虽然A*（读作A星）算法对初学者来说是比较深奥难懂，但是一旦你找到门路了，它又会变得非常简单。网上有很多解释A*算法的文章，但是大多数是写给那些有一定基础的人看的，而您看到的这一篇呢，是真正写给菜鸟的。

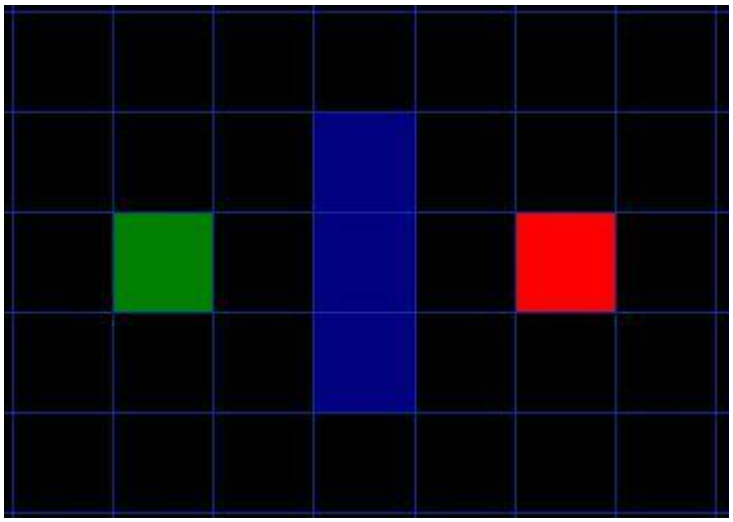
本篇文章并不想给这个算法题目作一些权威性论断，而是阐述它的基本原理，并为你理解更多相关资料与讨论打下基础。文章末尾给出了一些比较好的链接，放在“进阶阅读”一节之后。

最后，本文不是编程规范，你将可能使这里讲述的东西编写成任何计算机语言。在本文的末尾我还给出了一个例子程序包的下载链接，也许正合你意。在这个包中有C++和Blitz Basic两个版本的程序代码，如果你只是想看看A*算法是如何运作的，该包中也有可直接执行的文件供你研究。

我们还是要超越自己的（把算法弄懂），所以，让我们从头开始吧！

初步：搜索区域

我们假设某个人要从A点到达B点，而一堵墙把这两个点隔开了，如下图所示，绿色部分代表起点A，红色部分代表终点B，蓝色方块部分代表之间的墙。

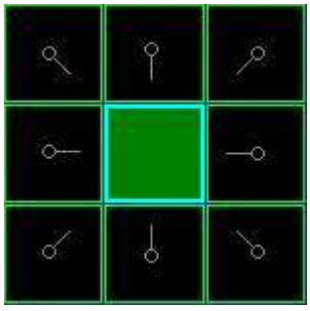


[图一]
你首先会注意到我们把这一块搜索区域分成了一个一个的方格，如此这般，使搜索区域简单化，正是寻找路径的第一步。这种方法将我们的搜索区域简化成了一个普通的二维数组。数组中的每一个元素表示对应的一个方格，该方格的状态被标记为可通过的和不可通过的。通过找出从A点到B点所经过的方格，就能得到AB之间的路径。当路径找出来以后，这个人就可以从一个格子中央移动到另一个格子中央，直到抵达目的地。
这些格子的中点叫做节点。当你在其他地方看到有关寻找路径的东西时，你会经常

发现人们在讨论节点。为什么不直接把它们称作方格呢？因为你不一定要把你的搜索区域分隔成方块，矩形、六边形或者其他任何形状都可以。况且节点还有可能位于这些形状内的任何一处呢？在中间、靠着边，或者什么的。我们就用这种设定，因为毕竟这是最简单的情况。

