即时战略游戏中寻径问题的算法及实现技术研究

The Algorithm of Real-Time Strategic-Game Seek-Route Engine Implement Technology Research

赵建民　朱信忠

(浙江师范大学计算机科学研究所　金华321004)

Abstract The Algorithm of Real-Time Strategic-Game Seek Route Engine Implement Technology, especially the huge network PC game Path-finding. Our game laboratoy is in process of a study of the performance difference of "Red Alert" and "Age". We feel that we should bring forword our viewpoints. The technology of creating running-circuit in the circumstance of barrier is discussed in this paper.

Keywords Real-Time Strategic-Game, Seek Route Engine, PC Game, DirectX

1　概论

游戏可以放松身心让生活丰富多彩，好的游戏还可陶冶情操，增进友谊，尤其象"红色警报"、"帝国时代"、"星际争霸"这样的大牌网络即时战略游戏，可以让人们体验到一种指挥千军万马纵横沙场的英雄意境。我们计算机科学研究所一直就想探索这类即时战略游戏背后隐藏的编程奥秘。经过一段时间的研究，终于摸索到了即时战略游戏中寻径(Path-finding)问题的一些简单算法，并将此算法用具体程序实现，用 Visual C^{++} 6.0编写了一个即时战略游戏的雏形，现将我们的部分研究成果及设想写成论文，敬请各位专家指导。

在即时战略计算机游戏中，大都采用一种实时的寻径算法为可移动物体(如：坦克、士兵等)计算一条较为"聪明"或者说根据一定算法是最优化的行走路线，如图1所示，下面我们进行详细论述。

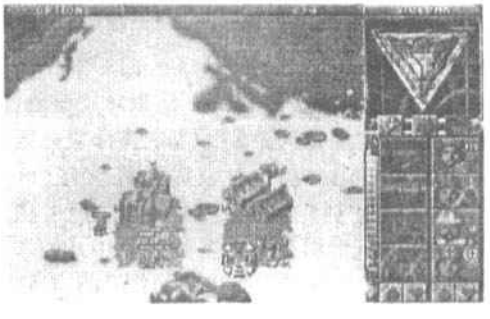

图1　红色警报游戏界面图

2　单个物体的寻径算法

寻径算法中需要解决的最基本问题是避开障碍物，这个很容易理解，您可以想象一下：当您兴致勃勃地坐在电脑前，正指挥着屏幕上的千军万马纵横驰骋时，突然发现那些坦克车一碰到障碍物便直撞上去不会拐弯或立即停止行动，您肯定会因为它们的"愚蠢行为"而大为扫兴。

2.1　A*算法

下面我们首先简单介绍一下比较先进的 A* 算法的一些基本理论知识，在微软公司的"帝国时代"游戏中就采用了此算法。在对象从起点到达目标结点前进的智能过程中，搜索是不可避免的，对象要进行种种试探，所谓"摸着石头过河"，从理论上说，只要存在能够到达目标结点的路径，广度优先搜索一定能够在有限步内找到解，而且路径最短，但是，广度优先搜索存在着结点数目随深度和复杂度的增长而指数增长并导致消耗很长时间的致命弱点。无论是广度优先搜索还是深度优先搜索，对寻找通向某一目标结点的路径来说都属于盲目搜索方法，它们只适用于不太复杂的任务，对 NP 完全类问题(即指数复杂性问题)，与传统的算法一样，都难以避免组合爆炸的厄运。对于许多任务来说，如果利用与任务有关的启发信息进行启发式搜索，往往能够大大减少搜索的代价，并找到比较满意的解。启发信息通常用在待扩展的结点上，使得搜索总是沿着那些被认为是最有希望的区段来扩展。为此，我们用估价函数 f(n)对结点 n 的"有希望"程度建立一种数值的评估。f(n)表示从初始结点 s 经结点 n 到某一目标 t 的最佳路径的代价 $f^*(n)$ 的估计，由两部分相加得到：一是从 s 到 n 的最佳代价 $g^*(n)$ 的估计 g(n)，另一是从 n 到 t 的最佳代价 $h^*(n)$ 的估计 h(n)。也就是说 $f(n)=g(n)+h(n)$ 作为 $f^*(n)=g^*(n)+h^*(n)$ 的估计。估计值越小的结点，被认为希望度越高，应该优先扩展。当然，代价均取非负值。g(n)的一个明显选择是取迄今为止已经搜索的从 s 到 n 的各条路径代价的最小值，每条路径的代价等于组成该路径的各段弧的代价之和。显然，$g(n)\geqslant g^*(n)$，h(n)涉及对未搜索路径的估计，它就是启发函数，其

