. 移动规则模块

角色在移动过程中如果周围区域是无障碍物地带，那么角色可以按照搜索到的路径畅行无阻，但是如果周围区域有障碍物时，角色的行动路径并不是简单的避开障碍物而已，对于静态障碍物，角色的行进路线要保证不会触碰到障碍物；对于动态障碍物，考虑到其移动速度和移动方向的不确定性，角色的行进路线要保证始终处于安全范围之内。

1. 估值量化模块

估值量化模块主要的作用是确定初始[状态](https://baike.baidu.com/item/%E7%8A%B6%E6%80%81/5809342)经由状态n到目标状态的代价估计，尤其是A\*算法使用启发函数h(n)来获得对于从任意结点n走到目标结点的最小代价的估计,启发函数的选择将会直接影响A\*算法运行的效率和准确性，因此选用一个好的启发函数是非常重要的，这也是这一模块的核心工作。

1. A\*算法模块

A\*算法模块是本系统的核心模块，A\*算法既可以像Dijkstra算法一样搜索最短路径，也可以在简单的情况下和BFS算法一样快，它把Dijkstra算法（靠近初始点的结点）和BFS算法（靠近目标点的结点）的信息块结合起来。在讨论A\*算法的标准术语中，g(n)表示从初始结点到任意结点n的代价，h(n)表示从结点n到目标点的启发式评估代价（heuristic estimated cost）。当从初始点向目标点移动时，A\*算法权衡这两者。每次进行主循环时，它检查f(n)最小的结点n，其中f(n) = g(n) + h(n)。

1. 数据存储模块

数据存储模块主要的作用是帮助系统保存当前运行过程中地图障碍物的设置，方便下次再次运行软件时可以直接获得障碍物信息。本软件编写的目标之一是寻找高效的建模方式和最适当的启发函数。数据存储模块的存在可以节省很多重新恢复地图的时间，对于整个实验过程有重要意义。

1. 图形界面模块

图形界面模块利用Eclipse图形界面用户程序开发框架，将Java编程与图形化界面相结合，将抽象的数据结构转化为可视化的图形界面，实现抽象与现实的结合，数字到图形的转化，展示了Java编程语言的魅力。图形界面模块是软件用户最直观感受到的模块，帮助用户感受到A\*算法在实现路径搜索过程中扮演的重要角色。

3软件设计方案

3.1软件功能介绍

基于A\*算法的角色寻路软件主要实现的功能是模拟流行游戏中人物角色的寻路过程，软件中，地图由60\*40的方格组成，地图中人物角色用蓝色方块模拟，目标点用绿色方块表示，静态障碍物用黄色方块表示，动态障碍物用黑色方块表示。

本软件提供了动态搜索路径和静态搜索路径过程。在静态搜索路径的过程中，地图中只存在静态障碍物，从角色当前位置到目标位置只需要一次寻路过程即可，因为地图是静态的，所以角色在移动过程中，只要能确保搜索的路径是最优的，就不会出现因为意外情况而发生当前路径不合适的情况，软件启动后默认的搜索方式是静态搜索，用户可以点击开始按钮启动角色的寻路过程，寻路开始后，角色会根据搜索到的路径向目标位置移动，在角色移动过程中用户可以右键点击任意位置为角色改变方向，鼠标左键点击可以为地图添加静态障碍物；在动态搜索过程中，地图中不仅会存在静态障碍物，还会有动态障碍物，鼠标左键点击可以为地图添加静态障碍物，通过按钮可以为地图添加动态障碍物，由于动态障碍物的活动方向和速度都是不确定的，所以无法预测动态障碍物下一刻的位置，因此从角色当前位置到目标位置只使用一次寻路过程是远远不够的，角色每走一步都需要搜索当前环境下到达目标位置的最优路径，而且只能选择最优路径中离当前位置最近的下一个地点作为当前角色要移动的位置，移动完成后需要再次启动寻路过程，重复以上过程直至到达目标地点。

使用软件的过程中，用户通过菜单栏中的保存选项可以保存当前地图中障碍物在地图中的分布，下次使用软件时可以直接打开之前保存的障碍物分布图，可以节省用户重新布置障碍物的时间。

3.2软件各模块原理说明

1.数据结构表示（结点和地图）

根据前面对角色寻路功能软件的介绍，结点是地图中的基本元素，结点首先应该具有坐标（x，y）,坐标决定了结点在地图中位置，其次软件的算法基础是A\*算法，所以每个结点都内部都应该具有计算源点经过自己到目标位置的代价，根据A\*算法的基本公式，确定g（从起点source移动到当前点的耗费）、h（从当前点到终点的估值耗费）、f（总代价，f = g + h）。

地图是A\*算法实现的界面，地图中首先包含了一个角色起始点source和目标点target，这两个的本质是前面提到的结点。其次地图中应该包含有各个位置的状态，我们用一个二维数组astarData代替，二维数组中每一个元素的取值决定了当前位置的状态，如果值为0，那么当前位置为空，角色可以在此位置上移动；如果值为2，那么当前位置为静态障碍物，角色无法经过此位置；如果值为3，那么当前位置为动态障碍物，角色无法经过此位置。为了维护软件的安全性，我们还为地图添加了动态障碍物的最大数maxbcount,并且有bcount记录当前动态障碍物的数量，bax[]、bay[]分别记录每个动态障碍物的当前位置。最后，为了保存A\*算法执行过程中需要保存的数据，我们加入了openList和closeMap 两个数据集合，为了保存搜索到的最佳路径，我们加入了ArrayList类型的数据集合path。

1. 障碍物设置

障碍物设置包括静态障碍物设置和动态障碍物设置。静态障碍物的设置比较简单，只要将地图中相应位置的值设为2即可，这里主要介绍一下动态障碍物的设置。动态障碍物的初始化比较简单，只需要利用随机函数随机选择一个地图上的位置，将值设为3即可，除此之外还要考虑到怎样设置动态障碍物的移动方向和移动速度。

移动过程算法说明如下：

movebarrier()

1 获得随机数发生器

2 For i=0 to bcount-1

3 While flag==1 do

4 //获得0-7内任意一数

5 f=r.nextInt(8)

6 If f==0 then

7 If 当前动态障碍物的上方位置在地图上 then

8 removebarrier(bax[i],bay[i])

9 bax[i]=bax[i]-1

10 setmbarrier(bax[i], bay[i])

11 flag=0

12 Else if f==1 then

13 If当前动态障碍物的上方位置在地图上 then

14 removebarrier(bax[i],bay[i])

15 bax[i]=bax[i]+1

16 setmbarrier(bax[i], bay[i])

17 flag=0

18 Else if f==2 then

19 If当前动态障碍物的左方位置在地图上 then

20 removebarrier(bax[i],bay[i])

21 bay[i]=bay[i]-1

22 setmbarrier(bax[i], bay[i])

23 flag=0

24 ......

