初识 A*算法

写这篇文章的初衷是应一个网友的要求，当然我也发现现在有关人工智能的中文站点实在太少，我在这里抛砖引玉，希望大家都来热心的参与。

还是说正题，我先拿 A*算法开刀，是因为 A*在游戏中有它很典型的用法，是人工智能在游戏中的代表。

A*算法在人工智能中是一种典型的启发式搜索算法，为了说清楚 A*算法，我看还是先说说何谓启发式算法。

一、何谓启发式搜索算法

在说它之前先提提状态空间搜索。状态空间搜索，如果按专业点的说法就是将问题求解 过程表现为从初始状态到目标状态寻找这个路径的过程。通俗点说，就是在解一个问题时，找到一条解题的过程可以从求解的开始到问题的结果（好象并不通俗 哦）。由于求解问题的过程中分枝有很多，主要是求解过程中求解条件的不确定性，不完备性造成的，使得求解的路径很多这就构成了一个图，我们说这个图就是状 态空间。问题的求解实际上就是在这个图中找到一条路径可以从开始到结果。这个寻找的过程就是状态空间搜索。

常用的状态空间搜索有深度优先和广度优先。广度优先是从初始状态一层一层向下找，直到找到目标为止。深度优先是按照一定的顺序前查找完一个分支，再查找另一个分支，以至找到目标为止。这两种算法在数据结构书中都有描述，可以参看这些书得到更详细的解释。

前面说的广度和深度优先搜索有一个很大的缺陷就是他们都是在一个给定的状态空间中穷举。这在状态空间不大的情况下是很合适的算法，可是当状态空间十分大，且不预测的情况下就不可取了。他的效率实在太低，甚至不可完成。在这里就要用到启发式搜索了。

启发式搜索就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置， 再从这个位置进行搜索直到目标。这样可以省略大量无畏的搜索路径，提到了效率。在启发式搜索中，对位置的估价是十分重要的。采用了不同的估价可以有不同的 效果。我们先看看估价是如何表示的。

启发中的估价是用估价函数表示的，如：

f(n) = g(n) + h(n)

其中 f(n)是节点 n 的估价函数，g(n)实在状态空间中从初始节点到 n 节点的实 际代价，h(n)是从 n 到目标节点最佳路径的估计代价。在这里主要是 h(n)体现了搜索的启发信息，因为 g(n)是已知的。如果说详细点，g(n)代表了 搜索的广度的优先趋势。但是当 h(n)>>g(n)时，可以省略 g(n),而提高效率。这些就深了，

不懂也不影响啦！我们继续看看何谓 A*算 法。

二、初识 A*算法

启发式搜索其实有很多的算法，比如：局部择优搜索法、最好优先搜索法等等。当然 A*也是。这些算法都使用了启发函数，但在具体的选取最佳搜索节点时的策略不同。**象局部择优搜索法，就是在搜索的过程中选取"最佳节点"后舍弃其他的兄弟 节点，父亲节点，而一直得搜索下去。这种搜索的结果很明显，由于舍弃了其他的节点，可能也把最好的节点都舍弃了，因为求解的最佳节点只是在该阶段的最佳并 不一定是全局的最佳。最好优先就聪明多了，他在搜索时，便没有舍弃节点（除非该节点是死节点），在每一步的估价中都把当前的节点和以前的节点的估价值比较 得到一个"最佳的节点"。这样可以有效的防止"最佳节点"的丢失。那么 A*算法又是一种什么样的算法呢？其实 A*算法也是一种最好优先的算法。**只不过要加 上一些约束条件罢了。由于在一些问题求解时，我们希望能够求解出状态空间搜索的最短路径，也就是用最快的方法求解问题，A*就是干这种事情的！我们先下个 定义，如果一个估价函数可以找出最短的路径，我们称之为可采纳性。A*算法是一个可采纳的最好优先算法。A*算法的估价函数可表示为：

f'(n) = g'(n) + h'(n)

这里，f'(n)是估价函数，g'(n)是起点到终点的最短路径值，h'(n)是 n 到目标的最断路经的启发值。由于这个 f'(n)其实是无法预先知道的，所以我们用前面的估价函数 f(n)做近似。g(n)代替 g'(n)，但 g(n)>=g'(n)才可（大多数情况下都是满足的，可以不用考虑），h(n)代替 h'(n)，但 h(n)<=h'(n)才可（这一点特别 的重要）。可以证明应用这样的估价函数是可以找到最短路径的，也就是可采纳的。我们说应用这种估价函数的最好优先算法就是 A*算法。哈！你懂了吗？肯定没 懂！接着看！

举一个例子，其实广度优先算法就是 A*算法的特例。其中 g(n)是节点所在的层数，h(n)=0，这种 h(n)肯定小于 h'(n)，所以由前述可知广度优先算法是一种可采纳的。实际也是。当然它是一种最臭的 A*算法。

