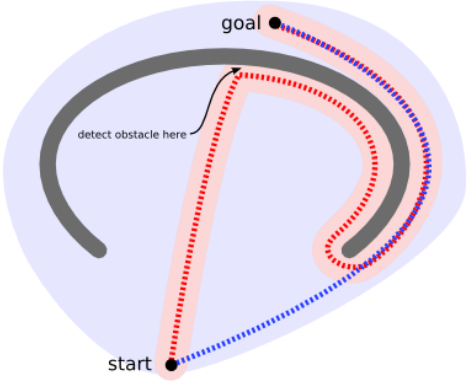
浅谈A*路径规划算法

原文地址：http://theory.stanford.edu/~amitp/GameProgramming/

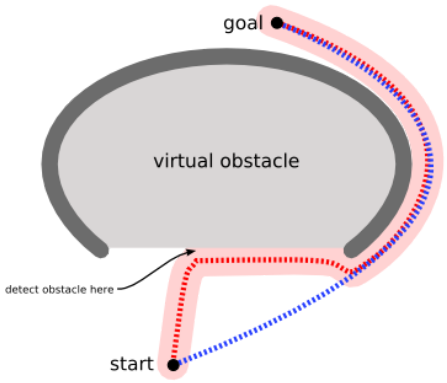
1 导言

移动一个简单的物体（object）看起来是容易的。而路径搜索是复杂的。为什么涉及到路径搜索就产生麻烦了？考虑以下情况：



物体（unit）最初位于地图的底端并且尝试向顶部移动。物体扫描的区域中(粉红色部分)没有任何东西显示它不能向上移动，因此它持续向上移动。在靠近顶部时，它探测到一个障碍物然后改变移动方向。然后它沿着U形障碍物找到它的红色的路径。相反的，一个路径搜索器（pathfinder）将会扫描一个更大的区域（淡蓝色部分），但是它能做到不让物体(unit)走向凹形障碍物而找到一条更短的路径(蓝色路径）。

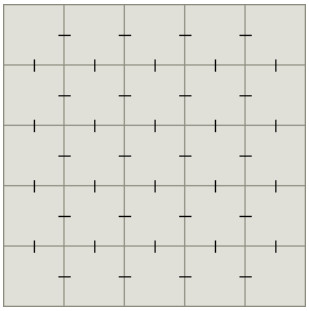
然而你可以扩展一个运动算法，用于对付上图所示的障碍物。或者避免制造凹形障碍，或者把凹形出口标识为危险的(只有当目的地在里面时才进去):



比起一直等到最后一刻才发现问题，路径搜索器让你提前作出计划。不带路径搜索的运动(movement)可以在很多种情形下工作，同时可以扩展到更多的情形，但是路径搜索是一种更常用的解决更多问题的方法。

1.1 算法

计算机科学教材中的路径搜索算法在数学视角的图上工作——由边联结起来的结点的集合。一个基于图块(tile)拼接的游戏地图可以看成是一个图，每个图块(tile)是一个结点，并在每个图块之间画一条边：

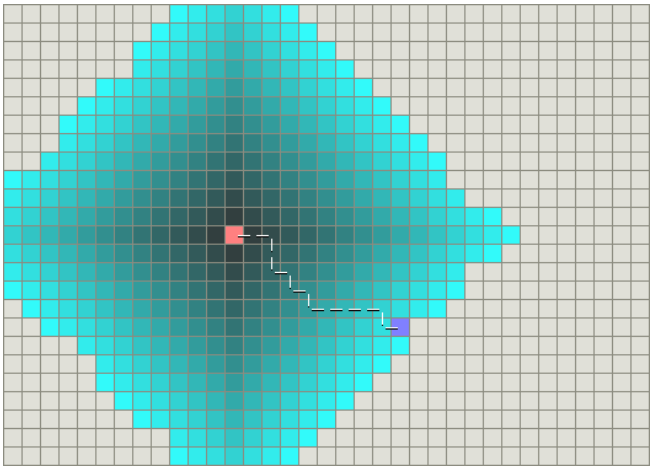


目前，我会假设我们使用二维网格(grid)。稍后我将讨论如何在你的游戏之外建立其他类型的图。

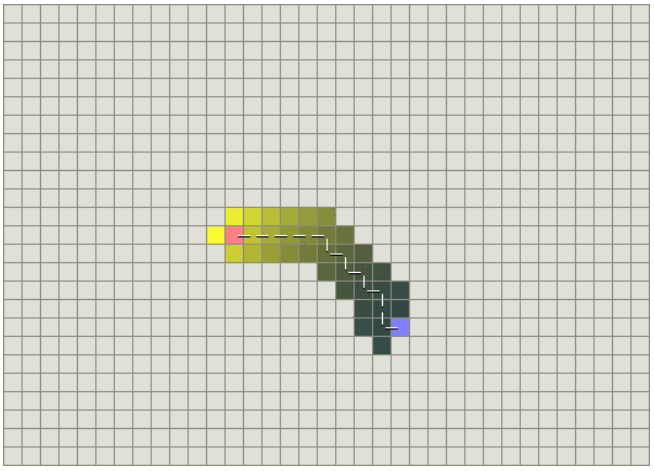
许多AI领域或算法研究领域中的路径搜索算法是基于任意(arbitrary)的图设计的，而不是基于网格(grid-based)的 图。我们可以找到一些能使用网格地图的特性的东西。有一些我们认为是常识，而算法并不理解。例如，我们知道一些和方向有关的东西：一般而言，如果两个物体 距离越远，那么把其中一个物体向另一个移动将花越多的时间；并且我们知道地图中没有任何秘密通道可以从一个地点通向另一个地点。（我假设没有，如果有的 话，将会很难找到一条好的路径，因为你并不知道要从何处开始。）

1.2 Dijkstra算法与最佳优先搜索

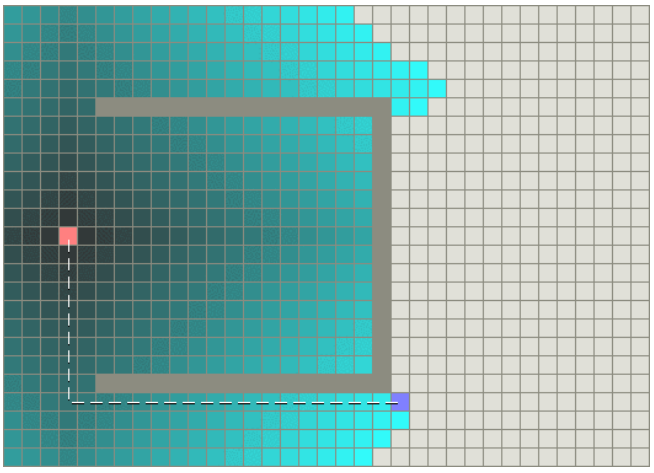
Dijkstra算法从物体所在的初始点开始，访问图中的结点。它迭代检查待检查结点集中的结点，并把和该结点最靠近的尚未检查的结点加入待检查结点集。该结点集从初始结点向外扩展，直到到达目标结点。Dijkstra算法保证能找到一条从初始点到目标点的最短路径，只要所有的边都有一个非负的代价值。（我说“最短路径”是因为经常会出现许多差不多短的路径。）在下图中，粉红色的结点是初始结点，蓝色的是目标点，而类菱形的有色区域（注：原文是teal areas）则是Dijkstra算法扫描过的区域。颜色最淡的区域是那些离初始点最远的，因而形成探测过程（exploration）的边境（frontier）：



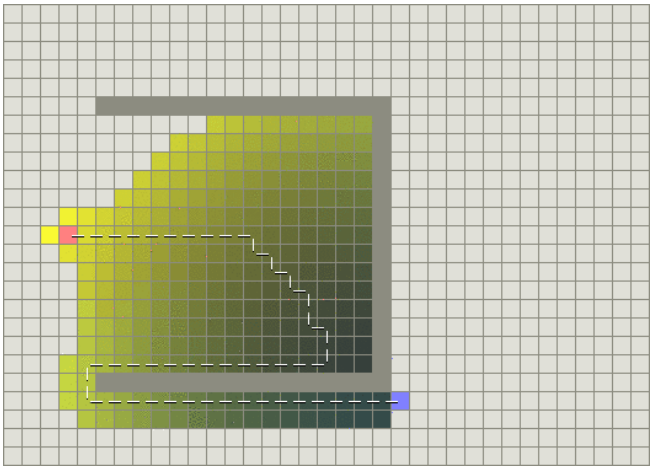
最佳优先搜索（BFS）算法按照类似的流程运行，不同的是它能够评估（称为启发式的）任意结点到目标点的代价。与选择离初始结点最近的结点不同的是，它选择离目标最近的结点。BFS不能保证找到一条最短路径。然而，它比Dijkstra算法快的多，因为它用了一个启发式函数（heuristic ）快速地导向目标结点。例如，如果目标位于出发点的南方，BFS将趋向于导向南方的路径。在下面的图中，越黄的结点代表越高的启发式值（移动到目标的代价高），而越黑的结点代表越低的启发式值（移动到目标的代价低）。这表明了与Dijkstra 算法相比，BFS运行得更快。