开始搜索

当我们把搜索区域简化成一些很容易操作的节点后，下一步就要构造一个搜索来寻找最短路径。在A*算法中，我们从A点开始，依次检查它的相邻节点，然后照此继续并向外扩展直到找到目的地。
我们通过以下方法来开始搜索：
1.
从A点开始，将A点加入一个专门存放待检验的方格的“开放列表”中。这个开放列表有点像一张购物清单。当前这个列表中只有一个元素，但一会儿将会有更多。列表中包含的方格可能会是你要途经的方格，也可能不是。总之，这是一个包含待检验方格的列表。
2.
检查起点A相邻的所有可达的或者可通过的方格，不用管墙啊，水啊，或者其他什么无效地形，把它们也都加到开放列表中。对于每一个相邻方格，将点A保存为它们的“父方格”。当我们要回溯路径的时候，父方格是一个很重要的元素。稍后我们将详细解释它。
3.
从开放列表中去掉方格A，并把A加入到一个“封闭列表”中。封闭列表存放的是你现在不用再去考虑的方格。
此时你将得到如下图所示的样子。在这张图中，中间深绿色的方格是你的起始方格，所有相邻方格目前都在开放列表中，并且以亮绿色描边。每个相邻方格有一个灰色的指针指向它们的父方格，即起始方格。



[图二]

接下来，我们在开放列表中选一个相邻方格并再重复几次如前所述的过程。但是我们该选哪一个方格呢？具有最小F值的那个。

路径排序

决定哪些方格会形成路径的关键是下面这个等式：

F = G + H
这里

G＝从起点A沿着已生成的路径到一个给定方格的移动开销。

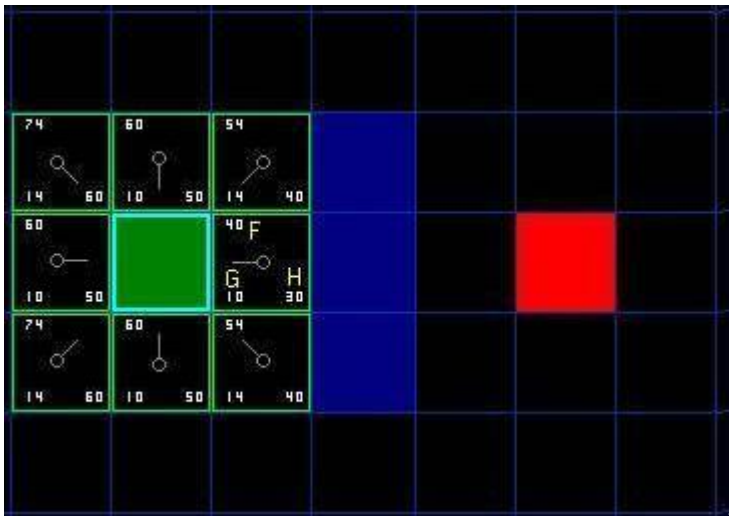
H＝从给定方格到目的方格的估计移动开销。这种方式常叫做试探，有点困惑人吧。其实之所以叫做试探法是因为这只是一个猜测。在找到路径之前我们实际上并不知道实际的距离，因为任何东西都有可能出现在半路上（墙啊，水啊什么的）。本文中给出了一种计算H值的方法，网上还有很多其他文章介绍的不同方法。
我们要的路径是通过反复遍历开放列表并选择具有最小F值的方格来生成的。本文稍后将详细讨论这个过程。我们先进一步看看如何计算那个等式。

如前所述，G是从起点A沿着已生成的路径到一个给定方格的移动开销，在本例中，我们指定每一个水平或者垂直移动的开销为 10，对角线移动的开销为 14。因为对角线的实际距离是 2 的平方根（别吓到啦），或者说水平及垂直移动开销的 1.414 倍。为了简单起见我们用了 10 和 14 这两个值。比例大概对就好，我们还因此避免了平方根和小数的计算。这倒不是因为我们笨或者说不喜欢数学，而是因为对电脑来说，计算这样的数字也要快很多。不然的话你会发现寻找路径会非常慢。

我们要沿特定路径计算给定方格的G值，办法就是找出该方格的父方格的G值，并根据与父方格的相对位置（斜角或非斜角方向）来给这个G值加上 14 或者 10。在本例中这个方法将随着离起点方格越来越远计算的方格越来越多而用得越来越多。