25 flag=1

判断棋子走法是否符合规则部分主要根据前面介绍的爱因斯坦棋走法规则。

具体代码如下：

在地图类中，我们设置了bax[]、bay[]数组分别存储了每个动态障碍物的位置，在控制动态障碍物移动过程中，通过生成随机数，为每一个动态障碍物生成下一个运动位置，上述算法仅给出了当随机数为0、1、2时的移动情况，若随机数为其他3、4、5、6、7，则分别在判定当前动态障碍物的右方、左上方、右上方、左下方、右下方位置在地图上后，便可以移动动态障碍物到相应的相邻位置，若相邻位置不在地图上，会继续生成随机数，为动态障碍物寻找另一个合法的相邻位置移动。

在解决动态障碍物移动速度随机变化的问题时，我们依靠的方法是线程，在图形界面模块，我们建立了一个线程用来时刻重新绘制图形界面，我们在这个线程中加入了动态障碍物的移动过程如下：

startAnimatorThread()

1 new Thread(){

2 public void run()

3 try

4 While true do

5 Thread.sleep(SPF)

6 //界面重绘函数

7 repaint()

8 //实时速度增加

1. speed++

10 //判断实时速度是否达到目标速度

11 If speed==mspeed then

12 astarMap.movebarrier()

13 //实时速度清0

14 speed=0

15 //随机数生成器

16 r=new Random();

1. //生成新的目标速度

18 mspeed=r.nextInt(5)+5;

19 //异常处理

20 catch (InterruptedException e)

21 e.printStackTrace()

22 }.start();

3.移动规则

角色在移动过程中，并非是沿着空白地带就可以上下左右斜方向移动，如果附近是静态障碍物的情况，如图1。

在下图中，蓝色方块代表角色的当前位置，黄色方块是静态障碍物，图中蓝色方块可以向上、下、左、左上、左下移动，但是右上、右下是不可以移动的。因为若角色沿直线移动到这两个位置势必会触碰到静态障碍物，所以这两个方向是不可达区域。



图1.静态移动规则

如果附近区域是动态障碍物，如图2，黑色方块为动态障碍物，考虑到动态障碍物的移动方向和移动速度都是不可预知的，所以角色为了保证自己的安全性，必须时刻与动态障碍物保持一定的隔离距离，经过计算，我们选择了当前动态障碍物方圆2个单位范围内的区域为危险区域。

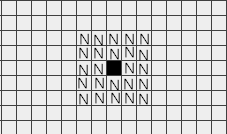


图2.动态移动规则

判断下一步位置是否符合静态移动规则的算法流程图如图3。

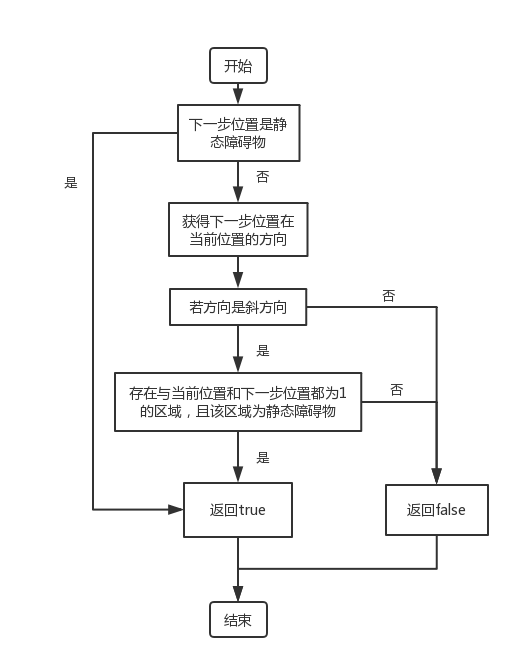


图3.下一步位置的静态判断

下一步位置的动态判断如下所示。

isCanNotGo1(AStarNode node)

1. flag=false
2. 遍历当前结点方圆2个单位内的所有位置astarData[i][j]

3 //判断该位置是否有静态障碍物

4 If STATE\_MBARRIER==astarData[i][j] then

5 flag=true

6 return flag

1. 估值量化

A\*算法的估算函数为f(n)=g(n)+h(n)，其中g(n)表示从起点A到当前点n实际走的代价即移动代价，h(n)表示当前点n到终点的估算代价。 所以上面两个合起来就是起点到当前点到终点的总代价函数f(n)。

在本软件中，横向和纵向的移动代价为10，对角线的移动代价为14。之所以使用这些数据，是因为实际的对角移动距离是2的平方根，或者是近似的1.414倍的横向或纵向移动代价。使用10和14就是为了简单起见。比例是对的，我们避免了开放和小数的计算。这并不是我们没有这个能力或是不喜欢数学。使用这些数字也可以使计算机运行更快。计算出该方格的g值的方法就是找出其父结点位置的g值，然后按其父结点是直线方向还是斜线方向加上10或14。

而对h(n)这个估计函数不同的估计情况，结果也不会相同。先考虑极端情况，在h(n)=0的情况下，只有g(n)起作用，那么A\*算法就是Dijkstra算法。如果h(n)始终小于等于实际n点到终点的距离，那么必然能够保证A\*算法找到的解就是最优解。而且h(n)越小，A\*算法扩展的节点也就越多，A\*算法运行的也就越慢。如果h(n)始终都等于实际n点到终点的距离，那么A\*算法只会严格走从起点到终点的最短路径。虽然这种情况一般不可能发生，当然一些特殊情况下会发生（比如没有障碍物的情况）。如果h(n)有时候大于实际n点到终点的距离，那么不能保证A\*算法能够找到最短路径。另外一种极端情况是只有h(n)发挥作用，那么A\*算法就相当于贪心算法。所以h(n)的选择成了一个有趣的情况，它取决于我们想要A\* 算法中获得什么结果。h(n)合适的时候，我们会非常快速地得到最短路径。如果h(n)估计的代价太低，我们仍会得到最短路径，但运行速度会减慢。如果估计的代价太高，我们就放弃最短路径，但A\* 算法将运行得更快。

A\* 算法基于启发式函数和代价函数来改变其行为的能力在游戏中非常有用。速度和准确性之间的折衷可以提高游戏速度。对于大多数游戏而言，并不需要两个点之间的最佳路径。只需要知道近似的路径就足够了。你所需要的路径往往取决于游戏中接下来要发生什么，或是运行游戏的计算机有多快。

假设游戏中有两种地形，平原和山地，它们的移动代价分别是1和3，A\* 算法沿着平原搜索的路径长度是沿着山区的三倍。这是因为可能有一条绕着山地的平原路径。可以把两个地图单位之间的启发式距离设为1.5可以加快A\* 算法的搜索速度。于是A\* 算法会将山区的移动成本3改为1.5，这个变化不像3到1那么大。这种方法在山区的移动成本不像之前那样高，因此不用花太多的时间去寻找绕着山地的路径。或者，可以通过告诉A\* 算法在山区的移动成本为2而不是3，以减少山区周围路径的搜索量，来加快A\* 算法的搜索速度。现在，沿着平原搜索路径的速度只是沿着山区的两倍。这两种方法都放弃了理想路径来获得更快的搜索速度。

速度和准确性之间的权衡不需要是固定的。可以根据CPU速率、用于寻路的时间片数、地图上物体的数量、物体的重要性、组(group)的大小、难度级别，或其他任何因素来进行动态地选择。一种动态的折衷启发式函数方法是，假设通过一个网格空间的最小代价为1，然后建立一个在下式范围内的代价函数：

g'(n) = 1 + alpha \* (g(n) - 1)。

如果alpha值为0，则修改后的代价函数的值将总是为1。在这种情况下，地形代价被完全忽略，A\* 算法的工作变成了简单判断一个网格能否通过。如果alpha的值为1，则初始代价函数将被使用，将会得到A\* 算法的所有优点。可以将alpha设为0到1之间的任意值。

也应该考虑在启发式函数返回的绝对最小代价和期望最小代价中做选择。例如，如果地图上大部分地形是移动代价为2的草地而一些地形是移动代价为1的道路，那么可以考虑让启发式函数假设没有道路，而只返回两倍的距离。