然而，这两个例子都仅仅是最简单的情况——地图中没有障碍物，最短路径是直线的。现在我们来考虑前边描述的凹型障碍物。Dijkstra算法运行得较慢，但确实能保证找到一条最短路径：



另一方面，BFS运行得较快，但是它找到的路径明显不是一条好的路径：



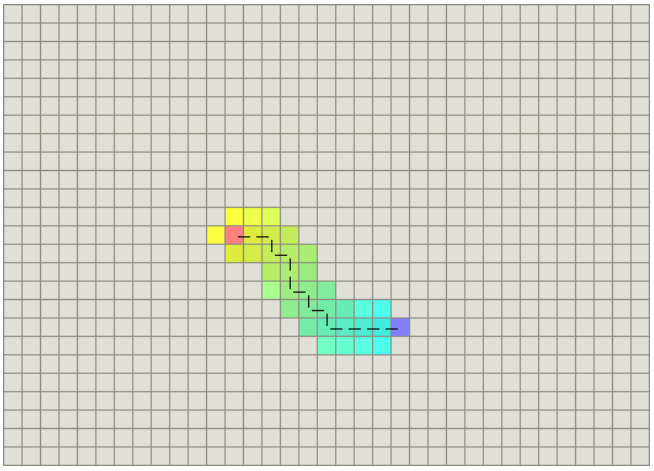
问题在于BFS是基于贪心策略的，它试图向目标移动尽管这不是正确的路径。由于它仅仅考虑到达目标的代价，而忽略了当前已花费的代价，于是尽管路径变得很长，它仍然继续走下去。

结合两者的优点不是更好吗？1968年发明的A*算法就是把启发式方法（heuristic approaches）如BFS，和常规方法如Dijsktra算法结合在一起的算法。有点不同的是，类似BFS的启发式方法经常给出一个近似解而不是保证最佳解。然而，尽管A*基于无法保证最佳解的启发式方法，A*却能保证找到一条最短路径。

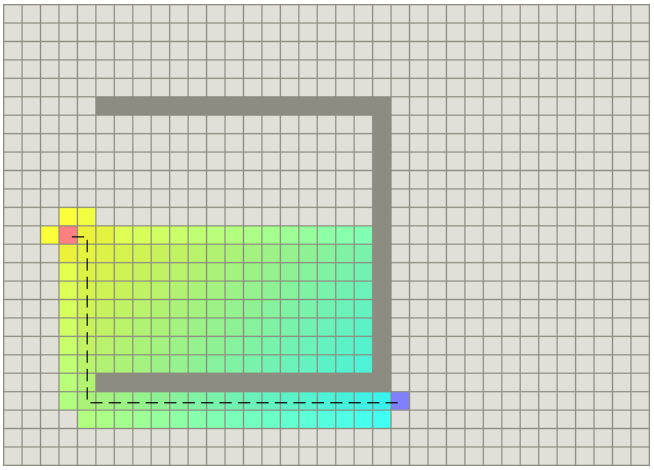
1.3 A*算法

我将集中讨论A*算法。A*是路径搜索中最受欢迎的选择，因为它相当灵活，并且能用于多种多样的情形之中。

和其它的图搜索算法一样，A*潜在地搜索图中一个很大的区域。和Dijkstra一样，A*能用于搜索最短路径。和BFS一样，A*能用启发式函数（注：原文为heuristic）引导它自己。在简单的情况中，它和BFS一样快。



在凹型障碍物的例子中，A*找到一条和Dijkstra算法一样好的路径：



成功的秘决在于，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*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）。在上图中，yellow(h)表示远离目标的结点而teal(g)表示远离初始点的结点。当从初始点向目标点移动时，A*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)。

2 启发式算法

启发式函数h(n)告诉A*从任意结点n到目标结点的最小代价评估值。选择一个好的启发式函数是重要的。

2.1 A*对启发式函数的使用

启发式函数可以控制A*的行为：

- 一种极端情况，如果h(n)是0，则只有g(n)起作用，此时A*演变成Dijkstra算法，这保证能找到最短路径。
- 如果h(n)经常都比从n移动到目标的实际代价小（或者相等），则A*保证能找到一条最短路径。h(n)越小，A*扩展的结点越多，运行就得越慢。
- 如果h(n)精确地等于从n移动到目标的代价，则A*将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等（译者：指让h(n)精确地等于实际值）。只要提供完美的信息，A*会运行得很完美，认识这一点很好。
- 如果h(n)有时比从n移动到目标的实际代价高，则A*不能保证找到一条最短路径，但它运行得更快。
- 另一种极端情况，如果h(n)比g(n)大很多，则只有h(n)起作用，A*演变成BFS算法。

所以我们得到一个很有趣的情况，那就是我们可以决定我们想要从A*中获得什么。理想情况下（注：原文为At exactly the right point），我们想最快地得到最短路径。如果我们的目标太低，我们仍会得到最短路径，不过速度变慢了；如果我们的目标太高，那我们就放弃了最短路径，但A*运行得更快。

在游戏中，A*的这个特性非常有用。例如，你会发现在某些情况下，你希望得到一条好的路径（"good" path）而不是一条完美的路径（"perfect" path）。为了权衡g(n)和h(n)，你可以修改任意一个。

注:在学术上，如果启发式函数值是对实际代价的低估，A*算法被称为简单的A算法（原文为simply A）。然而，我继续称之为A*，因为在实现上是一样的，并且在游戏编程领域并不区别A和A*。

2.2 速度还是精确度?

A*改变它自己行为的能力基于启发式代价函数，启发式函数在游戏中非常有用。在速度和精确度之间取得折衷将会让你的游戏运行得更快。在很多游戏中，你并不真正需要得到最好的路径，仅需要近似的就足够了。而你需要什么则取决于游戏中发生着什么，或者运行游戏的机器有多快。

假设你的游戏有两种地形，平原和山地，在平原中的移动代价是1而在山地则是3。A* is going to search three times as far along flat land as it does along mountainous land. 这是因为有可能有一条沿着平原到山地的路径。把两个邻接点之间的评估距离设为1.5可以加速A*的搜索过程。然后A*会将3和1.5比较，这并不比把3和1比较差。It is not as dissatisfied with mountainous terrain, so it won't spend as much time trying to find a way around it. Alternatively, you can

speed up up A*'s search by decreasing the amount it searches for paths around mountains–just tell A* that the movement cost on mountains is 2 instead of 3. Now it will search only twice as far along the flat terrain as along mountainous terrain. Either approach gives up ideal paths to get something quicker.

速度和精确度之间的选择前不是静态的。你可以基于CPU的速度、用于路径搜索的时间片数、地图上物体（units）的数量、物体的重要性、组（group）的大小、难度或者其他任何因素来进行动态的选择。取得动态的折衷的一个方法是，建立一个启发式函数用于假定通过一个网格空间的最小代价是1，然后建立一个代价函数（cost ）用于测量（scales）：

g’（n) = 1 + alpha * (g(n) – 1)

如果alpha是0，则改进后的代价函数的值总是1。这种情况下，地形代价被完全忽略，A*工作变成简单地判断一个网格可否通过。如果alpha是1，则最初的代价函数将起作用，然后你得到了A*的所有优点。你可以设置alpha的值为0到1的任意值。

你也可以考虑对启发式函数的返回值做选择：绝对最小代价或者期望最小代价。例如，如果你的地图大部分地形是代价为2的草地，其它一些地方是代价为1的道路，那么你可以考虑让启发式函数不考虑道路，而只返回2*距离。

速度和精确度之间的选择并不是 全局的。在地图上的某些区域，精确度是重要的，你可以基于此进行动态选择。例如，假设我们可能在某点停止重新计算路径或者改变方向，则在接近当前位置的地 方，选择一条好的路径则是更重要的，因此为何要对后续路径的精确度感到厌烦？或者，对于在地图上的一个安全区域，最短路径也许并不十分重要，但是当从一个 敌人的村庄逃跑时，安全和速度是最重要的。（译者注：译者认为这里指的是，在安全区域，可以考虑不寻找精确的最短路径而取近似路径，因此寻路快；但在危险 区域，逃跑的安全性和逃跑速度是重要的，即路径的精确度是重要的，因此可以多花点时间用于寻找精确路径。）