有很多方法可以用来估计H值。我们用的这个叫做曼哈顿（Manhattan）方法，即计算通过水平和垂直方向的平移到达目的地所经过的方格数乘以 10 来得到H值。之所以叫Manhattan方法是因为这就像计算从一个地方移动到另一个地方所经过的城市街区数一样，而通常你是不能斜着穿过街区的。重要的是，在计算H值时并不考虑任何障碍物。因为这是对剩余距离的估计值而不是实际值（通常是要保证估计值不大于实际值——译者注）。这就是为什么这个方式被叫做试探法的原因了。想要了解更多些吗？链接标记这里还有更多式子和关于试探法的额外说明。

G和H相加就得到了F。第一步搜索所得到的结果如下图所示。每个方格里都标出了F、G和H值。如起点方格右侧的方格标出的，左上角显示的是F值，左下角是G值，右下角是H值。



[图三]

我们来看看这些方格吧。在有字母的方格中，G＝10，这是因为它在水平方向上离起点只有一个方格远。起点紧挨着的上下左右都具有相同的G值 10。对角线方向的方块G值都是 14。

H值通过估算到红色目标方格的曼哈顿距离而得出。用这种方法得出的起点右侧方格到红色方格有 3 个方格远，则该方格H值就是 30。上面那个方格有 4 个方格远（注意只能水平和垂直移动），H就是 40。你可以大概看看其他方格的H值是怎么计算出来的。

每一个方格的F值，当然就不过是G和H值之和了。

继续搜索

为了继续搜索，我们简单的从开放列表中选择具有最小 F 值的方格，然后对选中的方格进行如下操作：

4.

将其从开放列表中移除，并加到封闭列表中。

5.

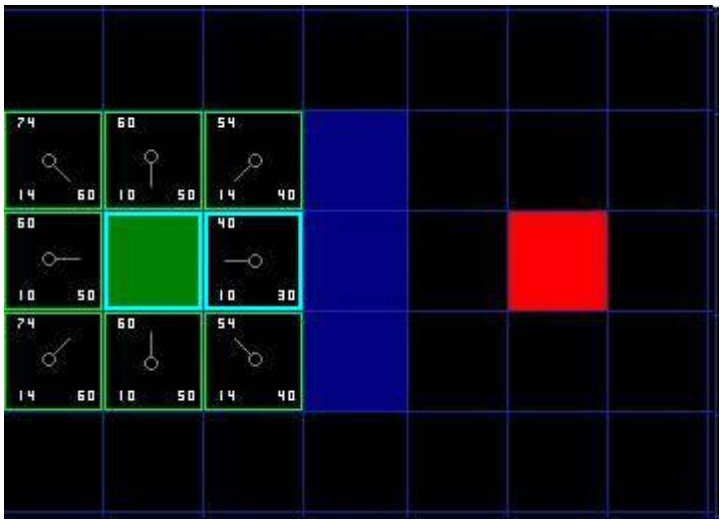
检验所有的相邻方格，忽略那些不可通过的或者已经在封闭列表里的方格。如果这个相邻方格不在开放列表中，就把它添加进去。并将当前选定方格设为新添方格的父方格。

6.

如果某个相邻方格已经在开放列表中了（意味着已经探测过，而且已经设置过父方格——译者），就看看有没有到达那个方格的更好的路径。也就是说，如果从当前选中方格到那个方格，会不会使那个方格的 G 值更小。如果不能，就不进行任何操作。相反的，如果新路径的 G 值更小，就将该相邻方格的父方格重设为当前选中方格。

（在上图中是改变其指针的方向为指向选中方格。最后，重新计算那个相邻方格的 F 和 G 值。如果你看糊涂了，下面会有图解说明。

好啦，咱们来看看具体点的例子。在初始时的 9 个方块中，当开始方格被加到封闭列表后，开放列表里还剩 8 个方格。在这八个方格当中，位于起点方格右边的那个方格具有最小的 F 值 40。所以我们选择这个方格作为下一个中心方格。下图中它以高亮的蓝色表示。



[图四]