速度和准确性之间的选择并不必是全局的。在地图上的某些区域，可以基于其准确性的重要性来进行动态选择。举个例子，假设我们在任意点都可能停止并重新计算路径或改变方向，那么为什么要困扰于后续路径的准确性呢？在这种情况下快速选择一条的路径更加重要。或者，对于地图上的某个安全区域，准确的最短路径并不那么重要；但在渡过危险区域时，安全和准确是必需的。构造精确的启发式函数的一种方法是预先计算每对结点之间的最短路径的长度。这种做法对于大多数游戏的地图而言并不可行。但是，有几种方法可以近似模拟启发式函数：

在细网格拟合合适密度的粗网格。 预先计算粗网格中任何一对结点之间的最短路径。 预先计算任何一对路径点(waypoints)之间的最短路径。(这是粗网格方法的一般化。)然后添加一个启发式函数h’来估计从任何位置到其邻近路径点的代价。(如果需要，后者也可以通过预计算得到。)最终的启发式函数将是：

h(n) = h'(n, w1) + distance(w1, w2) + h'(w2, goal)。

在特殊情况下，不需要预先计算也能使启发式函数很精确。如果你的地图没有障碍物或者移动缓慢的区域，那么从初始点到目标点的最短路径应该是一条直线。

或者，想要一个更好但代价更大的启发式函数，则分别用靠近结点和靠近目标的所有w1, w2对上式进行计算。

在网格地图中，有一些众所周知的启发式函数可供使用。启发式函数的距离与所允许的移动方式相匹配：

在正方形网格中，允许向4邻域的移动，使用曼哈顿距离(L1)。   
 在正方形网格中，允许向8邻域的移动，使用对角线距离(L∞)。   
 在正方形网格中，允许任何方向的移动，欧几里得距离(L2)可能适合，但也可能不适合。如果用A\* 算法在网格上寻找路径，但又不允许在网格上移动，可能要考虑用其它形式表现该地图。   
 在六边形网格中，允许6个方向的移动，使用适合于六边形网格的曼哈顿距离。

对于方形网格，标准的启发式函数就是曼哈顿距离。考虑一下的代价函数并确定从一个位置移动到相邻位置的最小代价D。在简单的情况下，可以将D设为1。在一个可以向4个方向移动的方向网格中，启发式函数是曼哈顿距离的D倍。

function heuristic(node) =

1. // dx

2 = abs(node.x - goal.x)

1. // dy

4 = abs(node.y - goal.y)

5 //return

6 D \* (dx + dy)

对于最佳路径，和“可采纳的”的启发式函数，应该将D设为邻近方格间移动的最低代价值。在一个没有障碍物、最小移动代价为D的地形上，每向目标靠近移动一步，g就增加D的移动代价同时h减少D的代价。此时将g和h相加时，f保持不变；这是启发式函数与代价函数的衡量单位相匹配的一个标识。也可以通过放弃最优路径增加代价D或是降低最低和最高边际代价之间比率的手段，来让A\* 的运行速度更快。 如下图所示。

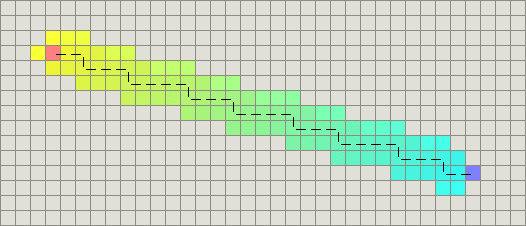


图4.启发示例

如果在地图中允许沿着对角线移动，那么需要一个不同的启发式函数（有时被称为契比雪夫距离(Chebyshev distance)）。偏东4个单位偏北4各单位(4 east, 4 north)的曼哈顿距离是8\*D。然而，对对角线距离而言，可以简单地移动4个对角线长度，因此启发式函数将为4\*D。

本软件中使用Manhattan方法，计算从当前方格横向或纵向移动到达目标所经过的方格数，忽略对角移动，然后把总数乘以10。之所以叫做Manhattan方法，是因为这很像统计从一个地点到另一个地点所穿过的街区数，而不能斜向穿过街区。重要的是，计算H是，要忽略路径中的障碍物。这是对剩余距离的估算值，而不是实际值，因此才称为试探法。

把G和H相加便得到F。我们第一步的结果如图5所示。每个方格都标上了F，G，H的值，就像起点右边的方格那样，左上角是F，左下角是G，右下角是H。

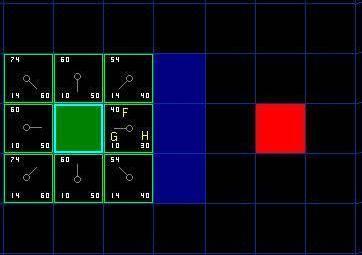


图5.计算F示意图

现在让我们看看其中的一些方格。在标有字母的方格，G = 10。这是因为水平方向从起点到那里只有一个方格的距离。与起点直接相邻的上方，下方，左方的方格的G值都是10，对角线的方格G值都是14。H值通过估算起点与终点(红色方格)的Manhattan距离得到，仅作横向和纵向移动，并且忽略沿途的墙壁。使用这种方式，起点右边的方格到终点有3个方格的距离，因此H = 30。这个方格上方的方格到终点有4个方格的距离(注意这里只计算横向和纵向距离)，因此H = 40。对于其他的方格，可以用同样的方法知道H值是如何得来的。每个方格的F值，就是直接把G值和H值相加就可以了。

1. A\*算法

A\*算法是最好的优先算法的一种。最好优先算法的寻路过程如图6的状态空间：（起始位置是A，目标位置是P，字母后的数字表示节点的估价值）。

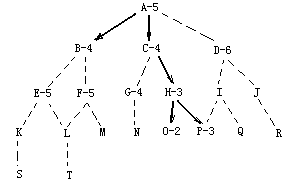


图6.优先算法图示

搜索过程中设置两个表：OPEN和CLOSED。OPEN表保存了所有已生成而未考察的节点，CLOSED表中记录已访问过的节点。算法中有一步是根据估价函数重排OPEN表。这样循环中的每一步只考虑OPEN表中状态最好的节点。具体搜索过程如下：

1. 初始状态：　　　　　　　　　　　　　　　　  
   　 OPEN=[A5]；CLOSED=[]；  
    2）估算A5，取得搜有子节点，并放入OPEN表中；  
   　 OPEN=[B4，C4，D6]；CLOSED=[A5]  
    3）估算B4，取得搜有子节点，并放入OPEN表中；  
   　 OPEN=[C4，E5，F5，D6]；CLOSED=[B4，A5]  
    4）估算C4；取得搜有子节点，并放入OPEN表中；  
   　 OPEN=[H3，G4，E5，F5，D6]；CLOSED=[C4，B4，A5]  
    5）估算H3，取得搜有子节点，并放入OPEN表中；  
   　 OPEN=[O2，P3，G4，E5，F5，D6]；CLOSED=[H3，C4，B4，A5]  
    6）估算O2，取得搜有子节点，并放入OPEN表中；  
   　 OPEN=[P3，G4，E5，F5，D6]；CLOSED=[O2，H3，C4，B4，A5]  
    7）估算P3，已得到解；