再说一个问题，就是有关 h(n)启发函数的信息性。h(n)的信息性通俗点说其实 就是在估计一个节点的值时的约束条件，如果信息越多或约束条件越多则排除的节点就越多，估价函数越好或说这个算法越好。这就是为什么广度优先算法的那么臭 的原因了，谁叫它的 h(n)=0，一点启发信息都没有。但在游戏开发中由于实时性的问题，h(n)的信息越多，它的计算量就越大，耗费的时间就越多。就应 该适当的减小 h(n)的信息，即减小约束条件。但算法的准确性就差了，这里就有一个平衡的问题。可难了，这就看你的了！

好了我的话也说得差不多了，我想你肯定是一头的雾水了，其实这是写给懂 A*算法的同志看的。哈哈！你还是找一本人工智能的书仔细看看吧！我这几百字是不足以将 A*算法讲清楚的。只是起到抛砖引玉的作用，希望大家热情参与嘛！

预知 A*算法的应用，请看姊妹篇《深入 A*算法》。

深入 A*算法

一、前言

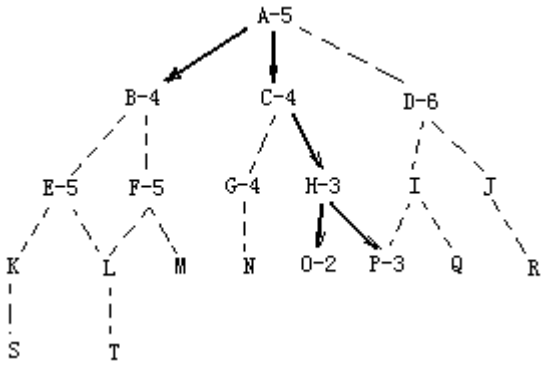
在这里我将对 A*算法的实际应用进行一定的探讨，并且举一个有关 A*算法在最短路径搜索的例子。值得注意的是这里并不对 A*的基本的概念作介绍，如果你还对 A*算法不清楚的话，请看姊妹篇《初识 A*算法》。

这里所举的例子是参考 AMIT 主页中的一个源程序，使用这个源程序时，应该遵守一定的公约。

二、A*算法的程序编写原理

我在《初识 A*算法》中说过，A*算法是最好优先算法的一种。只是有一些约束条件而已。我们先来看看最好优先算法是如何编写的吧。

如图有如下的状态空间：（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）



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；

看了具体的过程，再看看伪程序吧。算法的伪程序如下：

```
Best_First_Search()
{
  Open = [起始节点];
  Closed = [];
  while (Open 表非空)
  {
    从 Open 中取得一个节点 X，并从 OPEN 表中删除。
    if (X 是目标节点)
    {
      求得路径 PATH;
      返回路径 PATH;
    }
    for (每一个 X 的子节点 Y)
    {
      if (Y 不在 OPEN 表和 CLOSE 表中)
      {
        求 Y 的估价值;
        并将 Y 插入 OPEN 表中;
      }
      //还没有排序
      else if (Y 在 OPEN 表中)
      {
        if (Y 的估价值小于 OPEN 表的估价值)
          更新 OPEN 表中的估价值;
      }
      else //Y 在 CLOSE 表中
      {
        if (Y 的估价值小于 CLOSE 表的估价值)
        {
          更新 CLOSE 表中的估价值;
          从 CLOSE 表中移出节点，并放入 OPEN 表中;
        }
      }
      将 X 节点插入 CLOSE 表中;
      按照估价值将 OPEN 表中的节点排序;
    }//end for
  }//end while
}//end func
```

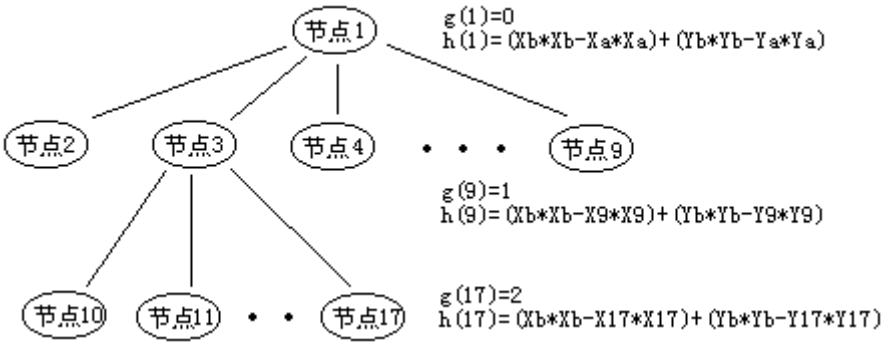
啊！伪程序出来了，写一个源程序应该不是问题了，依葫芦画瓢就可以。A*算法的程 序与此是一样的，只要注意估价函数中的 g(n)的 h(n)约束条件就可以了。不清楚的可以看看《初识 A*算法》。好了，我们可以进入另一个重要的话题，用 A*算法实现最短路径的搜索。在此之前你最好认真的理解前面的算法。不清楚可以找我。我的 Email 在文章尾。