2.3 衡量单位

A*计算f(n) = g(n) + h(n)。为了对这两个值进行相加，这两个值必须使用相同的衡量单位。如果g(n)用小时来衡量而h(n)用米来衡量，那么A*将会认为g或者h太大或者太小，因而你将不能得到正确的路径，同时你的A*算法将运行得更慢。

2.4 精确的启发式函数

如果你的启发式函数精确地等于实际最佳路径（optimal path），如下一部分的图中所示，你会看到此时A*扩展的结点将非常少。A*算法内部发生的事情是：在每一结点它都计算f(n) = g(n) + h(n)。当h(n)精确地和g(n)匹配（译者注：原文为match）时，f(n)的值在沿着该路径时将不会改变。不在正确路径（right path）上的所有结点的f值均大于正确路径上的f值（译者注：正确路径在这里应该是指最短路径）。如果已经有较低f值的结点，A*将不考虑f值较高的结点，因此它肯定不会偏离最短路径。

2.4.1 预计算的精确启发式函数

构造精确启发函数的一种方法是预先计算任意一对结点之间最短路径的长度。在许多游戏的地图中这并不可行。然后，有几种方法可以近似模拟这种启发函数：

- Fit a coarse grid on top of the fine grid. Precompute the shortest path between any pair of coarse grid locations.
- Precompute the shortest path between any pair of waypoints. This is a generalization of the coarse grid approach.

（译者：此处不好翻译，暂时保留原文）

然后添加一个启发函数h’用于评估从任意位置到达邻近导航点（waypoints）的代价。（如果愿意，后者也可以通过预计算得到。）最终的启发式函数可以是：

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

或者如果你希望一个更好但是更昂贵的启发式函数，则分别用靠近结点和目标的所有的w1,w2对对上式进行求值。（译者注：原文为or if you want a better but more expensive heuristic, evaluate the above with all pairs w1, w2 that are close to the node and the goal, respectively.)

2.4.2 线性精确启发式算法

在特殊情况下，你可以不通过预计算而让启发式函数很精确。如果你有一个不存在障碍物和slow地形，那么从初始点到目标的最短路径应该是一条直线。

如果你正使用简单的启发式函数（我们不知道地图上的障碍物），则它应该和精确的启发式函数相符合（译者注：原文为match）。如果不是这样，则你会遇到衡量单位的问题，或者你所选择的启发函数类型的问题。

2.5 网格地图中的启发式算法

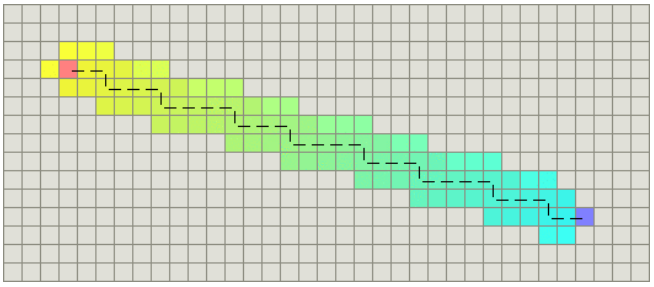
在网格地图中，有一些众所周知的启发式函数。

2.5.1 曼哈顿距离

标准的启发式函数是曼哈顿距离（Manhattan distance）。考虑你的代价函数并找到从一个位置移动到邻近位置的最小代价D。因此，我的游戏中的启发式函数应该是曼哈顿距离的D倍：

H(n) = D * (abs (n.x – goal.x) + abs (n.y – goal.y))

你应该使用符合你的代价函数的衡量单位。



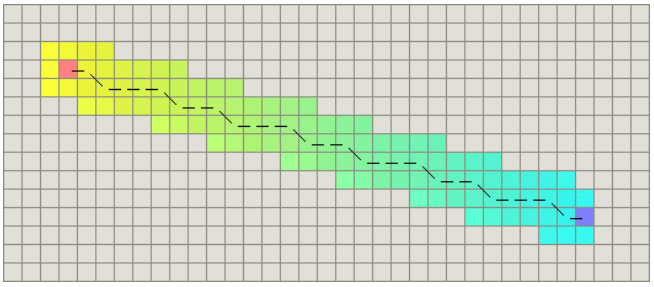
(Note: the above image has a tie-breaker added to the heuristic.}

（译者注：曼哈顿距离——两点在南北方向上的距离加上在东西方向上的距离，即D（I，J）=|XI-XJ|+|YI-YJ|。对于一个具有正南正北、正东正西方向规则布局的城镇街道，从一点到达另一点的距离正是在南北方向上旅行的距离加上在东西方向上旅行的距离因此曼哈顿距离又称为出租车距离，曼哈顿距离不是距离不变量，当坐标轴变动时，点间的距离就会不同——百度知道）

2.5.2 对角线距离

如果在你的地图中你允许对角运动那么你需要一个不同的启发函数。（4 east, 4 north）的曼哈顿距离将变成8*D。然而，你可以简单地移动（4 northeast）代替，所以启发函数应该是4*D。这个函数使用对角线，假设直线和对角线的代价都是D：

h(n) = D * max(abs(n.x - goal.x), abs(n.y - goal.y))



如果对角线运动的代价不是D，但类似于D2 = sqrt(2) * D，则上面的启发函数不准确。你需要一些更准确（原文为sophisticated）的东西：

h_diagonal(n) = min(abs(n.x - goal.x), abs(n.y - goal.y))

h_straight(n) = (abs(n.x - goal.x) + abs(n.y - goal.y))

h(n) = D2 * h_diagonal(n) + D * (h_straight(n) - 2*h_diagonal(n)))

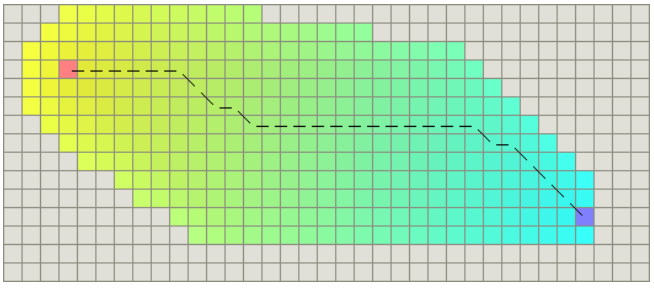
这里，我们计算h_diagonal(n)：沿着斜线可以移动的步数；h_straight(n)：曼哈顿距离；然后合并这两项，让所有的斜线步都乘以D2，剩下的所有直线步(注意这里是曼哈顿距离的步数减去2倍的斜线步数)都乘以D。

2.5.3 欧几里得距离

如果你的单位可以沿着任意角度移动（而不是网格方向），那么你也许应该使用直线距离：

h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)

然而，如果是这样的话，直接使用A*时将会遇到麻烦，因为代价函数g不会match启发函数h。因为欧几里得距离比曼哈顿距离和对角线距离都短，你仍可以得到最短路径，不过A*将运行得更久一些：

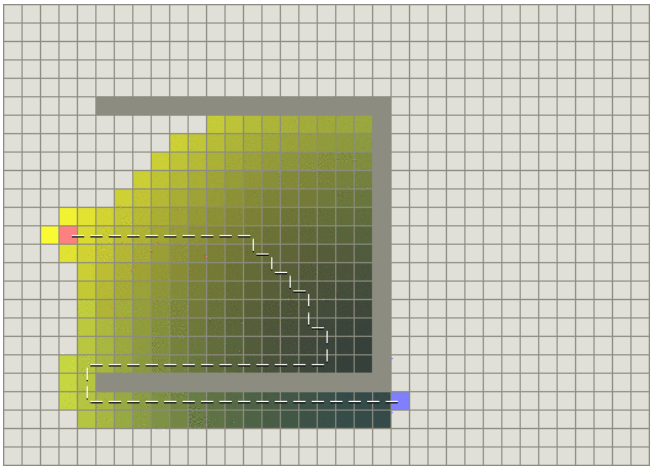


2.5.4 平方后的欧几里得距离

我曾经看到一些A*的网页，其中提到让你通过使用距离的平方而避免欧几里得距离中昂贵的平方根运算：

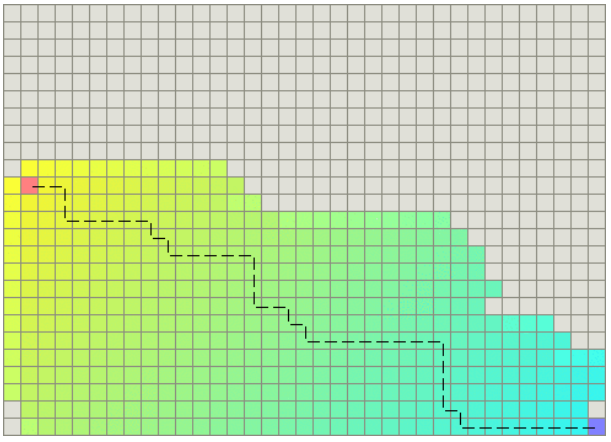
h(n) = D * ((n.x-goal.x)^2 + (n.y-goal.y)^2)