A\*算法的算法流程与上面所说的类似，流程如下：

把起始格添加到开启列表，然后重复如下的工作：  
 a) 寻找开启列表中F值最低的格子。我们称它为当前格。  
 b) 把它切换到关闭列表。  
 c) 对相邻的格中的每一个格子，如果它不可通过或者已经在关闭列表中，略过它。反之如下。如果它不在开启列表中，把它添加进去。把当前格作为这一格的父结点。记录这一格的F,G和H值。如果它已经在开启列表中，用G值为参考检查新的路径是否更好。更低的G值意味着更好的路径。如果是这样，就把这一格的父结点改成当前格，并且重新计算这一格的G和F值。如果保持开启列表按F值排序，改变之后可能需要重新对开启列表排序。如果它已经在关闭列表中，可能该点G值比旧点的更低，如果是这样，就把这一格的父结点改成当前格，并且重新计算这一格的G和F值。删除关闭列表的旧点，添加新点到开始列表中，如果保持开启列表按F值排序，改变之后可能需要重新对开启列表排序。(此步骤有时不需要)  
 d) 停止。当你把目标格添加进了关闭列表(注解)，这时候路径被找到，或者没有找到目标格，开启列表已经空了。这时候，路径不存在。保存路径。从目标格开始，沿着每一格的父结点移动直到回到起始格。这就是要找的路径。

最好最优算法的伪代码如下：

Best\_First\_Search()  
1 　Open = [起始节点]  
2 　Closed = []  
3 　while Open表非空 do  
4 　 从Open中取得一个节点X，并从OPEN表中删除。  
5 　 If X是目标节点 then  
6 求得路径PATH  
7　　　 返回路径PATH  
8 For 每一个X的子节点Y then  
9 　　 If Y不在OPEN表和CLOSE表中 then  
10 求Y的估价值  
11 　　　　并将Y插入OPEN表中  
12　　　 //还没有排序  
13　　 Else if Y在OPEN表中 then  
14 If Y的估价值小于OPEN表的估价值 then  
15　　　　　 更新OPEN表中的估价值  
16　　　 Else //Y在CLOSE表中  
17 If Y的估价值小于CLOSE表的估价值 then  
18 更新CLOSE表中的估价值；  
19　　　　　 从CLOSE表中移出节点，并放入OPEN表中  
20 将X节点插入CLOSE表中  
21 按照估价值将OPEN表中的节点排序

在下面这个例子中g(n)表示在状态空间从起始节点到n节点的深度，h(n)表示n节点所在地图的位置到目标位置的直线距离。有一个物体A，在地图上的坐标是(xa,ya)，A所要到达的目标b的坐标是(xb,yb)。则开始搜索时，设置一个起始节点1，生成八个子节点2- 9 (因为有八个方向)。如图7：

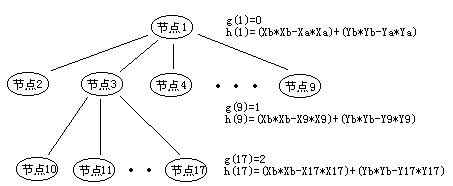


图7.A\*算法寻路过程

上面的例子是一个典型的g、h函数计算过程，唯一的区别是上述过程中计算h的方法不是Manhattan方法，本软件中采用的是Manhattan方法。下面给出A\*算法的流程图。

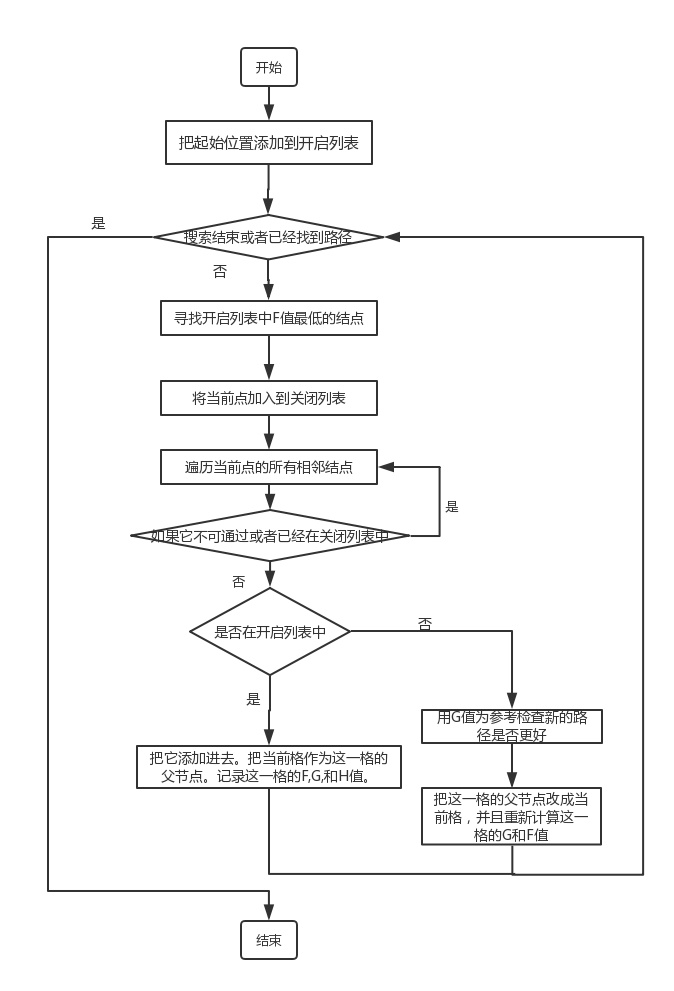


图8.静态A\*算法流程图

上面的流程图是标准的A\*算法寻路过程，也是本软件中静态地图下使用的算法。但是软件中提供了动态障碍物，拥有动态障碍物的地图不再是静态的，传统的A\*算法需要作出相应的更改以适应新的情形。动态障碍物的特点是移动方向不可预测，移动速度不可预测，因此在寻路过程中不能简单地以邻居结点是否为动态障碍物作为躲避危险的唯一依据。经过多次测试后，我们可以认为动态障碍物的方圆2个单位内的区域为危险区域，处在这一区域的结点随时有可能触碰到动态障碍物，所以使用A\*算法寻路的过程中要使移动路线随时保持在危险区域以外，因此对上述算法流程图稍作修改如图9。