三、用 A*算法实现最短路径的搜索

在游戏设计中，经常要涉及到最短路径的搜索，现在一个比较好的方法就是用 A*算法进行设计。他的好处我们就不用管了，反正就是好！^_*

注意下面所说的都是以 ClassAstar 这个程序为蓝本，你可以在这里下载这个程序。这个程序是一个完整的工程。里面带了一个 EXE 文件。可以先看看。

先复习一下，A*算法的核心是估价函数 f(n)，它包括 g(n)和 h(n)两部 分。g(n)是已经走过的代价，h(n)是 n 到目标的估计代价。在这个例子中 g(n)表示在状态空间从起始节点到 n 节点的深度即**所需步数**，h(n)表示 n 节点所在地 图的位置到目标位置的直线距离。啊！一个是状态空间，一个是实际的地图，不要搞错了。再详细点说，**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：**注意估价函数的计算**



仔细看看节点 1、9、17 的 g(n)和 h(n)是怎么计算的。现在应该知道了下面 程序中的 f(n)是如何计算的吧。开始讲解源程序了。其实这个程序是一个很典型的教科书似的程序，也就是说只要你看懂了上面的伪程序，这个程序是十分容易 理解的。不过他和上面的伪程序有一些的不同，我在后面会提出来。

先看搜索主函数：

```
void AstarPathfinder::FindPath(int sx, int sy, int dx, int dy)
{
    NODE *Node, *BestNode;
    int TileNumDest;
    //得到目标位置，作判断用
    TileNumDest = TileNum(sx, sy);
```

```
    //生成 Open 和 Closed 表
    OPEN = ( NODE* )calloc(1,sizeof( NODE ));
    CLOSED=( NODE* )calloc(1,sizeof( NODE ));
    //生成起始节点，并放入 Open 表中
    Node=( NODE* )calloc(1,sizeof( NODE ));
    Node->g = 0;
    //这是计算 h 值
    // should really use sqrt().
    Node->h = (dx-sx)*(dx-sx) + (dy-sy)*(dy-sy);
    //这是计算 f 值，即估价值
    Node->f = Node->g+Node->h;
    Node->NodeNum = TileNum(dx, dy);
    Node->x = dx; Node->y = dy;
    // make Open List point to first node
    OPEN->NextNode=Node;
    for (;;)
    {
        //从 Open 表中取得一个估价值最好的节点
        BestNode=ReturnBestNode();
        //如果该节点是目标节点就退出
        // if we've found the end, break and finish break;
        if (BestNode->NodeNum == TileNumDest)
        //否则生成子节点
        GenerateSuccessors(BestNode,sx,sy);
    }
    PATH = BestNode;
}
```

再看看生成子节点函数：

```
void AstarPathfinder::GenerateSuccessors(NODE *BestNode, int dx, int  dy)
{
    int x, y;
    //哦！依次生成八个方向的子节点，简单！
    // Upper-Left
    if ( FreeTile(x=BestNode->x-TILESIZE, y=BestNode->y-TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Upper
    if ( FreeTile(x=BestNode->x, y=BestNode->y-TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Upper-Right
    if ( FreeTile(x=BestNode->x+TILESIZE, y=BestNode->y-TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Right
```

```
    if ( FreeTile(x=BestNode->x+TILESIZE, y=BestNode->y) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Lower-Right
    if ( FreeTile(x=BestNode->x+TILESIZE, y=BestNode->y+TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Lower
    if ( FreeTile(x=BestNode->x, y=BestNode->y+TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Lower-Left
    if ( FreeTile(x=BestNode->x-TILESIZE, y=BestNode->y+TILESIZE) )
        GenerateSucc(BestNode,x,y,dx,dy);
    // Left
    if ( FreeTile(x=BestNode->x-TILESIZE, y=BestNode->y) )
        GenerateSucc(BestNode,x,y,dx,dy);
}
```

看看最重要的函数：

```
void AstarPathfinder::GenerateSucc(NODE *BestNode,int x, int y, int dx, int
dy)
{
    int g, TileNumS, c = 0;
    NODE *Old, *Successor;
    //计算子节点的 g 值
    // g(Successor)=g(BestNode)+cost of getting from BestNode to Successor
    g = BestNode->g+1;
    // identification purposes
    TileNumS = TileNum(x,y);
    //子节点再 Open 表中吗?
    // if equal to NULL then not in OPEN list, else it returns the Node in
Old
    if ( (Old=CheckOPEN(TileNumS)) != NULL )
    {
        //若在
        for( c = 0; c < 8; c++)
        // Add Old to the list of BestNode's Children (or Successors).
            if( BestNode->Child[c] == NULL )
                break;
        BestNode->Child[c] = Old;
        //比较 Open 表中的估价值和当前的估价值（只要比较 g 值就可以了）
        // if our new g value is < Old's then reset Old's parent to point
to BestNode
        if ( g < Old->g )
        {
```

```
            //当前的估价值小就更新Open表中的估价值
            Old->Parent = BestNode;
            Old->g = g;
            Old->f = g + Old->h;
        }
    }
    else
    //在Closed表中吗?
    // if equal to NULL then not in OPEN list, else it returns the Node in
Old
    if ( (Old=CheckCLOSED(TileNumS)) != NULL )
    {
        //若在
        for( c = 0; c< 8; c++)
        // Add Old to the list of BestNode's Children (or Successors).
            if ( BestNode->Child[c] == NULL )
                break;
        BestNode->Child[c] = Old;
        //比较Closed表中的估价值和当前的估价值（只要比较g值就可以了）
        // if our new g value is < Old's then reset Old's parent to point
to BestNode
        if ( g < Old->g )
        {
            //当前的估价值小就更新Closed表中的估价值
            Old->Parent = BestNode;
            Old->g = g;
            Old->f = g + Old->h;
            //再依次更新Old的所有子节点的估价值
            // Since we changed the g value of Old, we need
            // to propagate this new value downwards, i.e.
            // do a Depth-First traversal of the tree!
            PropagateDown(Old);
        }
    }
    //不在Open表中也不在Close表中
    else
    {
        //生成新的节点
        Successor = ( NODE* )calloc(1,sizeof( NODE ));
        Successor->Parent = BestNode;
        Successor->g = g;
        // should do sqrt(), but since we don't really
        Successor->h = (x-dx)*(x-dx) + (y-dy)*(y-dy);
        // care about the distance but just which branch looks
```

```
            Successor->f = g+Successor->h;
            // better this should suffice. Anyayz it's faster.
            Successor->x = x;
            Successor->y = y;
            Successor->NodeNum = TileNumS;
            //再插入 Open 表中，同时排序。
            // Insert Successor on OPEN list wrt f
            Insert(Successor);
            for( c =0; c < 8; c++)
            // Add Old to the list of BestNode's Children (or Successors).
            if ( BestNode->Child[c] == NULL )
                break;
            BestNode->Child[c] = Successor;
        }
    }
```