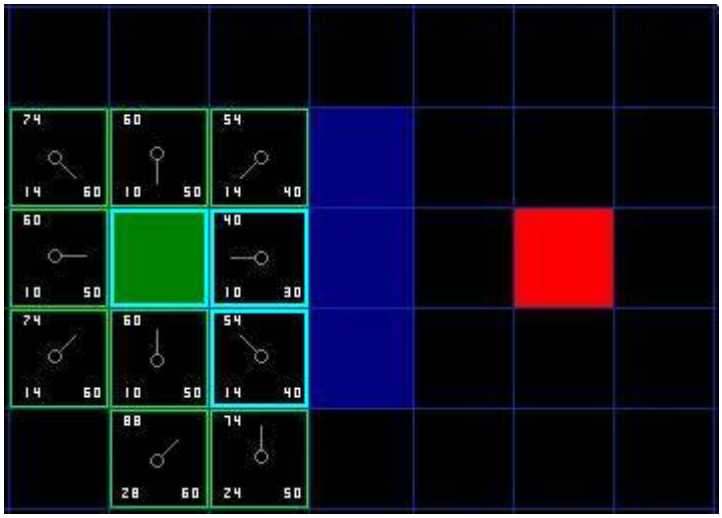
首先，我们将选中的方格从开放列表中移除，并加入到封闭列表中（所以用亮蓝色标记）。然后再检验它的相邻节点。那么在它紧邻的右边的方格都是墙，所以不管它们。左边挨着的是起始方格，而起始方格已经在封闭列表中了，所以我们也不管它。

其他四个方格已经在开放列表中，那么我们就要检验一下如果路径经由当前选中方格到那些方格的话会不会更好，当然，是用 G 值作为参考。来看看选中方格右上角的那一个方格，它当前的 G 值是 14，如果我们经由当前节点再到达那个方格的话，G 值会是 20（到当前方格的 G 值是 10，然后向上移动一格就再加上 10）。为 20 的 G 值比 14 大，因此这样的路径不会更好。你看看图就会容易理解些。显然从起始点沿斜角方向移动到那个方格比先水平移动一格再垂直移动一格更直接。

当我们按如上过程依次检验开放列表中的所有四个方格后，会发现经由当前方格的话不会形成更好的路径，那我们就保持目前的状况不变。现在我们已经处理了所有相邻方格，准备到下一个方格吧。

我们再遍历一下开放列表，目前只有 7 个方格了。我们挑个 F 值最小的吧。有趣的是，目前这种情况下，有两个 F 值为 54 的方格。那我们怎么选择呢？其实选哪个都没关系，要考虑到速度的话，选你最近加到开放列表中的那一个会更快些。当离目的地越来越近的时候越偏向于选最后发现的方格。实际上这个真的没关系（对待这个的不同造成了两个版本的 A*算法得到等长的不同路径）。

那我们选下面的那个好了，就是起始方格右边的，下图所示的那个。



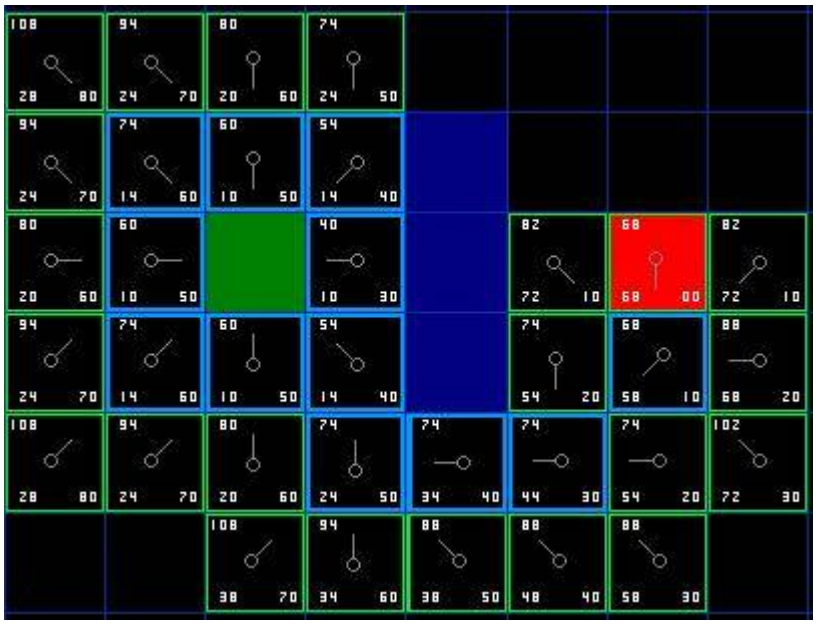
[图五]