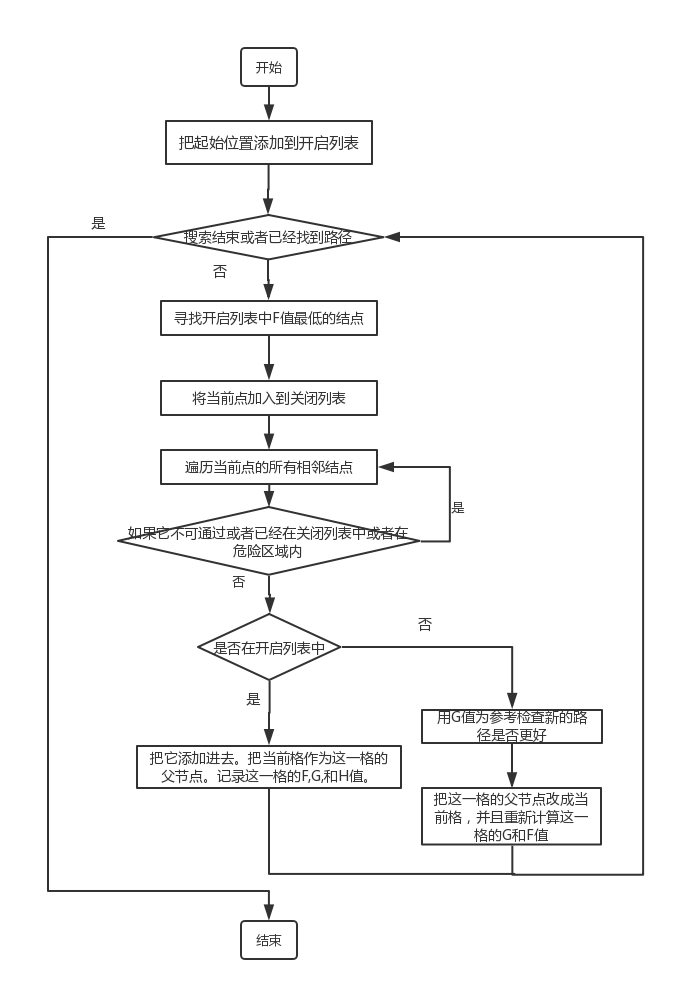


图9.动态A\*算法流程图

1. 数据存储

数据存储模块的作用是保存当前地图中静态障碍物的位置，不保存动态障碍物的原因是动态障碍物具有不固定性，这种随机性应该每时每刻都得到体现，如果保存了动态障碍物的位置，那么这种随机性就会收到破坏，由于本软件中动态障碍物设置的上限是10，所以重新设置动态障碍物基本不会浪费时间，不会影响软件的使用体验和运行效率。

数据存储部分包括数据文件的保存和导入，主要用到的类包括JFileChooser、File、FileWriter、BufferedWriter、StringBuilder等，下图给出了数据文件保存的算法流程。

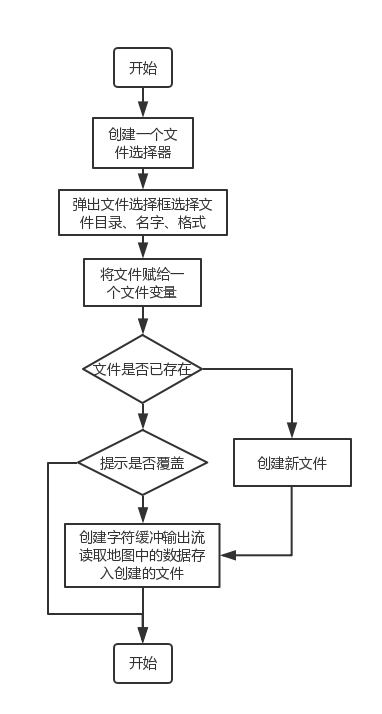


图10.地图数据存储过程

上述过程仅仅是存储地图数据的主要流程，具体实现的过程中还需要考虑几个问题，如果文件已经存在，当前文件是否支持重新写入？如果文件不存在，那么操作系统是否支持创建新文件？怎样将地图中的数据转化为规范地读入文件？前面两个问题只需要在处理流程中加入判断语句，如果文件不支持重新写入或者操作系统系统不支持创建新文件时直接退出文件操作即可。最后一个问题的解决方法的伪代码如下：

1 For i=0 to drawData.length-1 then

2 sb = new StringBuilder()

3 For j=0 to drawData[i].length-1 then

4 if(j!=0)

5 sb.append(DATA\_DELEMITER)//加入分隔符“，”

6 sb.append(drawData[i][j])

7 If i<drawData.length-1 then

8 sb.append(LINE\_DELEMITER)//加入行分隔符“；”

9 //将字符串写入文件

10 bwriter.write(sb.toString())

下面给出数据导入的流程图。

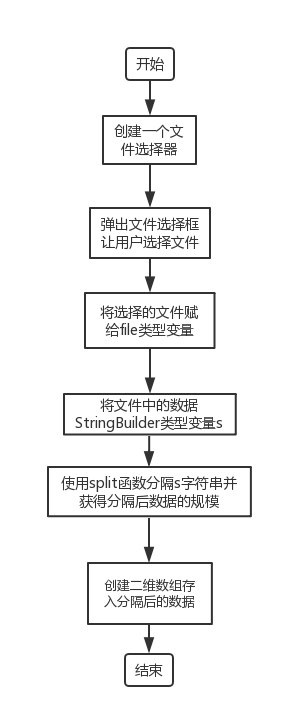


图11.数据导入流程

1. 图形界面

程序开始创建一个JFrame窗体，然后向JFrame类添加三个面板容器类，AstarPanel（模拟器主要运行界面容器，位置居中）；StatusPanel（用于展示搜索时间的容器，位置在最下方）；ControlPanel（控制按钮包含容器，位置在最上方）；AStarMenuBar（菜单栏）。

下面简要介绍各个面板容器的内容：

AstarPanel：

这个容器的主要作用是将算法计算出的结果用图形界面表示出来，其中，任务角色用一个蓝色矩形表示，，目的地点用浅绿色矩形表示，障碍物用黄色矩形表示，角色走过的路径用白色矩形表示，容器中设置鼠标事件监听器，左键用来设置障碍物，左键点击的位置会被设置为障碍物，目标点不允许通过，鼠标右键点击的位置会刷新目标点。

StatusPanel：

这个容器主要包含了JTextField文本框，文本框中用来显示搜索耗时，搜索耗时通过监测搜索结束时的系统时间与搜索前的系统时间。两者做差，即为搜索耗时。

ControlPanel：

这个容器主要包含了五个按钮，点击“开始”按钮，程序开始搜索角色移动到目标点的最佳路径，并且开始逐步移动到目的位置，移动过程中的路径在图中用白色方块表示；点击“清除”按钮，图中的白色路径将不再显示，只显示角色点的当前位置；点击“清理地图”按钮，图中的静态障碍物全部消失，图中将不再存在障碍物；点击“增加障碍物”按钮，图中将添加动态障碍物，寻路方式切换到动态寻路，每点击一次，增加一个动态障碍物，动态障碍物数量的上限是10个；点击“清理障碍物”图中的动态障碍物全部消失，寻路方式切换到静态寻路。

AStarMenuBar：

菜单栏中包括两个，一个是文件操作，文件操作中包括保存地图，打开地图，以及退出程序。保存地图利用的原理是利用Java自带的文件函数读取当前地图中所有位置的状态，将转态存储在一个数据文件中；打开地图利用的原理是打开之前保存的数据文件（如果文件格式有问题会报错，文件内容不对应也会报错），然后当前地图根据数据文件中各点的信息，刷新地图的各个点；退出程序选项则会彻底关闭程序。另一个菜单是用来展示程序制作信息的，主要用到了JDialog展示项目信息。