哈哈！A*算法我懂了！当然，我希望你有这样的感觉！不过我还要再说几句。仔细看 看这个程序，你会发现，这个程序和我前面说的伪程序有一些不同，在 GenerateSucc 函数中，当子节点在 Closed 表中时，没有将子节点从 Closed 表中删除并放入 Open 表中。而是直接的重新的计算该节点的所有子节点的估价值（用 PropagateDown 函数）。这样可以快一些！另当 子节点在 Open 表和 Closed 表中时，重新的计算估价值后，没有重新的对 Open 表中的节点排序，我有些想不通，为什么不排呢？:-(，会不会是一个 小小的 BUG。你知道告诉我好吗？

好了！主要的内容都讲完了，还是完整仔细的看看源程序吧！希望我所的对你有一点帮助，一点点也可以。如果你对文章中的观点有异议或有更好的解释都告诉我。我的 email 在文章最后！

A*高效搜索算法 2006/09/11 rickone

了解了基本搜索算法，下面就来看 A*，神奇的 A*。（--->搜索方法小结）

A*是一种启发式搜索，一种有序搜索，它之所以特殊完全是在它的估价函数上，如果我要求的是从初始结点到目的结点的一个最短路径（或加权代价）的可 行解，那对于一个还不是目标结点的结点，我对它的评价就要从两个方面评价：第一，离目标结点有多近，越近越好；第二，离起始结点有多远，越近越好。记号[a,b]是表示结点 a 到结点 b 的实际最短路径代价。设起始结点为 S，当前结点为 n，目标结点为 G，于是 n 的实际代价应该是 f*(n)=g*(n)+h* (n)，其中 g*(n)=[S,n],h*(n)=[n,G]，对于是 g*(n)是比较容易得到的，在搜索的过程中我们可以按搜索的顺序对它进行累积计 算，当然按 BFS 和 DFS 的不同，我们对它的估价 g(n)可以满足 g(n)>=g*(n)，大多可以是相等的。但是对于 h*(n)我们却了解得非常 少，目标结点正是要搜索的目的，我们是不知道在哪，就更不知

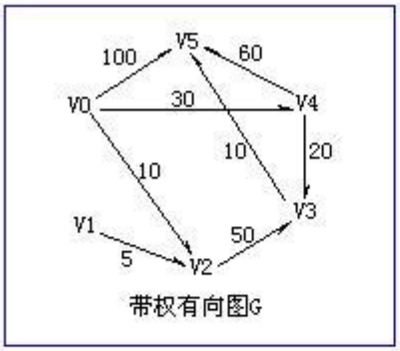
道从 n 到目标结点的路径代价，但是或多或少我们还是可以估计的，记估价函数 f(n)=g(n)+h(n)。

我们说如果在一般的图搜索算法中应用了上面的估价函数对 OPEN 表进行排序的，就称 A 算法。在 A 算法之上，如果加上一个条件，对于所有的结点 x，都有 h(x)<=h*(x)，那就称为 A*算法。如果取 h(n)=0 同样是 A*算法，这样它就退化成了有序算法。

A*算法是否成功，也就是说是否在效率上胜过蛮力搜索算法，就在于 h(n)的选取，它不能大于实际的 h*(n)，要保守一点，但越接近 h*(n)给我们的启发性就越大，是一个难把握的东西。

A*算法流程：
首先将起始结点 S 放入 OPEN 表，CLOSE 表置空，算法开始时：
1、如果 OPEN 表不为空，从表头取一个结点 n，如果为空算法失败
2、n 是目标解吗？是，找到一个解（继续寻找，或终止算法）；不是到 3
3、将 n 的所有后继结点展开，就是从 n 可以直接关联的结点（子结点），如果不在 CLOSE 表中，就将它们放入 OPEN 表，并把 S 放入 CLOSE 表，同时计算每一个后继结点的估价值 f(n)，将 OPEN 表按 f(x)排序，最小的放在表头，重复算法到 1