这一次，在我们检验相邻方格的时候发现右边紧挨的那个是墙，就不管它了。上面挨着的那个也同样忽略。还有右边墙下面那个方格我们也不管。为什么呢？因为你不可能切穿墙角直接到达那个格子。实际上你得先向下走然后再通过那个方格。这个过程中是绕着墙角走。（注意：穿过墙角的这个规则是可选的，取决于你的节点是如何放置的。）

那么还剩下其他五个相邻方格。当前方格的下面那两个还不在开放列表中，那我们把它们加进去并且把当前方格作为它们的父方格。其他三个中有两个已经在封闭列表中了（两个已经在图中用亮蓝色标记了，起始方格，上面的方格），所以就不用管了。最后那个，当前方格左边挨着的，要检查一下经由当前节点到那里会不会降低它的 G 值。结果不行，所以我们又处理完毕了，然后去检验开放列表中的下一个格子。

重复这个过程直到我们把目的方格加入到开放列表中了，那时候看起来会像下图这

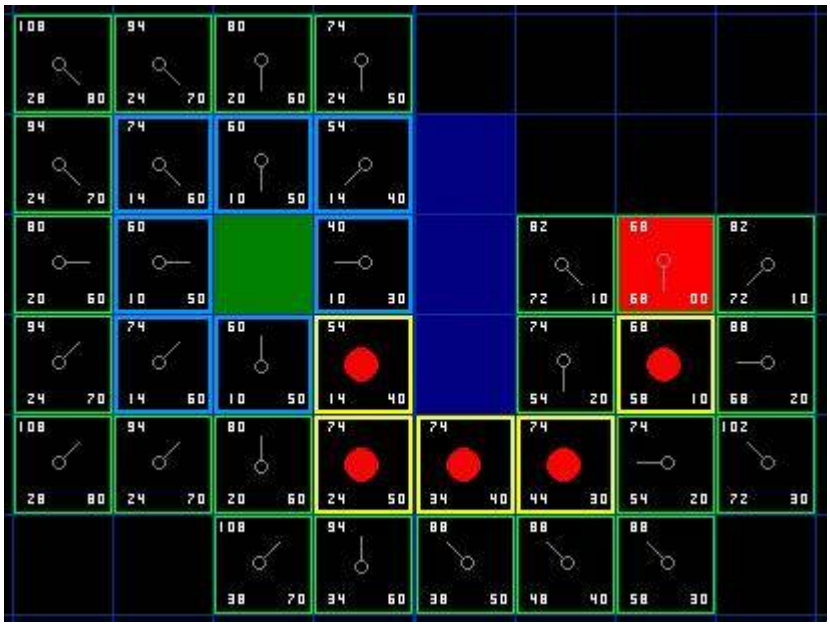
个样子。



[图六]

注意到没？起始方格下两格的位置，那里的格子已经和前一张图不一样了。之前它的 G 值是 28 并且指向右上方的那个方格。现在它的 G 值变成了 20 并且指向了正上方的方格。这个改变是在搜索过程中，它的 G 值被核查时发现在某个新路径下可以变得更小时发生的。然后它的父方格也被重设并且重新计算了 G 值和 F 值。在本例中这个改变看起来好像不是很重要，但是在很多种情况下这种改变会使到达目标的最佳路径变得非常不同。

那么我们怎样来自动得出实际路径的呢?很简单，只要从红色目标方格开始沿着每一个方格的指针方向移动，依次到达它们的父方格，最终肯定会到达起始方格。那就是你的路径！如下图所示。从 A 方格到 B 方格的移动就差不多是沿着这个路径从每个方格中心（节点）移动到另一个方格中心，直到抵达终点。简单吧！



[图七]

A*算法总结

1.
将开始节点放入开放列表(开始节点的 F 和 G 值都视为 0);
2.
重复一下步骤:

i.
在开放列表中查找具有最小 F 值的节点,并把查找到的节点作为当前节点;

ii.
把当前节点从开放列表删除, 加入到封闭列表;

iii.
对当前节点相邻的每一个节点依次执行以下步骤:

1.
如果该相邻节点不可通行或者该相邻节点已经在封闭列表中,则什么操作也不执行,继续检验下一个节点;
2.
如果该相邻节点不在开放列表中,则将该节点添加到开放列表中, 并将该相邻节点的父节点设为当前节点,同时保存该相邻节点的 G 和 F 值;
3.
如果该相邻节点在开放列表中, 则判断若经由当前节点到达该相邻节点的 G 值是否小于原来保存的 G 值,若小于,则将该相邻节点的父节点设为当前节点,并重新设置该相邻节点的 G 和 F 值.

iv.

循环结束条件:

当终点节点被加入到开放列表作为待检验节点时, 表示路径被找到,此时应终止循环;

或者当开放列表为空,表明已无可以添加的新节点,而已检验的节点中没有终点节点则意味着路径无法被找到,此时也结束循环;

3.

从终点节点开始沿父节点遍历, 并保存整个遍历到的节点坐标,遍历所得的节点就是最后得到的路径;

一点感慨

原谅我的离题。但毕竟值得指出的是，当你在网上和论坛里看到很多讨论A*路径寻找算法的东西时，偶尔会遇到一些人所指的A*算法并不是真正的A*的情况。实际上要应用真正的A*需要包含我们上面讨论的那些元素：专门的开放列表和封闭列表，用F、G和H值来排序的路径统计。有很多其他的寻找路径算法，但是其他的都不是A*，而A*一般认为是这些算法当中最好的。本文末尾的参考文档里有Bryan Stout对这些算法的讨论。在特定情况下某些算法可能会更好，但是你至少要理解你要准备做什么。好了，差不多了，还是回到主题吧。

实现时的注意事项

现在你已经理解了基本的算法，下面将讨论一些你在写自己的程序时要考虑的东西。下面一些材料和我用C++和Blitz Basic写的程序有关，但是那些要点是适用于任何其他语言的。

1.

开放列表的维护：实际上是A*算法中最耗时的因素。每次你处理开放列表时你都要从中找出具有最小F值的方格。有几种方法可以做到。你可以保存所需的路径元素，简单的遍历列表来找出最小F值的方格。这种方法是很简单，但是在路径很长的时候会很慢。也可以改进一下，变成维护一个有序列表，这样在需要最小F值方格的时候仅须获取该有序列表的第一个元素。我在写我那个程序的时候，开始就是用的这个办法。

这种方法很适合于地图不大的情况。但这还不是最快的解决办法。真正追求速度的认真的程序员往往会使用二叉堆。这也是我在我的代码中用的方法。据我的经验，这种方法会比大多数解决方案快至少 2－3 倍，在路径很长的时候更快（快 10 倍以上）。如果你有兴趣了解更多二叉堆的内容，去看看我的这篇文章，链接标记Using Binary Heaps in A* Pathfinding。

2.

其他单位。如果你仔细看了我的例子代码，你会注意到它完全忽略了地图上的其他单位。我的寻路者实际上是直接穿过彼此的。这样行得通行不通是取决于游戏的。如果你要考虑地图上的其他单位并且它们还能够移动，我建议你在路径寻找算法中

忽略它们而另外写一些代码去检测它们是否发生了碰撞。当碰撞发生时，你可以马上生成一个新的路径或者应用一些标准移动规则（比如总是靠右走等等）直到路径上不再有障碍物或，然后再生成一个新的路径。为什么在你最初计算路径的时候就把这些单位考虑进去呢？那其实是因为其他单位也会移动，它们的位置在不停的改变。这样会导致产生一些不可思议的结果，比如某个单位会突然转向来避免一个实际上已经不在那儿的另一个单位，或者撞上一个后来移动到它的预定路径上的单位。在寻路算法中忽略其他单位意味着你将不得不写一些单独的代码来避免冲突。这完全是由游戏特性决定的。所以我把解决方案留给你自己去琢磨。本文末尾参考文章一节Bryan Stout的一些地方提供了几个解决方案（比如鲁棒式跟踪等等），不妨去看看。