精确程度主要依赖于问题领域的启发信息。启发式图搜索过程一般称为A算法，如果启发函数$h(n)\equiv 0$，即毫无启发信息的情况，此时若用$d(n)$表示结点n的深度，则如果$g(n)=d(n)$，那么A算法就退化为广度优先搜索算法。实际上，已经证明：若对所有结点n，$h(n)\leqslant h^*(n)$，则算法A一定能够找到一条到达目标结点的最佳路径。此时，算法A称为A*算法，它是一种可采纳的一定能够找到最佳求解路径的搜索算法。$h(n)$决定着算法的启发能力，一般来说，启发能力强则搜索效率高。在不牺牲算法可采纳性的条件下，为增强启发能力，通常尽量取$h(n)$为$h^*(n)$下界的最大值；有时，在求解复杂智能型问题时，我们常常只追求满意解，而不强调最佳解，因此有时选用不属$h^*(n)$下界范围的$h(n)$，也可以使启发能力大为改善。

2.2 简单单个物体寻径方法

下面我们介绍一种浅显易懂，并且在计算机上非常容易实现的单个物体的寻径方法，“红色警报”(Red Alert)游戏的人员车辆寻径算法就是以这种方法为基础开发的(参考图2)。

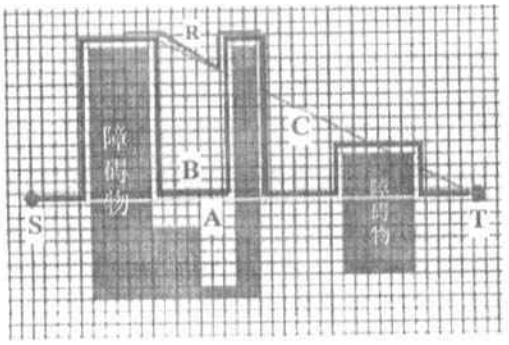

图2 物体行走路线标识图

[注：红色圆点表示起点S，蓝色圆点表示终点T；黄色直线标记为A，黑色直线标记为B，红色直线标记为C，蓝色直线为物体实际行走的路线，标记为R。]

现在假设我们的任务是使物体从起点S到达目标结点T，则“红警”的基础算法为：

(1)首先从起点S到终点T拉一条虚拟直线A。

(2)沿着直线A的方向朝终点T行进，一旦遇到障碍物便按顺时针方向绕着障碍物行走，直至碰到直线A或者沿着直线A的方向前进。

(3)重复步骤(2)，便一定能够最终达到目的地T。

该算法计算出来的路径并不是最短的，有时还会走出“愚蠢”的路线，但速度很快，能在较低档次PC机上满足游戏中实时速度的要求，玩过“红警”的人都知道，其物体寻径速度是很快的。

2.3 改进后的单物体寻径方法

我们可以对上述算法作几点改进，使可移动物体的行走路线更趋合理。

(1)从图2可以看出，显然可移动物体应按沿逆时针方向绕障碍物走，这样距离近，而不该死按顺时针方向去绕弯子。于是，我们可以设想，当遇到障碍物时，首先要判断环绕方向。如果比起顺时针方向，逆时针方向行走能以更短的路径碰到直线A，那么选取逆时针方向绕着障碍物行走，反之亦然。

(2)尽量走直线路径，减少绕行路程。如图2所示，先按上述算法计算出行走路线B，然后当可移动物体在路线B上绕着障碍物行进时，每走一步便从当前位置向终点拉一条直线C，如果沿着直线C前进能与目前还未经过的行走路线B的某段相遇(中间没有障碍物)，就终止绕行，直接按直线C行走至路线B，从而缩短了路途。

(3)当终点处于障碍物包围之中，可移动物体永远不可能到达终点，无论是按顺时针方向还是逆时针方向绕行都只能回到原地，无法行进了。对于这种情况的处理方式就是绕障碍物一周，选择离终点距离最近的地方停下来。

当然，这里讨论的寻径算法在具体实现中主要涉及到直线和环绕障碍物路线的生成技术。至于直线的光栅化生成技术，因限于篇幅，在此不再赘述，读者可参考计算机图形学方面的专著。

2.4 环绕障碍物路线生成算法

下面我们给出环绕障碍物路线的生成算法(以顺时针方向为例)。

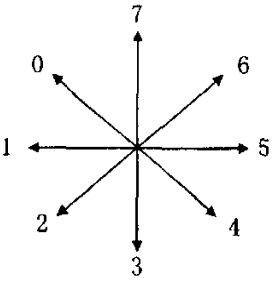

图3 方向标识图

(1)当环绕障碍物行走时，先要判断当前障碍物相对于可移动物体的方向，标记为整数i(见图3)。例如：障碍物在可移动物体的右上方，就记作方向6，左下方，就记作方向2。