最短路径问题，Dijkstra 算法与 A*
A*是求这样一个和最短路径有关的问题，那单纯的最短路径问题当然可以用 A*来算，对于 g(n) 就是[S,n]，在搜索过程中计算，而 h(n)我想不出很好的办法，对于一个抽象的图搜索，很难找到很好的 h(n)，因为 h(n)和具体的问题有关。只好 是 h(n)=0，退为有序搜索，举一个小小的例子：



带权有向图G

A*求最短路径过程，起始结点V0

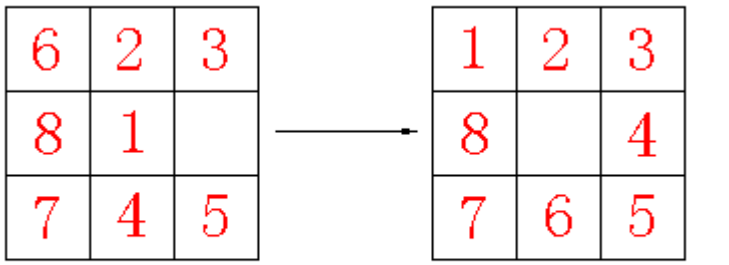
OPEN	(V0, 0)
CLOSE	空
OPEN	(V2, 10) (V4, 30) (V5, 100)
CLOSE	(V0, 0)
OPEN	(V4, 30) (V3, 60) (V5, 100)
CLOSE	(V0, 0) (V2, 10)
OPEN	(V3, 50) (V3, 60) (V5, 90) (V5, 100)
CLOSE	(V0, 0) (V2, 10) (V4, 30)
OPEN	(V5, 60) (V5, 90) (V5, 100)
CLOSE	(V0, 0) (V2, 10) (V4, 30) (V3, 50)
OPEN	空
CLOSE	(V0, 0) (V2, 10) (V4, 30) (V3, 50) (V5, 60)

rickone 2006/09/11

与结点写在一起的数值表示那个结点的价值 f(n)，当 OPEN 表为空时 CLOSE 表中将求得从 V0 到其它所有结点的最短路径。考虑到算法性能，外循 环中每次从 OPEN 表取一个元素，共取了 n 次（共 n 个结点），每次展开一个结点的后续结点时，需 O(n)次，同时再对 OPEN 表做一次排序，OPEN 表大 小是 O(n)量级的，若用快排就是 O(nlogn)，乘以外循环总的复杂度是 O(n^2logn)，如果每次不是对 OPEN 表进行排序，因为总是不断地有 新的结点添加进来，所以不用进行排序，

而是每次从 OPEN 表中求一个最小的，那只需要 O(n)的复杂度，所以总的复杂度为 O(n*n)，这相当于 Dijkstra 算法。在这个算法基础之上稍加改进就是 Dijkstra 算法。OPEN 表中常出现这样的表项：(Vk,fk1)(Vk,fk2) (Vk,fk3)，而从算法上看，只有 fk 最小的一个才有用，于是可以将它们合并，整个 OPEN 表表示当前的从 V0 到其它各点的最短路径，定长为 n，且初 始时为 V0 可直接到达的权值（不能到达为 INFINITY），于是就成了 Dijkstra 算法。

另外一个问题就是八数码难题，一个 A*的好例子。
问题描述为有这样一个 3*3 方阵格子：



rickone 2006/09/11

格子上有 1-8 八个数字外加一个空格，每次只能把与空格相临的一个数字移到空格内，移动一次算作一步，给出初始状态和目标状态，求如何以最少的步数完成移动？

设计 A*算法时，g(n)就取当前已移动的步数，h(n)取各个数字到目标状态中对应数字的位置的最短距离之和，这样选取的原因是，对于每一次移动，只能使一个数字改变一个相临位置，所以 h(n)步是至少需要的，所以满足 h(n)<=h*(n)。

A*的成功之处就是在选择好的 h(n)，如果实在没办法令它为 0 也是可以求得问题的解的。

A*（A Star）算法：启发式（heuristic）算法

A*（A-Star)算法是一种静态路网中求解最短路最有效的方法。

公式表示为： f(n)=g(n)+h(n)，

其中 f(n) 是节点 n 从初始点到目标点的估价函数，

g(n) 是在状态空间中从初始节点到 n 节点的实际代价，
h(n)是从 n 到目标节点最佳路径的估计代价。

保证找到最短路径（最优解的）条件，关键在于估价函数 h(n)的选取：
估价值 h(n)<= n 到目标节点的距离实际值，这种情况下，搜索的点数多，搜索范围大，效率低。但能得到最优解。
如果 估价值>实际值，搜索的点数少，搜索范围小，效率高，但不能保证得到最优解。
估价值与实际值越接近，估价函数取得就越好。
例如对于几何路网来说，可以取两节点间欧几理德距离（直线距离）做为估价值，即 f=g(n)+sqrt((dx-nx)*(dx-nx)+(dy- ny)*(dy-ny))；这样估价函数 f 在 g 值一定的情况下，会或多或少的受估价值 h 的制约，节点距目标点近，h 值小，f 值相对就小，能保证最短路的搜 索向终点的方向进行。明显优于 Dijstra 算法的毫无无方向的向四周搜索。

conditions of heuristic
Optimistic (must be less than or equal to the real cost)
As close to the real cost as possible