软件具体界面如下图所示：

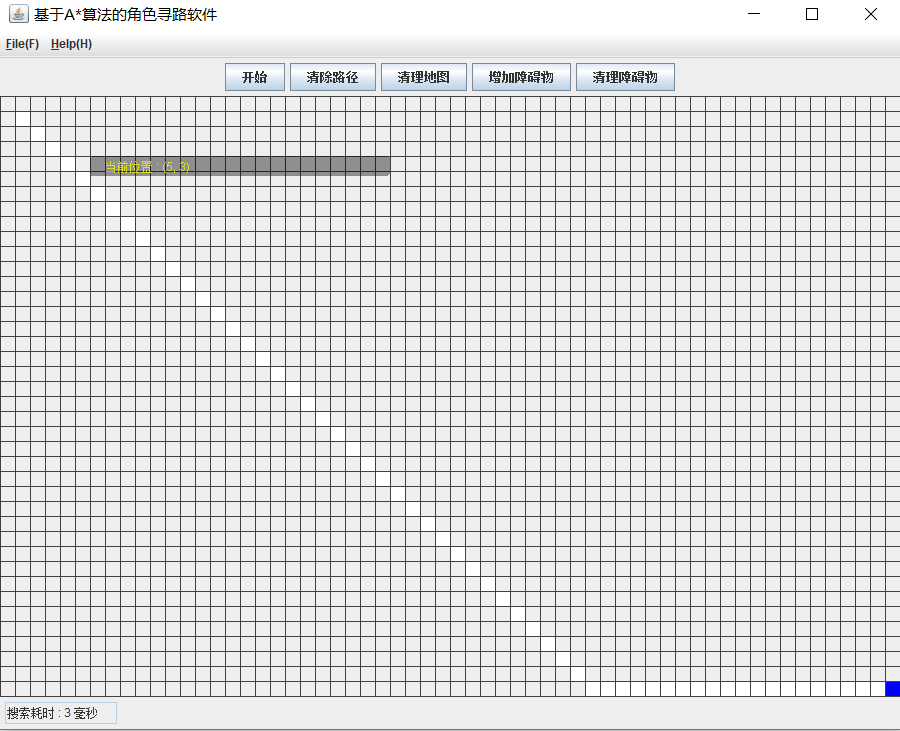


图12.软件界面

4系统测试

4.1系统运行说明

程序经过构建之后，点击运行，弹出软件界面，如下图所示：

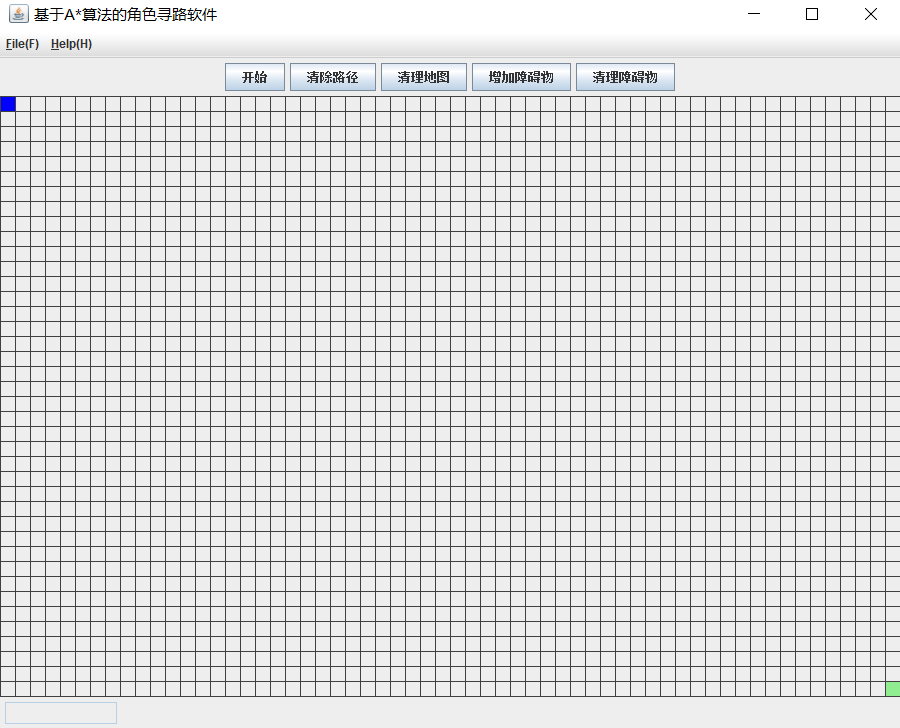


图13.软件初始界面

蓝方方块表示寻路的角色，绿色方块代表目标位置，方块构成的区域为地图，点击开始按钮，蓝色方块开始寻路，此时寻路方式为静态寻路方式，蓝色方块根据搜索到的路径到达目的位置。搜索结果如下图所示：

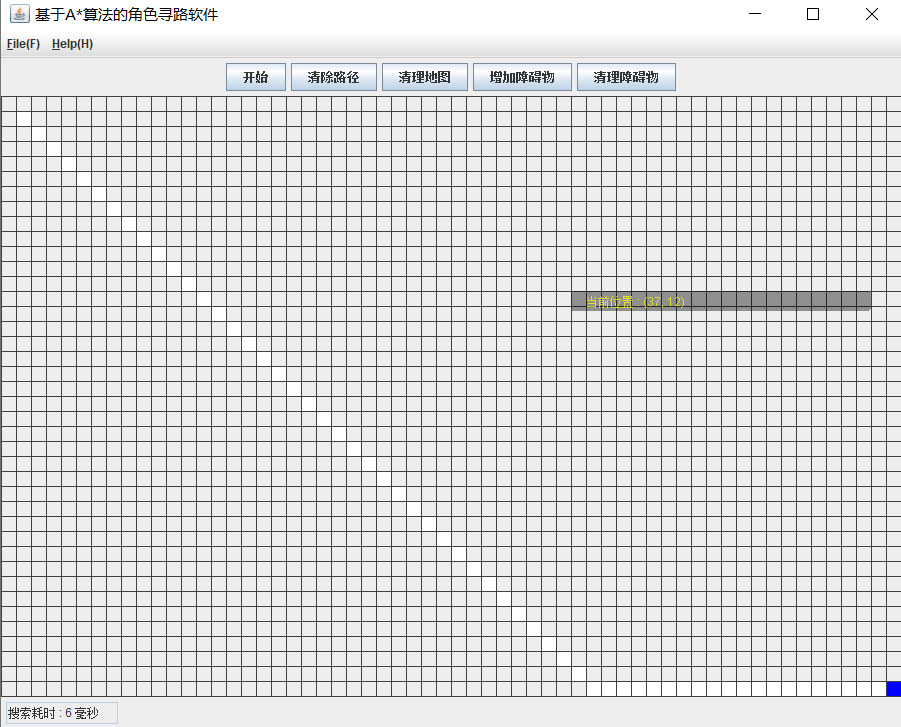


图14.寻路过程

白色区域为方块的移动轨迹，根据手动计算，此路径为最佳路径之一，界面下方显示此次搜索耗时为6毫秒。蓝色方块到达目的位置后，用户可以鼠标右键点击图中任意位置，为蓝色方块指定新的目标位置，此时不需要再次点击“开始”按钮，蓝色方块将自动搜索并且移动到目标位置。此次寻路过程中没有障碍物，若想为地图添加静态障碍物可以选择鼠标左键点击图中任意位置添加，或者鼠标左键按住，在地图中拖动，拖动区域即将会生成静态障碍物。如下图：