不要这样做！这明显地导致衡量单位的问题。当A*计算f(n) = g(n) + h(n)，距离的平方将比g的代价大很多，并且你会因为启发式函数评估值过高而停止。对于更长的距离，这样做会靠近g(n)的极端情况而不再计算任何东西，A*退化成BFS：



2.5.5 Breaking ties

导致低性能的一个原因来自于启发函数的ties（注：这个词实在不知道应该翻译为什么）。当某些路径具有相同的f值的时候，它们都会被搜索（explored），尽管我们只需要搜索其中的一条：



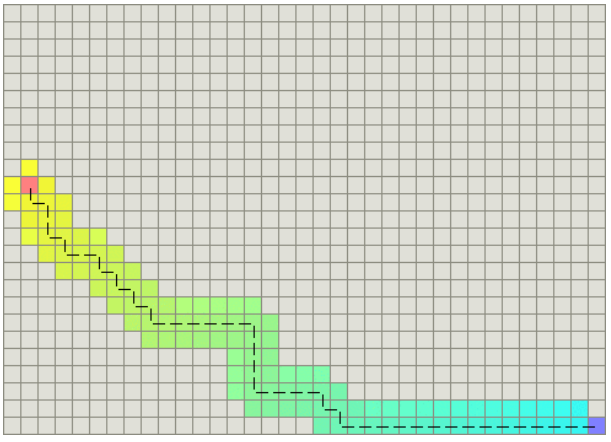
Ties in f values.

为了解决这个问题，我们可以为启发函数添加一个附加值（译者注：原文为small tie breaker）。附加值对于结点必须是确定性的（也就是说，不能是随机的数），而且它必须让f值体现区别。因为A*对f值排序，让f值不同意味着只有一个"equivalent"的f值会被检测。

一种添加附加值的方式是稍微改变（译者注：原文为nudge）h的衡量单位。如果我们减少衡量单位（译者注：原文为scale it downwards），那么当我们朝着目标移动的时候f将逐渐增加。很不幸，这意味着A*倾向于扩展到靠近初始点的结点，而不是靠近目标的结点。我们可以增加衡量单位（译者注：原文为scale it downwards scale h upwards slightly）（甚至是0.1%），A*就会倾向于扩展到靠近目标的结点。

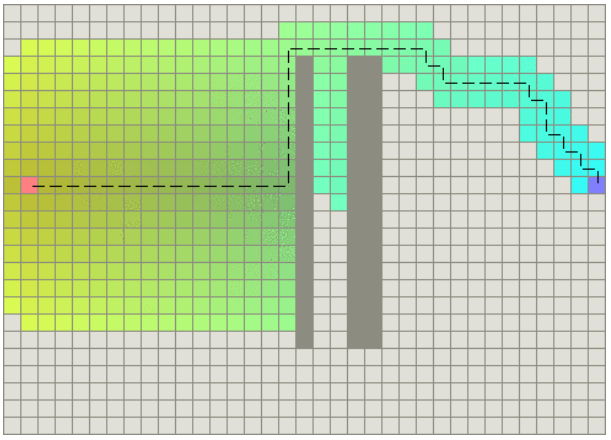
heuristic *= (1.0 + p)

选择因子p使得p < 移动一步（step）的最小代价 / 期望的最长路径长度。假设你不希望你的路径超过1000步（step），你可以使p = 1 / 1000。添加这个附加值的结果是，A*比以前搜索的结点更少了。



Tie-breaking scaling added to heuristic.

当存在障碍物时，当然仍要在它们周围寻找路径，但要意识到，当绕过障碍物以后，A*搜索的区域非常少：



Tie-breaking scaling added to heuristic, works nicely with obstacles.

Steven van Dijk建议，一个更直截了当的方法是把h传递到比较函数（comparison ）。当f值相等时，比较函数检查h，然后添加附加值。

一个不同的添加附加值的方法是，倾向于从初始点到目标点的连线（直线）：

dx1 = current.x - goal.x

dy1 = current.y - goal.y

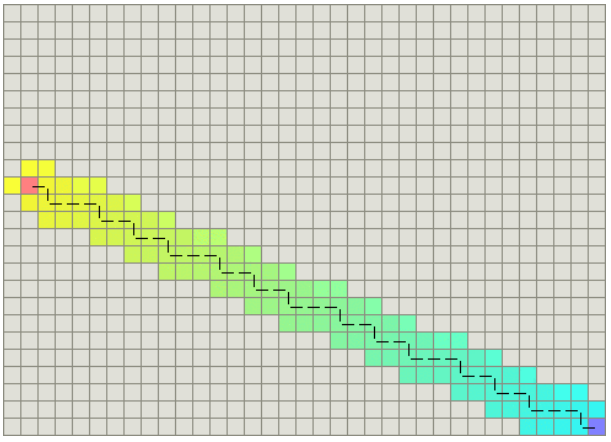
dx2 = start.x - goal.x

dy2 = start.y - goal.y

cross = abs(dx1*dy2 - dx2*dy1)

heuristic += cross*0.001

这段代码计算初始-目标向量（start to goal vector）和当前-目标向量（current point to goal vector）的向量叉积（vector cross-product）。When these vectors don't line up, the cross product will be larger.结果是，这段代码选择的路径稍微倾向于从初始点到目标点的直线。当没有障碍物时，A*不仅搜索很少的区域，而且它找到的路径看起来非常棒：



Tie-breaking cross-product added to heuristic, produces pretty paths.

然而，因为这种附加值倾向于从初始点到目标点的直线路径，当出现障碍物时将会出现奇怪的结果（注意这条路径仍是最佳的，只是看起来很奇怪）：

Tie-breaking cross-product added to heuristic, less pretty with obstacles.

为了交互地研究这种附加值方法的改进，请参考James Macgill的A*确applet（http://www.ccg.leeds.ac.uk/james/aStar/ ）[如果链接无效，请使用这个镜像（http://www.vision.ee.ethz.ch/~buc/astar/AStar.html）]（译者注：两个链接均无效）。使用“Clear”以清除地图，选择地图对角的两个点。当你使用“Classic A*”方法，你会看到附加值的效果。当你使用“Fudge”方法，你会看到上面给启发函数添加叉积后的效果。

然而另一种添加附加值的方法是，小心地构造你的A*优先队列，使新插入的具有特殊f值的结点总是比那些以前插入的具有相同f值的旧结点要好一些。

你也许也想看看能够更灵活地（译者注：原文为sophisticated）添加附加值的AlphA*算法（http://home1.stofanet.dk/breese/papers.html），不过用这种算法得到的路径是否能达到最佳仍在研究中。AlphA*具有较好的适应性，而且可能比我在上面讨论的附加值方法运行得都要好。然而，我所讨论的附加值方法非常容易实现，所以从它们开始吧，如果你需要得到更好的效果，再去尝试AlphA*。

2.5.6 区域搜索

如果你想搜索邻近目标的任意不确定结点，而不是某个特定的结点，你应该建立一个启发函数h’(x)，使得h’(x)为h1(x), h2(x), h3(x)。。。的最小值，而这些h1, h2, h3是邻近结点的启发函数。然而，一种更快的方法是让A*仅搜索目标区域的中心。一旦你从OPEN集合中取得任意一个邻近目标的结点，你就可以停止搜索并建立一条路径了。

3 Implementation notes

3.1 概略

如果不考虑具体实现代码，A*算法是相当简单的。有两个集合，OPEN集和CLOSED集。其中OPEN集保存待考查的结点。开始时，OPEN集只包含一个元素：初始结点。CLOSED集保存已考查过的结点。开始时，CLOSED集是空的。如果绘成图，OPEN集就是被访问区域的边境（frontier）而CLOSED集则是被访问区域的内部（interior）。每个结点同时保存其父结点的指针因此我们可以知道它是如何被找到的。

在主循环中重复地从OPEN集中取出最好的结点n（f值最小的结点）并检查之。如果n是目标结点，则我们的任务完成了。否则，结点n被从OPEN集中删除并加入CLOSED集。然后检查它的邻居n’。如果邻居n’在CLOSED集中，那么它是已经被检查过的，所以我们不需要考虑它*；如果n’在OPEN集中，那么它是以后肯定会被检查的，所以我们现在不考虑它*。否则，把它加入OPEN集，把它的父结点设为n。到达n’的路径的代价g(n’)，设定为g(n) + movementcost(n, n’)。

(*)这里我忽略了一个小细节。你确实需要检查结点的g值是否更小了，如果是的话，需要重新打开（re-open）它。

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): **
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

(**) This should never happen if you have an admissible heuristic. However in games we often have inadmissible heuristics.