主要搜索过程：
创建两个表，OPEN 表保存所有已生成而未考察的节点，CLOSED 表中记录已访问过的节点。
遍历当前节点的各个节点，将 n 节点放入 CLOSE 中，取 n 节点的子节点 X,->算 X 的估价值->
While(OPEN!=NULL)
{
从 OPEN 表中取估价值 f 最小的节点 n;
if(n 节点==目标节点) break;
else
{

if(X in OPEN) 比较两个 X 的估价值 f //注意是同一个节点的两个不同路径的估价值

if(X 的估价值小于 OPEN 表的估价值)

    更新 OPEN 表中的估价值; //取最小路径的估价值

if(X in CLOSE) 比较两个 X 的估价值 //注意是同一个节点的两个不同路径的估价值

if(X 的估价值小于 CLOSE 表的估价值)

    更新 CLOSE 表中的估价值; 把 X 节点放入 OPEN //取最小路径的估价值

if(X not in both)

求 X 的估价值;

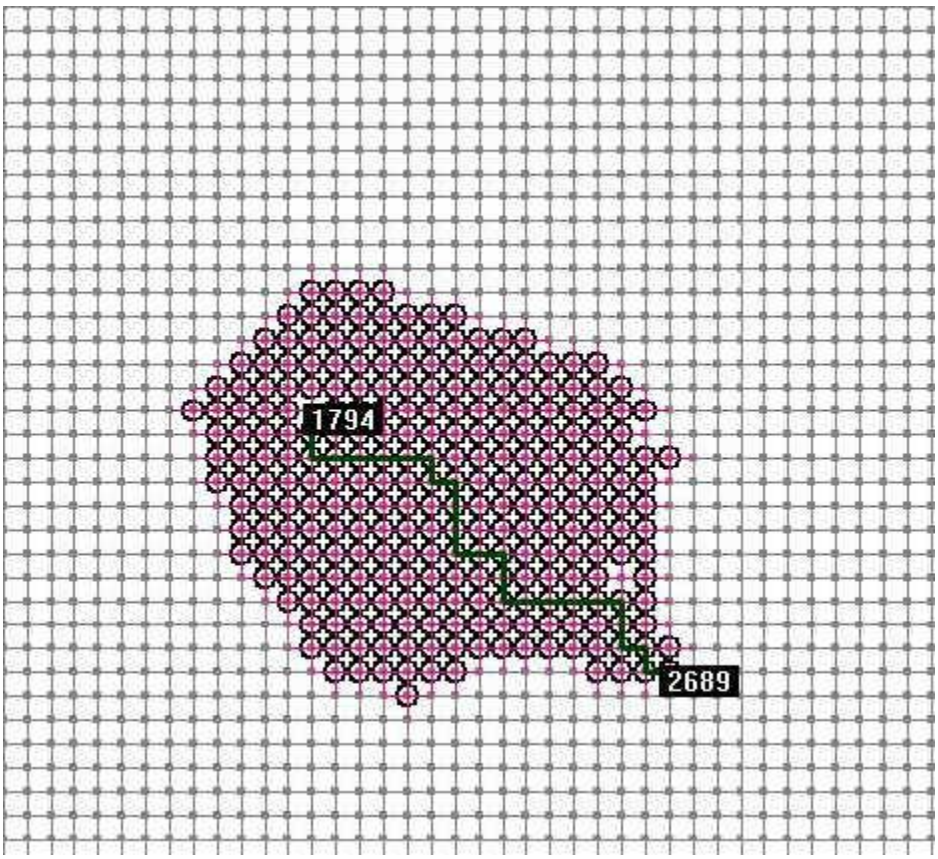
    并将 X 插入 OPEN 表中; //还没有排序

}

将 n 节点插入 CLOSE 表中;

按照估价值将 OPEN 表中的节点排序; //实际上是比较 OPEN 表内节点 f 的大小，从最小路径的节点向下进行。

}



上图是和上面 Dijkstra 算法使用同一个路网，相同的起点终点，用 A*算法的情况，计算的点数从起始点逐渐向目标点方向扩展，计算的节点数量明显比 Dijkstra 少得多，效率很高，且能得到最优解。

A*算法和 Dijistra 算法的区别在于有无估价值，Dijistra 算法相当于 A*算法中估价值为 0 的情况。

推荐文章链接：

Amit 斯坦福大学一个博士的游戏网站，上面有关于 A*算法介绍和不少有价值的链接 http://theory.stanford.edu/~amitp/GameProgramming/