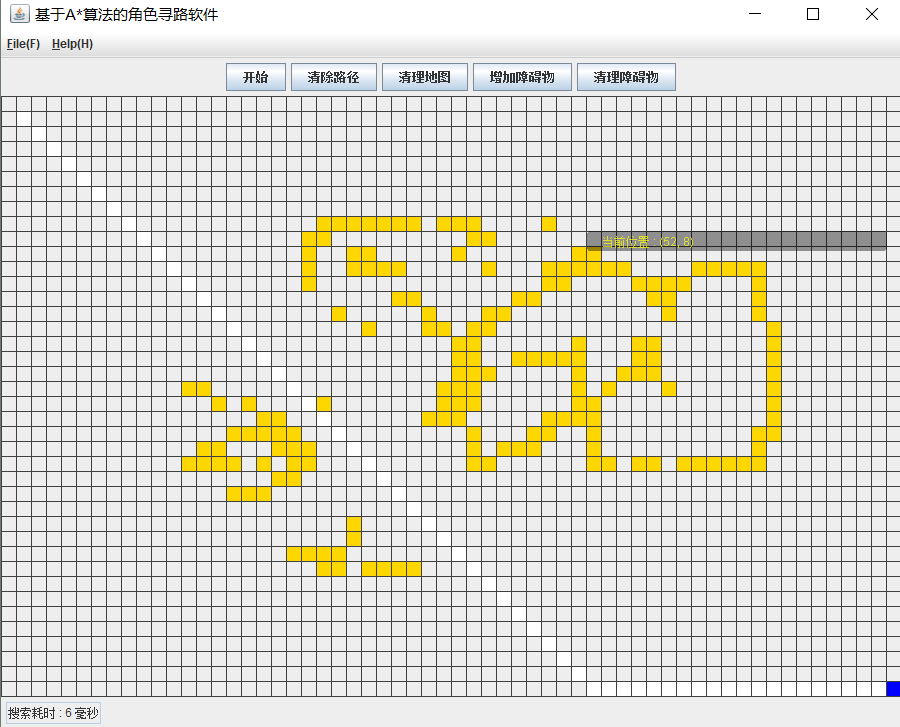


图15.生成静态障碍物

上图中黄色区域就是新添加的静态障碍物，此时鼠标右键点击图中任意位置为蓝色方块再次选择目标位置，蓝色方块将再次寻路并移动到目标位置，如图16所示。

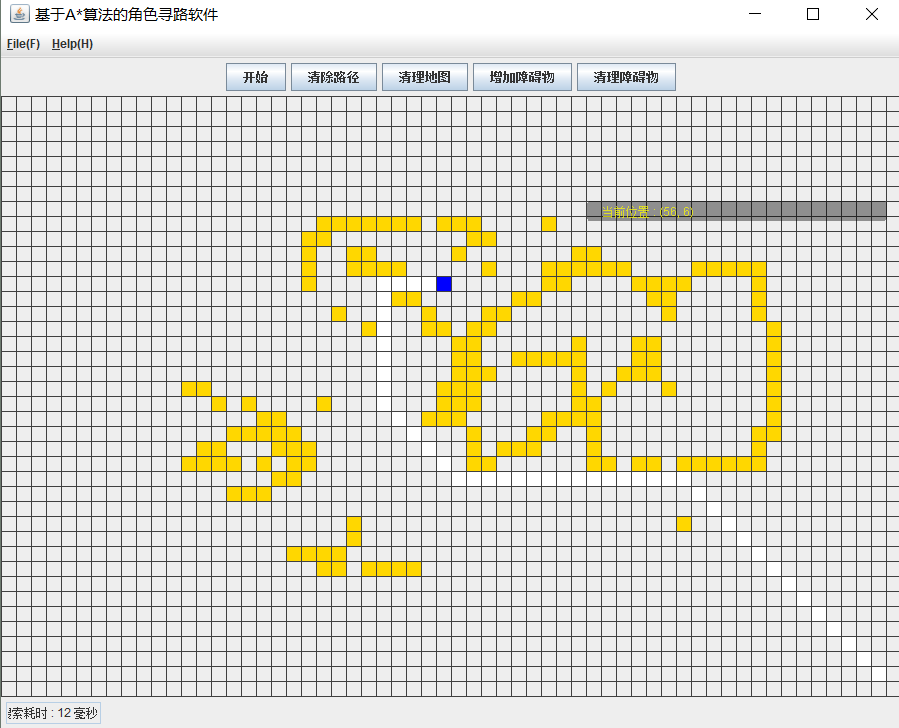


图16.有静态障碍物的寻路过程

白色区域为移动踪迹，经过手动计算，图中踪迹为最佳寻路过程。点击图中的“清除路径”按钮，可以清除图中的白色踪迹，如下图所示：

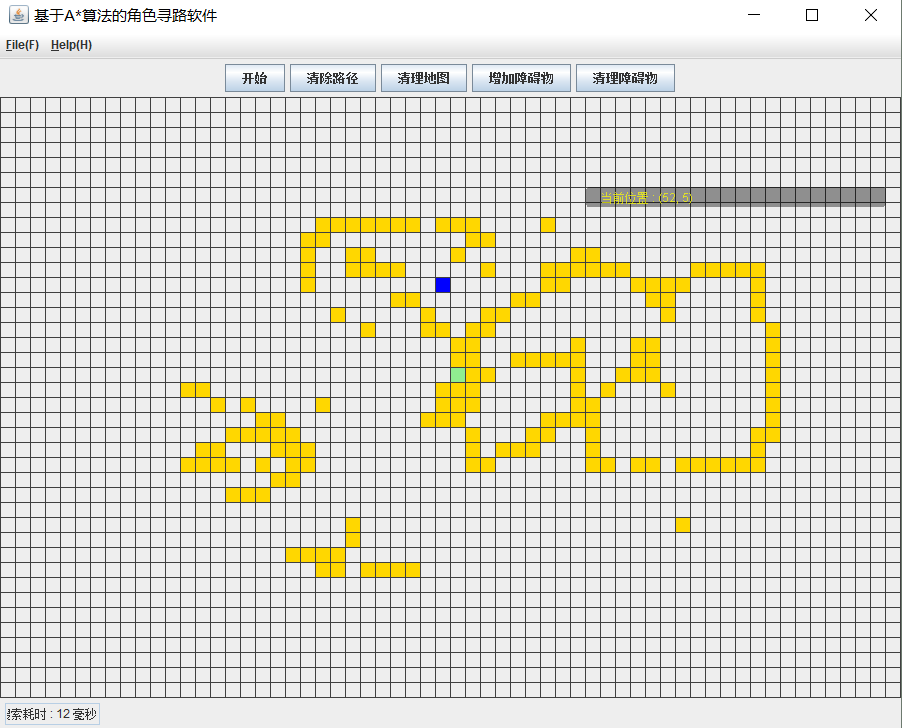


图17.清除路径

如图所示，白色路径已清除完毕，点击“清理地图”按钮，图中的静态障碍物将会全部清除，如下图所示：

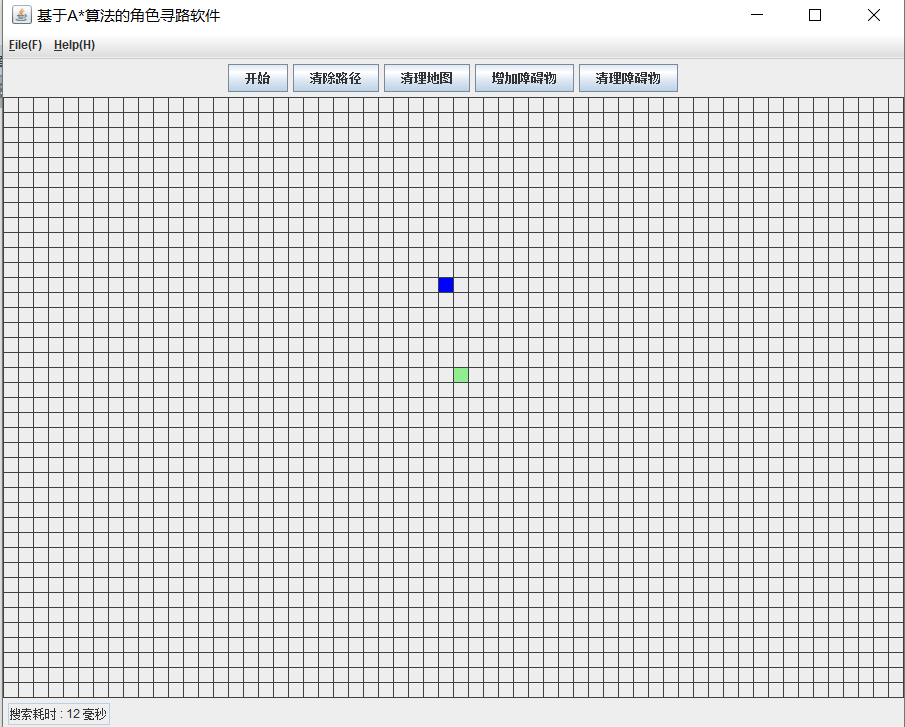


图18.清理地图

上面展示的过程为静态寻路过程，点击“增加障碍物”按钮，可以为地图添加动态障碍物，此后的寻路过程将变为动态寻路过程，每点击一次按钮，添加一个动态障碍物，上限为10个，如下图：