3.
一些提速技巧：你在开发你自己的A*程序或者改变我写的那个时，总会发现这个寻路算法很耗CPU，特别是地图上有大量的寻路者和地图相当大的时候。如果你在网上读过一些资料你会知道这个对于像星际争霸或者帝国时代这样专业的游戏也不例外。当你发现你的寻路算法使你写的东西变慢了的话，可以参考下面这些提速的办法：

用小一点的地图或者寻路者少一点。

不要在同一时间为很多个寻路者计算路径。不如把它们放进一个队列里并分散在几个游戏循环中。如果你的游戏运行时大约是，比如说，40 个循环/秒的话，没人会注意到。但是要是有一堆寻路者在同一时间计算路径而导致游戏在短时间内变慢的话，就非常引人注意了。

将地图的区块划分得大一些。这样会减少搜索路径时的区块总数。如果你够强，可以设计两种以上的寻路系统来适应于路径长度不同的情况。这正是那些专业人士所做的，路径长的时候用大区块，当接近目标路径变短的时候又用小区块来进行精确些的搜索。要是你对这个概念感兴趣，不妨读读我的文章链接标记Two-Tiered A* Pathfinding
考虑采用路标系统来处理长路径的情况，或者预先处理一些对游戏来说很少变化的路径。
对地图进行预处理并计算出哪些区域是从其他地方根本到达不了的。我把这些区域叫做“岛”。实际中这些区域可能是岛也可能是墙围起来的地方。A*算法的一个缺陷是当你在找到这样一个不可达区域的路径时，几乎会把整个地图都搜个遍，直到地图上每一个区块都被搜索过了才会停。这样会浪费很多CPU时间。解决这个问题的办法是预先得出哪些区域是根本不可达的（通过洪泛法或者其他方式），用一个数组记录下这些数据并在寻找路径前先检查一下。在我那个Blitz版本的代码中，我创建了一个地图预处理器来完成这样的处理。这个预处理器还能提前找到可在寻路算法中忽略的死角，因而大大提高了算法速度。
4.
地形多样化的开销：在这个入门教程和我附带的程序中，地形地貌只包括两种情况：可通过的和不可通过的。但是假如还有一些地形那是可以通过的只是移动的开销更

大一点呢？比如沼泽、山丘、地牢里的梯子等等呢？这些都是可以通行的地形的例子，只不过比开阔的平地具有更高的开销。类似的，公路地形会比路郊的移动开销小。
这个问题很容易解决。只要在计算给定区块的G值时把地形的开销加上去就行。把某个区块加上额外的开销，A*算法仍然有效。在我描述的那个简单例子里，地形只分可通过和不可通过两种，A*算法会找到最直接最短的路径。但当在一个地形多样化的环境里时，最小开销路径可能会是比较长的一段路程。例如从公路上绕过去显然比直接通过沼泽地开销要小。
还有一个有趣的办法是专业人士们叫做“感应映射”的东西。如同上面描述的多样化地形开销一样，你可以创建一个额外的点数制度并将之应用于AI方面的路径中。假设在一张地图上有一堆家伙在守卫一个山区要道，每次电脑派遣某人通过那条要道时都会被困住。那你就可以创建一个感应地图使得发生很多战斗冲突的那些区块开销增大，以此来告知电脑去选一个更安全的路径，并避免仅凭着是最短路径（但是更危险）就持续派遣人员通过某条路径的愚蠢情形。

5.
处理未探索区域：你有玩过电脑总是知道怎么走，甚至地图都还没探索过的PC游戏吗？由此看来这个路径寻找算法真是好得不现实。还好这个问题可以很容易解决。方法就是为不同的玩家和电脑（不是每个单位，不然会耗掉好多内存）创建一个独立的“已知可通行”数组。每个数组包含了玩家已探索区域和未知区域的信息，并把未知区域全部视为可通行区域来对待，除非后来发现是其他的地形。使用这种方法的话，移动单位就会绕到死角或犯类似错误，直到它们发现了周围的路。一旦地图都被探索过了，路径寻找算法就会正常运行了。
6.
平滑路径：A*算法会得出开销最小路程最短的路径，但没法得到看起来最平滑的路径。看看那个例子的最终路径（图七），该路径的第一步是起始点的右下方的方格。如果第一步是正下方那个方格，那得到的路径不是就要平滑很多吗？
有几种方法可以处理这个问题。如在计算路径的时候你可以给那些改变了路径方向的方格一个额外开销，使其G值增大。这样，你可以沿着所得路径看看哪些取舍能使你的路径看起来更平滑。对于这个话题，可以看看链接标记Toward More Realistic Pathfinding（是免费的，但是查看需要注册）来了解更多。