3.2 源代码

我自己的（旧的）C++A*代码是可用的：path.cpp (http://theory.stanford.edu/~amitp/ GameProgramming/path.cpp)和path.h (http://theory.stanford.edu/~amitp/GameProgramming/ path.h)，但是不容易阅读。还有一份更老的代码（更慢的，但是更容易理解），和很多其它的A*实现一样，它在Steve Woodcock'的游戏AI页面（http://www.gameai.com/ai.html）。

在网上，你能找到C，C++，Visual Basic，Java(http://www.cuspy.com/software/pathfinder/ doc/)，Flash/Director/Lingo, C# (http://www.codeproject.com/csharp/CSharpPathfind.asp), Delphi, Lisp, Python, Perl, 和Prolog 实现的A*代码。一定的阅读Justin Heyes-Jones的C++实现（http://www.geocities.com/jheyesjones/astar.html）。

3.3 集合的表示

你首先想到的用于实现OPEN集和CLOSED集的数据结构是什么？如果你和我一样，你可能想到“数组”。你也可能想到“链表”。我们可以使用很多种不同的数据结构，为了选择一种，我们应该考虑我们需要什么样的操作。

在OPEN集上我们主要有三种操作：主循环重复选择最好的结点并删除它；访问邻居结点时需要检查它是否在集合里面；访问邻居结点时需要插入新结点。插入和删除最佳是优先队列（http://members.xoom.com/killough/heaps.html）的典型操作。

选择哪种数据结构不仅取决于操作，还取决于每种操作执行的次数。检查一个结点是否在集合中这一操作对每个被访问的结点的每个邻居结点都执行一次。删除最佳操作对每个被访问的结点都执行一次。被考虑到的绝大多数结点都会被访问；不被访问的是搜索空间边缘（*fringe*）的结点。当评估数据结构上面的这些操作时，必须考虑fringe(F)的最大值。

另外，还有第四种操作，虽然执行的次数相对很少，但还是必须实现的。如果正被检查的结点已经在OPEN集中（这经常发生），并且如果它的f值比已经在OPEN集中的结点要好（这很少见），那么OPEN集中的值必须被调整。调整操作包括删除结点（f值不是最佳的结点）和重插入。这两个步骤必须被最优化为一个步骤，这个步骤将移动结点。

3.3.1 未排序数组或链表

最简单的数据结构是未排序数组或链表。集合关系检查操作（Membership test）很慢，扫描整个结构花费O(F)。插入操作很快，添加到末尾花费O(1)。查找最佳元素（Finding the best element）很慢，扫描整个结构花费O(F)。对于数组，删除最佳元素（Removing the best element）花费O(F)，而链表则是O(1)。调整操作中，查找结点花费O(F)，改变值花费O(1)。

3.3.2 排序数组

为了加快删除最挂操作，可以对数组进行排序。集合关系检查操作将变成O(log F)，因为我们可以使用折半查找。插入操作会很慢，为了给新元素腾出空间，需要花费 O(F)以移动所有的元素。查找最佳元素操作会很快，因为它已经在末尾了所以花费是O(1)。如果我们保证最佳排序至数组的尾部（best sorts to the *end* of the array），删除最佳元素操作花费将是O(1)。调整操作中，查找结点花费O(logF)，改变值/位置花费O(F)。

3.3.3 排序链表

在排序数组中，插入操作很慢。如果使用链表则可以加速该操作。集合关系检查操作很慢，需要花费O(F)用于扫描链表。插入操作是很快的，插入新元素只花费O(1)时间，但是查找正确位置需要花费O(F)。查找最佳元素很快，花费O(1)时间，因为最佳元素已经在表的尾部。删除最佳元素也是O(1)。调整操作中，查找结点花费O(F)，改变值/位置花费O(1)。

3.3.4 排序跳表

在未排序链表中查找元素是很慢的。如果用跳表（http://en.wikipedia.org/wiki/Skip_list）代替链表的话，可以加速这个操作。在跳表中，如果有排序键（sort key）的话，集合关系检查操作会很快：O(log F)。如果你知道在何处插入的话，和链表一样，插入操作也是O(1)。如果排序键是f，查找最佳元素很快，达到O(1)，删除一个元素也是O(1)。调整操作涉及到查找结点，删除结点和重插入。

如果我们用地图位置作为跳表的排序键，集合关系检查操作将是O(log F)。在完成集合关系检查后，插入操作是O(1)。查找最佳元素是O(F)，删除一个结点是O(1)。因为集合关系检查更快，所以它比未排序链表要好一些。

如果我们用f值作为跳表的排序键，集合关系检查操作将是O(F)。插入操作是O(1)。查找最佳元素是O(1)，删除一个结点是O(1)。这并不比排序链表好。

3.3.5 索引数组

如果结点的集合有限并且数目是适当的，我们可以使用直接索引结构，索引函数i(n)把结点n映射到一个数组的索引。未排序与排序数组的长度等于OPEN集的最大值，和它们不同，对所有的n，索引数组的长度总是等于max(i(n))。如果你的函数是密集的（没有不被使用的索引），max(i(n))将是你地图中结点的数目。只要你的地图是网格的，让索引函数密集就是容易的。

假设i(n)是O(1)的，集合关系检查将花费O(1)，因为我们几乎不需要检查Array[i(n)]是否包含任何数据。Insertion is O(1), as we just ste Array[i(n)].查找和删除最佳操作是O(numnodes)，因为我们必须搜索整个结构。调整操作是O(1)。

3.3.6 哈希表

索引数组使用了很多内存用于保存不在OPEN集中的所有结点。一个选择是使用哈希表。哈希表使用了一个哈希函数h(n)把地图上每个结点映射到一个哈希码。让哈希表的大小等于N的两倍，以使发生冲突的可能性降低。假设h(n) 是O(1)的，集体关系检查操作花费O(1)；插入操作花费O(1)；删除最佳元素操作花费O(numnodes)，因为我们需要搜索整个结构。调整操作花费O(1)。

3.3.7 二元堆

一个二元堆（不要和内存堆混淆）是一种保存在数组中的树结构。和许多普通的树通过指针指向子结点所不同，二元堆使用索引来查找子结点。C++ STL包含了一个二元堆的高效实现，我在我自己的A*代码中使用了它。

在二元堆中，集体关系检查花费O(F)，因为你必须扫描整个结构。插入操作花费O(log F)而删除最佳操作花费也是O(log F)。调整操作很微妙（tricky），花费O(F)时间找到节点，并且很神奇，只用O(log F)来调整。

我的一个朋友（他研究用于最短路径算法的数据结构）说，除非在你的fringe集里有多于10000个元素，否则二元堆是很不错的。除非你的游戏地图特别大，否则你不需要更复杂的数据结构（如multi-level buckets（http://www-cs-students.stanford.edu/~csilvers/））。你应该尽可能不用Fibonacci 堆（http://www.star-lab.com/goldberg/pub/neci-tr-96-062.ps），因为虽然它的渐近复杂度很好，但是执行起来很慢，除非F足够大。

3.3.8 伸展树

堆是一种基于树的结构，它有一个期望的O(log F)代价的时间操作。然而，问题是在A*算法中，通常的情况是，一个代价小的节点被移除（花费O(log F)的代价，因为其他结点必须从树的底部向上移动），而紧接着一些代价小的节点被添加（花费O(log F)的代价，因为这些结点被添加到底部并且被移动到最顶部）。在这里，堆的操作在预期的情况下和最坏情况下是一样的。如果我们找到这样一种数据结构，最坏情况还是一样，而预期的情况好一些，那么就可以得到改进。