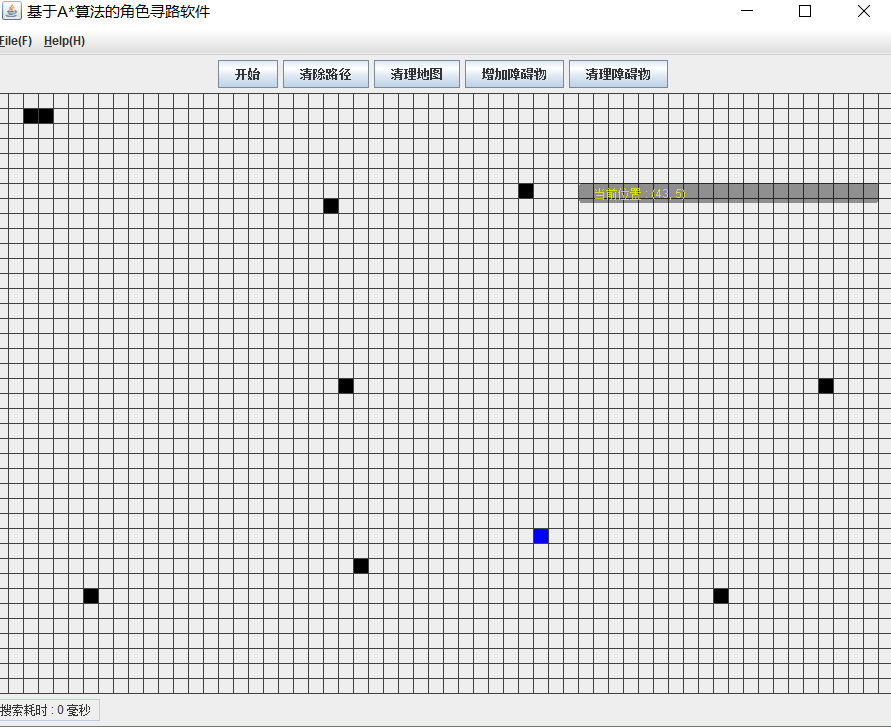


图19.添加动态障碍物后的寻路过程

此时的寻路过程为动态的，每走出一步都要进行一次寻路过程，由图中可以看出，最后一步寻路搜索耗时为0毫秒（四舍五入），经过观察此次移动路径为最优路径。下面展示一次复杂的寻路过程。

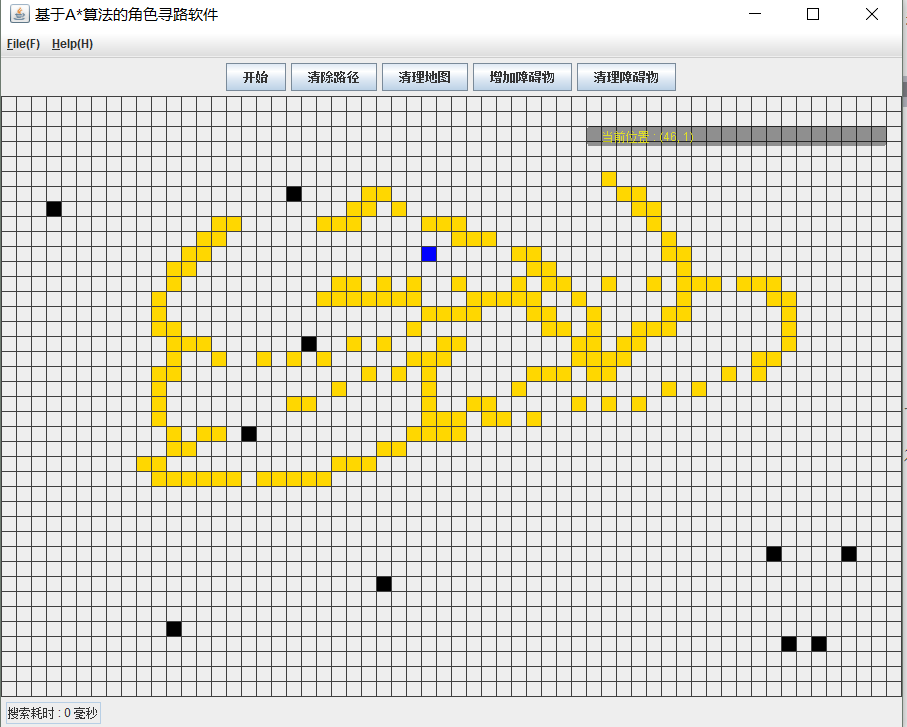


图20.两种障碍物下的寻路结果

上图中存在两种障碍物，此时的寻路过程也为动态的，每走出一步都要进行一次寻路过程，由图中可以看出，最后一步寻路搜索也耗时为0毫秒（四舍五入），因为最后一次搜索过程只需要搜索一步即可，经过观察此次移动路径为最优路径。

菜单栏中包括“File”选项和“Help”选项，第一个选项中包含Load Map、Save Map、Quit三个选项，第二个菜单栏中Tips和About。

选择Load Map或者Save Map选项时会弹出文件选择框，可以选择创建或者打开文件的目录，名字和类型，如下图：

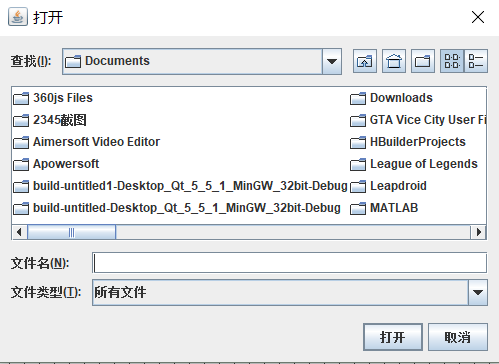


图21.文件选择对话框

点击Quit选项会退出软件，点击“Help”栏中的Tips会出现使用说明，如图22，点击About会出现创作团队介绍，如图23。

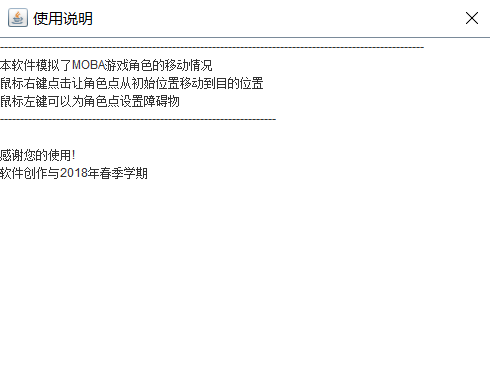


图22.使用说明



图23.设计团队

4.2系统测试总结

基于A\*算法的角色寻路软件的实现符合我们的预期效果，软件运行过程与我们的设计思路一致。经过检测，软件各个功能完美实现，软件运行原理合理，没有出现异常情况。