(2)接着可移动物体依次查看方向$j=(i+n) \bmod 8 (n=1\cdots 7)$，直至发现方向j处无障碍物为止。若障碍物位于正上方(7)，可移动物体首先看左上方向$(0=(7+1) \bmod 8)$能否通过，如果不通就接着试左边方向$(1=(7+2) \bmod 8)$，还不行再依次按图上所示箭头试探其余几个方向，直至找到出口，并往该

方向行走一步。若障碍物位于左下方(2),则依次按下方(3)、右下方(4)、右方(5)、右上方(6)、上方(7)、左上方(0)、左方(1)等七个方向查看是否有通路。至于障碍物相对于可移动物体的其他位置的处理,照此类推。

(3)重复步骤(1)、(2),便能完成顺时针环绕障碍物行走。

该算法详细的伪语言代码描述如下,我们可通过反复调用自定义的函数 ClockwiseWalkOneStep,即可实现物体沿顺时针方向绕障碍物一周的功能。

```
boolean ClockwiseWalkOneStep(int& nCol,int& nRow,int&
        i);
{
  int n;
for ( n = 1; n <=7 ; n++)
  {
    i =(i + n) mod 8;
if ( Map[nCol,nRow]. IsPassable(i) = true )
    break;
  }
if ( n==7 && Map[nCol,nRow]. IsPassable(i) = false )
return false;
 switch ( i )
    case 1:
      {
        nCol = nCol-1;
        i = 7;
        break;
      }
    case 2:
      {
        if (Map[nCol,nRow]. IsPassable(3) == false )
          // 尽量走垂直或水平方向
        {
          nCol = nCol-1;
          nRow = nRow+1;
          i = 7;
        }
        else
        {
          nRow = nRow+1;
          i = 0;
        }
        break;
      }
    case 3:
      {
        nRow = nRow+1;
        i = 1;
        break;
      }
    case 4:
      {
        if ( Map[nCol,nRow]. IsPassable(5) == false )
        {
          nCol = nCol+1;
          nRow = nRow+1;
          i = 1;
        }
        else
        {
          nCol = nCol+1;
          i = 2;
          end
        }
        break;
      }
    case 5:
      {
        nCol = nCol+1;
        i = 3;
        break;
      }
    case 6:
      {
        if ( Map[nCol,nRow]. IsPassable(7) == false )
        {
          nCol = nCol+1;
          nRow = nRow-1;
          i = 3;
        }
        else
        {
          nRow = nRow-1;
          i = 4;
        }
        break;
      }
    case 7:
      {
        nRow = nRow-1;
        i = 5;
        break;
      }
    case 0:
      {
        if ( Map[nCol,nRow]. IsPassable(1) = false )
        {
          nCol = nCol-1;
          nRow = nRow-1;
          i = 5
        }
        else
        {
          nCol = nCol-1;
          i = 6;
        }
        break;
      }
  }
  return true;
}
```

3 动态障碍物环境中的行走路径生成技术

刚才描述的算法只适用于单个可移动物体在静止的障碍物环境中生成行走路径。然而在实际的即时战略游戏中,经常是一群坦克在一个动态的地图上纵横驰骋,不同的坦克在行走过程中相遇时互为障碍物,而且是可移动的障碍物。因此,必须对上面的算法作出修改使得物体在运动中避开可移动的障碍物。

最直观的解决办法是,一碰到可移动的障碍物,马上重新计算行走路径,该方法的缺点是重复计算路径,耗时太多。其实,当某个物体碰撞到一个处于运动状态的可移动障碍物时,可以先等待一段时间,接着若发现可移动障碍物已挪离开,就可以按原来的行走路径继续前进,从而节省了重复计算路径的时间;反之,若发现可移动障碍物仍在原地,就比较两者的优先级(预先为每一个行走物体分配一个优先级),优先级较低则继续等待,优先级高者立刻重新计算行走路径避开可移动的障碍物。