伸展树（Splay tree）是一种自调整的树结构。任何对树结点的访问都尝试把该结点推到树的顶部（top）。这就产生了一个缓存效果（"caching" effect）：很少被使用的结点跑到底部（bottom）去了并且不减慢操作（don't slow down operations）。你的splay树有多大并不重要，因为你的操作仅仅和你的“cache size”一样慢。在A*中，低代价的结点使用得很多，而高代价结点经常不被使用，所以高代价结点将会移动到树的底部。

使用伸展树后，集体关系检查，插入，删除最佳和调整操作都是期望的O(log F)（注：原文为expected O(log F)），最坏情况是O(F)。然而有代表性的是，缓存过程（caching）避免了最坏情况的发生。Dijkstra算法和带有低估的启发函数（underestimating heuristic）的A*算法却有一些特性让伸展树达不到最优。特别是对结点n和邻居结点n’来说，f(n') >= f(n)。当这发生时，也许插入操作总是发生在树的同一边结果是使它失去了平衡。我没有试验过这个。

3.3.9 HOT队列

还有一种比堆好的数据结构。通常你可以限制优先队列中值的范围。给定一个限定的范围，经常会存在更好的算法。例如，对任意值的排序可以在O(N log N)时间内完成，但当固定范围时，桶排序和基数排序可以在O(N)时间内完成。

我们可以使用HOT（Heap On Top）队列(http://www.star-lab.com/goldberg/pub /neci-tr-97-104.ps）来利用f(n') >= f(n)，其中n’是n的一个邻居结点。我们删除f(n)值最小的结点n，插入满足f(n) <= f(n') <= f(n) + delta的邻居n'，其中delta <= C。常数C是从一结点到邻近结点代价改变量的最大值。因为f(n)是OPEN集中的最小f值，并且正要被插入的所有结点都小于或等于f(n) + delta，我们知道OPEN集中的所有f值都不超过一个0..delta的范围。在桶/基数排序中，我们可以用“桶”（buckets）对OPEN集中的结点进行排序。

使用K个桶，我们把O(N)的代价降低到平均O(N/K)。通过HOT队列，顶端的桶使用二元堆而所有其他的桶都是未排序数组。因而，对顶部的桶，集合关系检查代价是预期的O(F/K)，插入和删除最佳是O(log (F/K))。对其他桶，集合关系检查是O(F/K)，插入是O(1)，而删除最佳根本不发生！如果顶端的桶是空的，那么我们必须把下一个桶即未排序数组转换为二元堆。这个操作（“heapify”）可以在O(F/K)时间内完成。在调整操作中，删除是O(F/K)，然后插入是O(log (F/K))或O(1)。

在A*中，我们加入OPEN集中的许多结点实际上根本是不需要的。在这方面HOT队列很有优势，因为不需要的元素的插入操作只花费O(1)时间。只有需要的元素被heapified（代价较低的那些）。唯一一个超过O(1)的操作是从堆中删除结点，只花费O(log (F/K))。

另外，如果C比较小，我们可以只让K = C，则对于最小的桶，我们甚至不需要一个堆，国为在一个桶中的所有结点都有相同的f值。插入和删除最佳都是O(1)时间！有人研究过，HOT队列在至多在OPEN集中有800个结点时和堆一样快，并且如果OPEN集中至多有1500个结点，则比堆快20%。我期望随着结点的增加，HOT队列也更快。

HOT队列的一个简单的变化是一个二层队列（two-level queue）：把好的结点放进一个数据结构（堆或数组）而把坏的结点放进另一个数据结构（数组或链表）。因为大多数进入OPEN集中的结点都“坏的”，它们从不被检查，因而把它们放进出一个大数组是没有害处的。

3.3.10 比较

注意有一点很重要，我们并不是仅仅关心渐近的行为（大O符号）。我们也需要关心小常数（low constant）下的行为。为了说明原因，考虑一个O(log F)的算法，和另一个O(F)的算法，其中F是堆中元素的个数。也许在你的机器上，第一个算法的实现花费10000*log(F)秒，而另一个的实现花费2*F秒。当F=256时，第一个算法将花费80000秒而第二个算法花费512秒。在这种情况下，“更快”的算法花费更多的时间，而且只有在当F>200000时才能运行得更快。

你不能仅仅比较两个算法。你还要比较算法的实现。同时你还需要知道你的数据的大小（size）。在上面的例子中，第一种实现在F>200000时更快，但如果在你的游戏中，F小于30000，那么第二种实现好一些。

基本数据结构没有一种是完全合适的。未排序数组或者链表使插入操作很快而集体关系检查和删除操作非常慢。排序数组或者链表使集体关系检查稍微快一些，删除（最佳元素）操作非常快而插入操作非常慢。二元堆让插入和删除操作稍微快一些，而集体关系检查则很慢。伸展树让所有操作都快一些。HOT队 列让插入操作很快，删除操作相当快，而集体关系检查操作稍微快一些。索引数组让集体关系检查和插入操作非常快，但是删除操作不可置信地慢，同时还需要花费 很多内存空间。哈希表和索引数组类似，但在普通情况下，它花费的内存空间少得多，而删除操作虽然还是很慢，但比索引数组要快。

关于更高级的优先队列的资料和实现，请参考Lee Killough的优先队列页面（http://members.xoom.com/killough/heaps.html）。

3.3.11 混合实现

为了得到最佳性能，你将希望使用混合数据结构。在我的A*代码中，我使用一个索引数组从而集合关系检查是O(1)的，一个二元堆从而插入操作和删除最佳都是O(log F)的。对于调整操作，我使用索引数组从而花费O(1)时间检查我是否真的需要进行调整（通过在索引数组中保存g值），然后在少数确实需要进行调整的情况中，我使用二元堆从而调整操作花费O(F)时间。你也可以使用索引数组保存堆中每个结点的位置，这让你的调整操作变成O(log F)。

3.4 与游戏循环的交互

交互式的（尤其是实时的）游戏对最佳路径的计算要求很高。能够得到一个解决方案比得到最佳方案可能更重要。然而在所有其他因素都相同的情况下，短路径比长路径好。

一般来说，计算靠近初始结点的路径比靠近目标结点的路径更重要一些。立即开始原理（The principle of *immediate start*）：让游戏中的物体尽可能快地开始行动，哪怕是沿着一条不理想的路径，然后再计算一条更好的路径。在实时游戏中，应该更多地关注A*的延迟情况（*latency*）而不是吞吐量（*throughput*）。

可以对物体编程让它们根据自己的本能（简单行为）或者智力（一条预先计算好的路径）来行动。除非它们的智力告诉它们怎么行动，否则它们就根据自己的本能来行动（这是实际上使用的方法，并且Rodney Brook在他的机器人体系结构中也用到）。和立即计算所有路径所不同，让游戏在每一个，两个，或者三个循环中搜索一条路径。让物体在开始时依照本能行动(可能仅仅是简单地朝着目标直线前进)，然后才为它们寻找路径。这种方法让让路径搜索的代价趋于平缓，因此它不会集中发生在同一时刻。

3.4.1 提前退出

可以从A*算法的主循环中提前退出来同时得到一条局部路径。通常，当找到目标结点时，主循环就退出了。然而，在此之前的任意结点，可以得到一条到达OPEN中当前最佳结点的路径。这个结点是到达目标点的最佳选择，所以它是一个理想的中间结点（原文为so it's a reasonable place to go）。

可以提前退出的情况包括检查了一定数量的结点，A*算法已经运行了几毫秒时间，或者扫描了一个离初始点有些距离的结点。当使用路径拼接时，应该给被拼接的路径一个比全路径（full path）小的最大长度。

3.4.2 中断算法

如果需要进行路径搜索的物体较少，或者如果用于保存OPEN和CLOSED集的数据结构较小，那么保存算法的状态是可行的，然后退出到游戏循环继续运行游戏。

3.4.3 组运动

路径请求并不是均匀分布的。即时策略游戏中有一个常见的情况，玩家会选择多个物体并命令它们朝着同样的目标移动。这给路径搜索系统以沉重的负载。

在这种情况下，为某个物体寻找到的路径对其它物体也是同样有用的。一种方法是，寻找一条从物体的中心到目的地中心的路径P。对所有物体使用该路径的绝大部分，对每一个物体，前十步和后十步使用为它自己寻找的路径。物体i得到一条从它的开始点到P[10]的路径，紧接着是共享的路径P[10..len(P)-10]，最后是从P[len(P)-10]到目的地的路径。

为每个物体寻找的路径是较短的（平均步数大约是10），而较长的路径被共享。大多数路径只寻找一次并且为所有物体所共享。然而，当玩家们看到所有的物体都沿着相同的路径移动时，将对游戏失去兴趣。为了对系统做些改进，可以让物体稍微沿着不同的路径运动。一种方法是选择邻近结点以改变路径。