Sunway 写的两篇很好的介绍启发式和 A*算法的中文文章并有 A*源码下载：

初识 A*算法 http://creativesoft.home.shangdu.net/AStart1.htm

深入 A*算法 http://creativesoft.home.shangdu.net/AStart2.htm

需要注意的是 Sunway 上面文章“深入 A*算法”中引用了一个 A*的游戏程序进行讲解，并有这个源码的下载，不过它有一个不小的 Bug，就是新的子节点放入 OPEN 表中进行了排序，而当子节点在 Open 表和 Closed 表中时，重新计算估价值后，没有重新的对 Open 表中的节点排序，这个问题会导致计算有时得不到最优解，另外在路网权重悬殊很大时，搜索范围不但超过 Dijkstra，甚至搜索全部路网，使效率大大降低。

Drew 对这个问题进行了如下修正，当子节点在 Open 表和 Closed 表中时，重新计算估价值后，删除 OPEN 表中的老的节点，将有新估价值的节点插入 OPEN 表中，重新排序，经测试效果良好，修改的代码如下，红色部分为 Drew 添加的代码. 添加进程序的相应部分即可。

在函数 GenerateSucc（）中

```
.....................................
g=BestNode->g+1;
TileNumS=TileNum((int)x,(int)y);
if ((Old=CheckOPEN(TileNumS)) != NULL)
{
for(c=0;c<8;c++)
if(BestNode->Child[c] == NULL)
break;
BestNode->Child[c]=Old;

if (g < Old->g)
{
Old->Parent=BestNode;
Old->g=g;
Old->f=g+Old->h;
```

```
//Drew 在该处添加如下红色代码
//Implement by Drew
NODE *q,*p=OPEN->NextNode, *temp=OPEN->NextNode;
while(p!=NULL && p->NodeNum != Old->NodeNum)
{
    q=p;
    p=p->NextNode;
}
if(p->NodeNum == Old->NodeNum)
{
   if(p==OPEN->NextNode)
  {
     temp = temp->NextNode;
     OPEN ->NextNode = temp;
  }
  else
  q->NextNode = p->NextNode;
 }
Insert(Old); // Insert Successor on OPEN list wrt f
}
.....................................................
```

另一种 A*（A Star）算法：

这种算法可以不直接用估价值，直接用 Dijkstra 算法程序实现 A*算法，Drew 对它进行了测试，达到和 A*完全一样的计算效果，且非常简单。

以邻接矩阵为例，更改原来邻接矩阵 i 行 j 列元素 Dij 为 Dij+Djq-Diq；起始点到目标点的方向 i->j，终点 q．Dij 为（i 到 j 路段的权重或距离）

其中：Djq,Diq 的作用相当于估价值 Djq=（j 到 q 的直线距离）；Diq=（i 到 q 的直线距离）

原理：i 到 q 方向符合 Dij+Djq > Diq ，取 Dij+Djq-Diq 小，如果是相反方向 Dij+Djq-Diq 会很大。因此达到向目标方向寻路的作用。

动态路网，最短路径算法 D*

A* 在静态路网中非常有效（very efficient for static worlds），但不适于在动态路网，环境如权重等不断变化的动态环境下。

D*是动态 A*（D-Star,Dynamic A Star）卡内及梅隆机器人中心的 Stentz 在 1994 和 1995 年两篇文章提出，主要用于机器人探路。是火星探测器采用的寻路算法。

Optimal and Efficient Path Planning for Partially-Known Environments

The Focussed D* Algorithm for Real-Time Replanning

主要方法（这些完全是 Drew 在读了上述资料和编制程序中的个人理解，不能保证完全正确，仅供参考）：

1.先用 Dijstra 算法从目标节点 G 向起始节点搜索。储存路网中目标点到各个节点的最短路和该位置到目标点的实际值 h,k（k 为所有变化 h 之中最小的 值,当前为 k=h。每个节点包含上一节点到目标点的最短路信息 1(2),2(5),5(4),4(7)。

则 1 到 4 的最短路为 1-2-5-4。

原 OPEN 和 CLOSE 中节点信息保存。

2.机器人沿最短路开始移动，在移动的下一节点没有变化时，无需计算，利用上一步 Dijstra 计算出的最短路信息从出发点向后追述即可，当在 Y 点探测到 下一节点 X 状态发生改变，如堵塞。机器人首先调整自己在当前位置 Y 到目标点 G 的实际值 h(Y)，h(Y)=X 到 Y 的新权值 c(X,Y)+X 的原实际值 h(X).X 为下一节点(到目标点方向 Y->X->G），Y 是当前点。k 值取 h 值变化前后的最小。

3.用 A*或其它算法计算，这里假设用 A*算法,遍历 Y 的子节点，点放入 CLOSE,调整 Y 的子节点 a 的 h 值，h(a)=h(Y)+Y 到子节点 a 的权重 C(Y,a),比较 a 点是否存在于 OPEN 和 CLOSE 中，方法如下：

```
while()
{
从 OPEN 表中取 k 值最小的节点 Y;
遍历 Y 的子节点 a,计算 a 的 h 值 h(a)=h(Y)+Y 到子节点 a 的权重 C(Y,a)
{
    if(a in OPEN)       比较两个 a 的 h 值
  if( a 的 h 值小于 OPEN 表 a 的 h 值 )
    {
          更新 OPEN 表中 a 的 h 值;k 值取最小的 h 值
     有未受影响的最短路经存在
     break;
    }
    if(a in CLOSE) 比较两个 a 的 h 值 //注意是同一个节点的两个不同路径的估价值
  if( a 的 h 值小于 CLOSE 表的 h 值 )
    {
```