显然,运用该方法可以减少很多重复计算,加快了执行速度。该算法伪语言代码描述如下:

```
//首先要绘制背景;
for (i = 1; i <= n; i++)
      Draw(Tank[i]. CurrentMovingPos);
//然后遍历所有可移动物体
for (i = 1; i <= n; i++)
```

```
{
  Switch ( Tank[i].State ) {
    case Run:
      {
        if (Tank[i]接到指挥命令)
        {
          Tank[i].GoalPos = 目的地址;
          Tank[i].State = FindNewPath;
        }
        if (Tank[i].Pos == RouteList[Tank[i]].GetEndPos()) //到达终点
        {
          Tank[i].State = Stillness;
          break;
        }
        Tank [i].PosNext = RouteList[Tank[i]].GetNextPos(); //取出下一步
        if ( Map[Tank[i].PosNext].Passable == true)
        {
          Tank[i].State = Moving;
          Tank[i].CurrentMovingPos = Tank[i].Pos;
          Map[Tank[i].PosNext].Passable = false;
          Map[Tank[i].Pos].Passable = false;
        }
        else
        {
          if (Map[Tank[i].PosNext].State == Stillness
          .|| (Tank[i].WaitTime > 规定时间 &&
            Tank [i].PriorNum > Map [Tank [i]
            .PosNext].PriorNum))
            Tank[i].State = FindNewPath;
          else
            Tank[i].WaitTime++;
        }
        break;
      }
    case FindNewPath:
      {
        根据 Tank[i].GoalPos,计算 Tank[i]的行走路径,
并添入 RouteList[Tank[i]];
        if ( RouteList[Tank[i]] 为空 )
          Tank[i].State = FindNewPath;
        else
        {
          Tank[i].State = Run;
          Tank[i].WaitTime = 0;
        }
        break;
      }
    case Moving:
      {
          // 在 Pos 与 PosNext 之间作线性插值
if ((Tank[i].CurrentMovingPos + nStep) < Tank[i]
  .PosNext )
        {
          Tank[i].CurrentMovingPos += nStep;
          Tank[i].State = Moving;
        }
        else
        {
          Tank[i].CurrentMovingPos=Tank[i].Pos-
            Next;
          Tank[i].State = Run;
          Tank[i].WaitTime = 0;
          Map[Tank[i].PosNext].Passable = false;
          Map[Tank[i].Pos].Passable = true;
          Tank[i].Pos = Tank[i].PosNext;
        }
      }
    case Stillness:
      {
        if (Tank[i]接到指挥命令)
        {
          Tank[i].GoalPos = 目的地址;
          Tank[i].State = FindNewPath;
        }
        break;
      }
  }
}
```

在上述算法中，可移动物体具有四种状态：Stillness、Run、FindNewPath 和 Moving(见图4)。

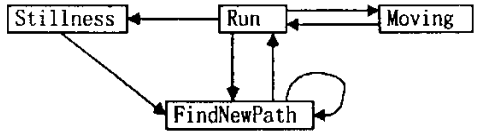

图4 可移动物体状态转换图

在初始情况下，可移动物体处于 Stillness 静止状态。当接受指挥命令后，便进入 FindNewPath 状态。一旦计算出行走路径，可移动物体就置为 Run 状态。在 Run 状态下，从行走路径列表中取出下一步的坐标，若该坐标所示位置无障碍物，状态变迁为 Moving，进行前后两步间的坐标插值，结束后又回到 Run 状态；若发现原先计算出来的行走路径已经不适用了，便从 Run 状态转为 FindNewPath 状态，等找到新路径后，再恢复至 Run 状态。最后，当到达终点时，可移动物体的状态又恢复为 Stillness 静止状态。

本算法的实际效果如图5所示[注：可参考本文所附程序 GameDemo.exe]，另外本演示游戏还有一些其他功能，如：可使用鼠标左键选取移动物体，鼠标右键确定目标；按 Ctrl＋〈数字键〉可定义战斗小组编号；按住 Insert 键，可在光标处添加障碍物等，当然游戏还很简单，很多功能还有待完善。

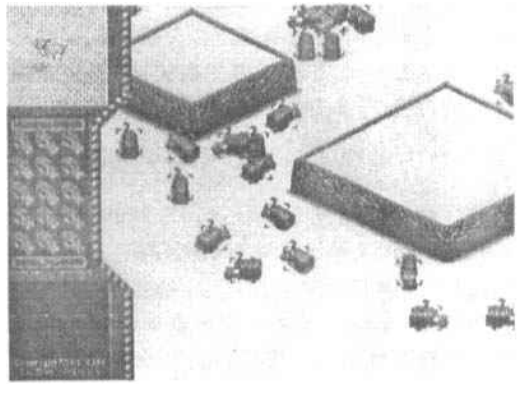

图5 游戏寻径问题演示

参 考 文 献

1 吴泉源，刘江宁. 人工智能与专家系统. 长沙：国防科技大学出版社，1999.3