另一种方法是让每个物体都意识到其它物体的存在（或许是通过随机选择一个“领导”物体，或者是通过选择一个能够最好地意识到当前情况的物体），同时仅仅为领导寻路。然后用flocking算法让它们以组的形式运动。

然而还有一种方法是利用A*算法的中间状态。这个状态可以被朝着相同目标移动的多个物体共享，只要物体共享相同的启发式函数和代价函数。当主循环退出时，不要消除OPEN和CLOSED集；用A*上一次的OPEN和CLOSED集开始下一次的循环（下一个物体的开始位置）。（这可以被看成是中断算法和提前退出部分的一般化）

3.4.4 细化

如果地图中没有障碍物，而有不同代价的地形，那么可以通过低估地形的代价来计算一条初始路径。例如，如果草地的代价是1，山地代价是2，山脉的代价是3，那么A*会考虑通过3个草地以避免1个山脉。通过把草地看成1，山地看成1.1，而山脉看成1.2来计算初始路径，A*将会用更少的时间去设法避免山脉，而且可以更快地找到一条路径（这接近于精确启发函数的效果）。一旦找到一条路径，物体就可以开始移动，游戏循环就可以继续了。当多余的CPU时间是可用的时候，可以用真实的移动代价去计算更好的路径。

4 A*算法的变种

4.1 beam search

在A*的主循环中，OPEN集保存所有需要检查的结点。Beam Search是A*算法的一个变种，这种算法限定了OPEN集的尺寸。如果OPEN集变得过大，那些没有机会通向一条好的路径的结点将被抛弃。缺点是你必须让排序你的集合以实现这个，这限制了可供选择的数据结构。

4.2 迭代深化

迭代深化是一种在许多AI算法中使用的方法，这种方法从一个近似解开始，逐渐得到更精确的解。该名称来源于游戏树搜索，需要查看前面几步（比如在象棋里），通过查看前面更多步来提高树的深度。一旦你的解不再有更多的改变或者改善，就可以认为你已经得到足够好的解，当你想要进一步精确化时，它不会再有改善。在ID-A*中，深度是f值的一个cutoff。当f的值太大时，结点甚至将不被考虑（例如，它不会被加入OPEN集中）。第一次迭代只处理很少的结点。此后每一次迭代，访问的结点都将增加。如果你发现路径有所改善，那么就继续增加cutoff，否则就可以停止了。更多的细节请参考这些关于ID-A*的资料：
http://www.apl.jhu.edu/~hall/AI-Programming/IDA-Star.html。

我本人认为在游戏地图中没有太大的必要使用ID-A*寻路。ID算法趋向于增加计算时间而减少内存需求。然而在地图路径搜索中，“结点”是很小的——它们仅仅是坐标而已。我认为不保存这些结点以节省空间并不会带来多大改进。

4.3 动态衡量

在动态衡量中，你假设在开始搜索时，最重要的是讯速移动到任意位置；而在搜索接近结束时，最重要的是移动到目标点。

f(p) = g(p) + w(p) * h(p)

启发函数中带有一个权值（weight）（w>=1）。当你接近目标时，你降低这个权值；这降低了启发函数的重要性，同时增加了路径真实代价的相对重要性。

4.4 带宽搜索

带宽搜索（*Bandwidth Search*）有两个对有些人也许有用的特性。这个变种假设h是过高估计的值，但不高于某个数e。如果这就是你遇到的情况，那么你得到的路径的代价将不会比最佳路径的代价超过e。重申一次，你的启发函数设计的越好，最终效果就越好。

另一个特性是，你可以丢弃OPEN集中的某些结点。当h+d比路径的真实代价高的时候（对于某些d），你可以丢弃那些f值比OPEN集中的最好结点的f值高至少e+d的结点。这是一个奇怪的特性。对于好的f值你有一个“范围”（"band"），任何在这个范围之外的结点都可以被丢弃掉，因为这个结点肯定不会在最佳路径上。

好奇地（Curiously），你可以对这两种特性使用不同的启发函数，而问题仍然可以得到解决。使用一个启发函数以保证你得到的路径不会太差，另一个用于检查从OPEN集中去掉哪些结点。

4.5 双向搜索

与从开始点向目标点搜索不同的是，你也可以并行地进行两个搜索——一个从开始点向目标点，另一个从目标点向开始点。当它们相遇时，你将得到一条好的路径。

这听起来是个好主意，但我不会给你讲很多内容。双向搜索的思想是，搜索过程生成了一棵在地图上散开的树。一棵大树比两棵小树差得多，所以最好是使用两棵较小的搜索树。然而我的试验表明，在A*中你得不到一棵树，而只是在搜索地图中当前位置附近的区域，但是又不像Dijkstra算法那样散开。事实上，这就是让A*算法运行得如此快的原因——无论你的路径有多长，它并不进行疯狂的搜索，除非路径是疯狂的。它只尝试搜索地图上小范围的区域。如果你的地图很复杂，双向搜索会更有用。

面对面的方法（The *front-to-front* variation）把这两种搜索结合在一起。这种算法选择一对具有最好的g(start,x) + h(x,y) + g(y,goal)的结点，而不是选择最好的前向搜索结点——g(start,x) + h(x,goal)，或者最好的后向搜索结点——g(y,goal) + h(start,y)。

Retargeting方法不允许前向和后向搜索同时发生。它朝着某个最佳的中间结点运行前向搜索一段时间，然后再朝这个结点运行后向搜索。然后选择一个后向最佳中间结点，从前向最佳中间结点向后向最佳中间结点搜索。一直进行这个过程，直到两个中间结点碰到一块。

4.6 动态A*与终身计划A*

有一些A*的变种允许当初始路径计算出来之后，世界发生改变。D*用于当你没有全局所有信息的时候。如果你没有所有的信息，A*可能会出错；D*的贡献在于，它能纠正那些错误而不用过多的时间。LPA*用于代价会改变的情况。在A*中，当地图发生改变时，路径将变得无效；LPA*可以重新使用之前A*的计算结果并产生新的路径。然而，D*和LPA*都需要很多内存——用于运行A*并保存它的内部信息（OPEN和CLOSED集，路径树，g值），当地图发生改变时，D*或者LPA*会告诉你，是否需要就地图的改变对路径作调整。在一个有许多运动着的物体的游戏中，你经常不希望保存所有这些信息，所以D*和LPA*在这里并不适用。它们是为机器人技术而设计的，这种情况下只有一个机器人——你不需要为别的机器人寻路而重用内存。如果你的游戏只有一个或者少数几个物体，你可以研究一下D*或者LPA*。

- Overview of D*（http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html）
- D* Paper 1（http:// http://www.frc.ri.cmu.edu/~axs/doc/icra94.ps）
- D* Paper 2（http:// http://www.frc.ri.cmu.edu/~axs/doc/ijcai95.ps）
- Lifelong planning overview（http://idm-lab.org/project-a.html）
- Lifelong planning paper (PDF)（http://csci.mrs.umn.edu/UMMCsciwiki/pub/Csci3903s03/KellysPaper/seminar.pdf）
- Lifelong planning A* applet（http://idm-lab.org/applet.html）

5 处理运动障碍物

一个路径搜索算法沿着固定障碍物计算路径，但是当障碍物会运动时情况又怎样？当一个物体到达一个特写的位置，原来的障碍物也许不再在那儿了，或者一个新的障碍物也许到达那儿。处理该问题的一个方法是放弃路径搜索而使用运动算法（movement algorithms）替代，这就不能look far ahead；这种方法会在后面的部分中讨论。这一部分将对路径搜索方法进行修改从而解决运动障碍物的问题。

5.1 重新计算路径

当时间渐渐过去，我们希望游戏世界有所改变。以前搜索到的一条路径到现在也许不再是最佳的了。对旧的路径用新的信息进行更新是有价值的。以下规则可以用于决定什么时候需要重新计算路径：

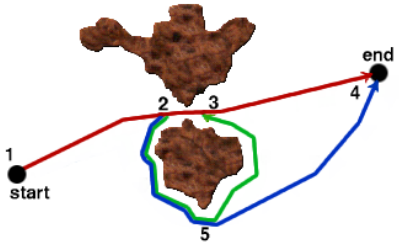
- 每N步：这保证用于计算路径的信息不会旧于N步。
- 任何可以使用额外的CPU时间的时候：这允许动态调整路径的性质；在物体数量多时，或者运行游戏的机器比较慢时，每个物体对CPU的使用可得到减少。
- 当物体拐弯或者跨越一个导航点（waypoint）的时候。
- 当物体附近的世界改变了的时候。

重计算路径的主要缺点是许多路径信息被丢弃了。例如，如果路径是100步长，每10步重新计算一次，路径的总步数将是100+90+80+70+60+50+40+30+20+10 = 550。对M步长的路径，大约需要计算M^2步。因此如果你希望有许多很长的路径，重计算不是个好主意。重新使用路径信息比丢弃它更好。

5.2 路径拼接

当 一条路径需要被重新计算时，意味着世界正在改变。对于一个正在改变的世界，对地图中当前邻近的区域总是比对远处的区域了解得更多。因此，我们应该集中于在 附近寻找好的路径，同时假设远处的路径不需要重新计算，除非我们接近它。与重新计算整个路径不同，我们可以重新计算路径的前M步：

1. 令p[1]..p[N]为路径（N步）的剩余部分
2. 为p[1]到p[M]计算一条新的路径
3. 把这条新路径拼接（Splice）到旧路径：把p[1]..p[M]用新的路径值代替



4.