更新 CLOSE 表中 a 的 h 值; k 值取最小的 h 值;将 a 节点放入 OPEN 表

有未受影响的最短路经存在

break;

}

if(a not in both)

将 a 插入 OPEN 表中; //还没有排序

}

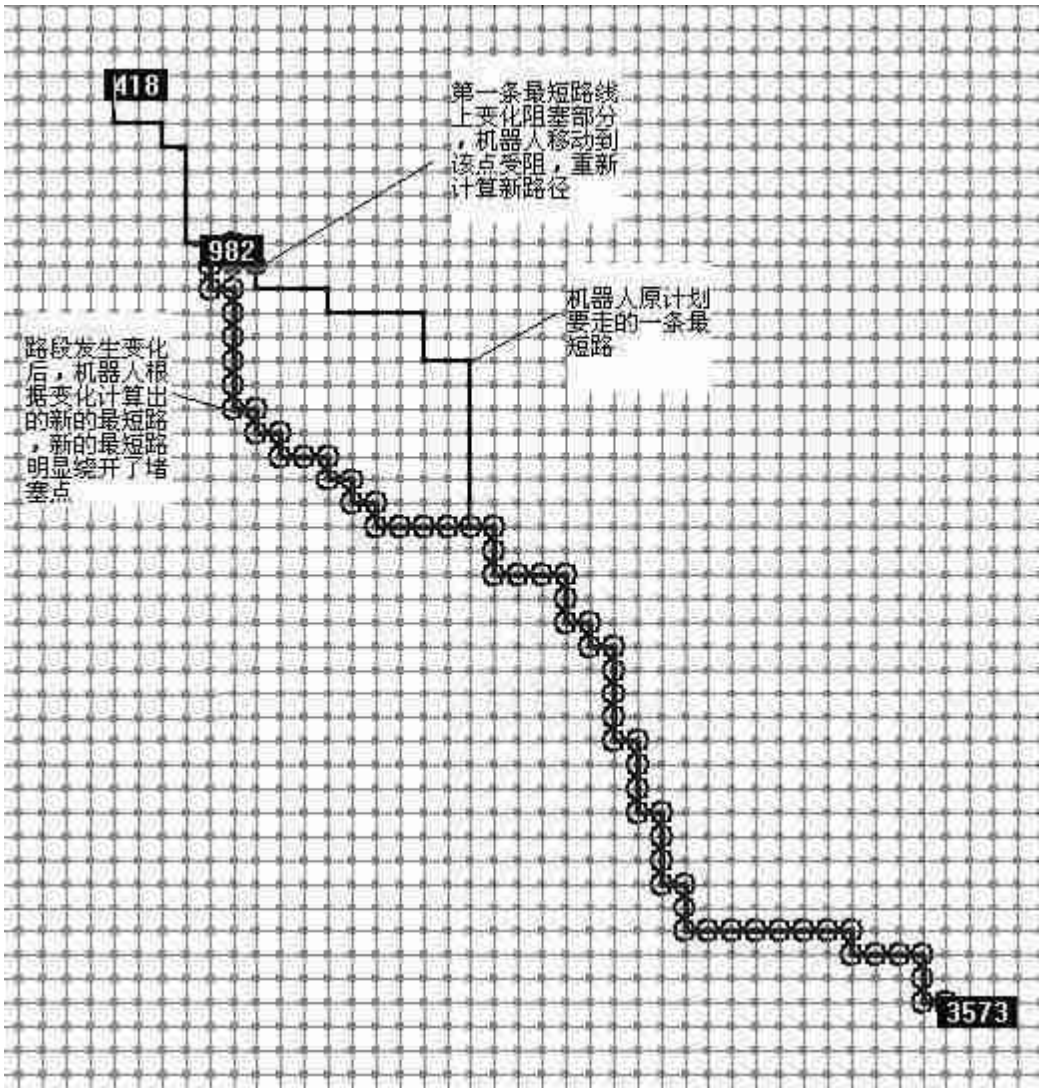
放 Y 到 CLOSE 表;

OPEN 表比较 k 值大小进行排序;

}

机器人利用第一步 Dijstra 计算出的最短路信息从 a 点到目标点的最短路经进行。

D*算法在动态环境中寻路非常有效，向目标点移动中，只检查最短路径上下一节点或临近节点的变化情况，如机器人寻路等情况。对于距离远的最短路径上发生的变化，则感觉不太适用。



上图是 Drew 在 4000 个节点的随机路网上做的分析演示，细黑线为第一次计算出的最短路，红点部分为路径上发生变化的堵塞点，当机器人位于 982 点时， 检测到前面发生路段堵塞，在该点重新根据新的信息计算路径，可以看到圆圈点为重新计算遍历过的点，仅仅计算了很少得点就找到了最短路，说明计算非常有效，迅速。绿线为计算出的绕开堵塞部分的新的最短路径。