7.
非方格的搜索区域：在上面的例子中，我们用的是简单的二维方格布局。你可以不这样弄，不规则区域也行。想想棋盘游戏Risk和Risk里的那些国家。你可以设计一个那样的寻路的关卡。要做这样的关卡，你得创建一个查找表来存储每个国家对应的邻国，和从一个国家移动到另一个国家的开销G。另外也得选一种计算估计值H的方法。其他的处理就和前面的例子差不多了。只是当对新区域进行检验时，即添加新项目到开放列表中的时候，是从表中查找邻近国家而不是选择邻近方格。
类似的，你可以针对固定地形的地图来创建一个路标系统。路标通常是横穿路径的点，可能是在公路上也可能是在地牢的主隧道里。对于游戏设计者来说，需要提前设置好这些路标。如果两个路标所成的直线路径之间没有障碍物的话就称之为“相邻

的”。在Risk游戏的例子中，你会将相邻信息保存在一个查找表中，并会在新增开放列表元素时用到这个查找表。然后你会用到相关的G值（可能通过两个节点间的直线距离得到）和H值（可能通过该节点到终点的直线距离得到）其他的运算就和普通的差不多。
另外一个使用非方格搜索区域的斜视角RPG游戏的例子参见我的文章链接标记Two-Tiered A* Pathfinding。

进阶阅读

好了！现在你基本上掌握了基础知识，并对高级的概念也有了些印象。在此我建议把我的代码拿来研究研究。压缩包里有两个版本，分别是C++的和Blitz Basic的。它们的注释都很详细，我想应该很容易理解。下面是链接：

链接标记Sample Code: A* Pathfinder (2D) Version 1.71

如果你没法用C++或者Blitz Basic，在C++版里有两个程序文件可以直接运行。而Blitz Basic版本要在链接标记Blitz Basic的网站上下载免费演示版的Blitz Basic 3D才能运行。链接标记这里还有Ben O’Neill写的在线示例。
你也应通读一下下面的网页。这些在你读了本文之后应该很好理解了。

链接标记Amit's A* Pages:这是Amit Patel的一篇广为引用的文章。不过如果你没读先读本文的话看他这个可能会有些困惑。非常值得一读。特别是Amit对此的独特观点。

链接标记Smart Moves: Intelligent Path Finding: Gamasutra.com网站上Bryan的这篇文章是需要注册才能阅读的。不过注册是免费的而且为了这篇文章麻烦一点也值得。Bryan用Delphi写的那个程序帮助我了解了A*，那也是我的A*程序的灵感来源。这篇文章也写了一些A*的变种
链接标记Terrain Analysis:这是一篇高阶的文章，不过很有趣，它是由Ensemble Studios的专家Dave Pottinger所写。这家伙负责协调帝国时代II：Age of Kings的开发。要想把这里所有东西都看明白是不可能的，不过它或许会给你自己的项目带来一些有趣的想法。文章还包括一些关于材质贴图，感应映射和其他一些高级AI/路径寻找概念。其中关于“洪泛（flood filling）”的讨论是我的死角和岛屿等地图预处理代码的灵感来源。在我的Blitz Basic版的程序中包含了这个东西。。

其他值得一读的文章：

链接标记aiGuru: Pathfinding
链接标记Game AI Resource: Pathfinding
链接标记GameDev.net: Pathfinding

好了！如果你在写程序的时候用到了任何以上的观念，我将非常高兴。你可以通过Email联系我：

就到这儿吧，Good luck!

最后贴上eidiot的代码，原配注释，大家一起学习下~~

这里是eidiot的教程：链接标记http://eidiot.net/2007/04/17/a-star-pathfinding/