因为p[1]和p[M]比分开的M步小（原文：Since p[1] and p[M] are fewer than M steps apart），看起来新路径不会很长。不幸的是，新的路径也许很长而且不够好。上面的图显示了这种情况。最初的红色路径是1-2-3-4，褐色的是障碍物。如果我们到达2并且发现从2到达3的路径被封锁了，路径拼接技术会把2-3用2-5-3取代，结果是物体沿着路径1-2-5-3-4运动。我们可以看到这不是一条好的路径，蓝色的路径1-2-5-4是一条更好的路径。

通常可以通过查看新路径的长度检测到坏的路径。如果这严格大于M，就可能是不好的。一个简单的解决方法是，为搜索算法设置一个最大路径长度。如果找不到一条短的路径，算法返回错误代码；这种情况下，用重计算路径取代路径拼接，从而得到路径1-2-5-4.。

对于其它情况，对于N步的路径，路径拼接会计算2N或者3N步，这取决于拼接新路径的频率。对于对世界的改变作反应的能力而言，这个代价是相当低的。令人吃惊的是这个代价和拼接的步数M无关。M不影响CPU时间，而控制了响应和路径质量的折衷。如果M太大，物体的移动将不能快速对地图的改变作出反应。如果M太小，拼接的路径可能太短以致不能正确地绕过障碍物；许多不理想的路径（如1-2-5-3-4）将被找到。尝试不同的M值和不同的拼接标准(如每3/4 *M*步)，看看哪一种情况对你的地图最合适。

路径拼接确实比重计算路径要快，但它不能对路径的改变作出很好的反应。经常可以发现这种情况并用路径重计算来取代。也可以调整一些变量，如M和寻找新路径的时机，所以可以对该方法进行调整(甚至在运行时)以用于不同的情况。

Note：Bryan Stout 有两个算法，Patch-One和Patch-All，他从路径拼接中得到灵感，并在实践中运行得很好。他出席了GDC 2007（https://www.cmpevents.com/GD07/a.asp?option =C &V=11& SessID=4608）；一旦他把资料放在网上，我将链接过去。

Implementation Note:

反向保存路径，从而删除路径的开始部分并用不同长度的新路径拼接将更容易，因为这两个操作都将在数组的末尾进行。本质上你可以把这个数组看成是堆栈因为顶部的元素总是下一个要使用的。

5.3 监视地图变化

与 间隔一段时间重计算全部或部分路径不同的是，可以让地图的改变触发一次重计算。地图可以分成区域，每个物体都可以对某些区域感兴趣（可以是包含部分路径的 所有区域，也可以只是包含部分路径的邻近区域）。当一个障碍物进入或者离开一个区域，该区域将被标识为已改变，所有对该区域感兴趣的物体都被通知到，所以 路径将被重新计算以适应障碍物的改变。

这种技术有许多变种。例如，可以每隔一定时间通知物体，而不是立即通知物体。多个改变可以成组地触发一个通知，因此避免了额外的重计算。另一个例子是，让物体检查区域，而不是让区域通知物体。

监视地图变化允许当障碍物不改变时物体避免重计算路径，所以当你有许多区域并不经常改变时，考虑这种方法。

5.4 预测障碍物的运动

如果障碍物的运动可以预测，就能为路径搜索考虑障碍物的未来位置。一个诸如A*的算法有一个代价函数用以检查穿过地图上一点的代价有多难。A*可以被改进从而知道到达一点的时间需求（通过当前路径长度来检查），而现在则轮到代价函数了。代价函数可以考虑时间，并用预测的障碍物位置检查在某个时刻地图某个位置是否可以通过。这个改进不是完美的，然而，因为它并不考虑在某个点等待障碍物自动离开的可能性，同时A*并不区分到达相同目的地的不同的路径，而是针对不同的目的地，所以还是可以接受的。

6 预计算路径的空间代价

有时，路径计算的限制因素不是时间，而是用于数以百计的物体的存储空间。路径搜索器需要空间以运行算法和保存路径。算法运行所需的临时空间（在A*中是OPEN和CLOSED集）通常比保存结果路径的空间大许多。通过限制在一定的时间计算一条路径，可以把临时空间数量最小化。另外，为OPEN和CLOSED集所选择的数据结构的不同，最小化临时空间的程度也有很大的不同。这一部分聚集于优化用于计算路径的空间代价。

6.1 位置VS方向

一条路径可以用位置或者方向来表示。位置需要更多的空间，但是有一个优点，易于查询路径中的任意位置或者方向而不用沿着路径移动。当保存方向时，只有方向容易被查询；只有沿着整个路径移动才能查询位置。在一个典形的网格地图中，位置可以被保存为两个16位整数，每走一步是32位。而方向是很少的，因此用极少的空间就够了。如果物体只能沿着四个方向移动，每一步用两位就够了；如果物体能沿着6个或者8个方向移动，每一步也只需要三位。这些对于保存路径中的位置都有明显的空间节省。Hannu Kankaanpaa指出可以进一步减少空间需求，那就是保存相对方向（右旋60度）而不是绝对方向（朝北走）。有些相对方向对某些物体来说意义不大。比如，如果你的物体朝北移动，那么下一步朝南移动的可能性很小。在只有六种方向的游戏中，你只有五个有意义的方向。在某些地图中，也许只有三个方向（直走，左旋60度，右旋60度）有意义，而其它地图中，右旋120度是有效的（比如，沿着陡峭的山坡走之字形的路径时）。

6.2 路径压缩

一旦找到一条路径，可以对它进行压缩。可以用一个普通的压缩算法，但这里不进行讨论。使用特定的压缩算法可以缩小路径的存储，无论它是基于位置的还是基于方向的。在做决定之前，考察你的游戏中的路径以确定哪种压缩效果最好。另外还要考虑实现和调试，代码量，and whether it really matters.如果你有300个物体并且在同一时刻只有50个在移动，同时路径比较短（100步），内存总需求大概只有不到50k，总之，没有必要担心压缩的效果。

6.2.1 位置存储

在障碍物比地形对路径搜索影响更大的地图中，路径中有大部分是直线的。如果是这种情况，那么路径只需要包含直线部分的终止点（有时叫waypoints）。此时移动过程将包含检查下一结点和沿着直线向前移动。

6.2.2 方向存储

保存方向时，有一种情况是同一个方向保存了很多次。可以用简单的方法节省空间。

一种方法是保存方向以及朝着该方向移动的次数。和位置存储的优化不同，当一个方向并不是移动很多次时，这种优化的效果反而不好。同样的，对于那些可以进行位置压缩的直线来说，方向压缩是行不通的，因为这条直线可能没有和正在移动的方向关联。通过相对方向，你可以把“继续前进”当作可能的方向排除掉。Hannu Kankaanpaa指出，在一个八方向地图中，你可以去掉前，后，以及向左和向右135度（假设你的地图允许这个），然后你可以仅用两个比特保存每个方向。

另一种保存路径的方法是变长编码。这种想法是使用一个简单的比特（0）保存最一般的步骤：向前走。使用一个“1”表示拐弯，后边再跟几个比特表示拐弯的方向。在一个四方向地图中，你只能左转和右转，因此可以用“10”表示左转，“11”表示右转。

变长编码比run length encoding更一般，并且可以压缩得更好，但对于较长的直线路径则不然。序列（向北直走6步，左转，直走3步，右转，直走5步，左转，直走2步）用run length encoding表示是[(NORTH, 6), (WEST, 3), (NORTH, 5), (WEST, 2)]。如果每个方向用2比特，每个距离用8比特，保存这条路径需要40比特。而对于变长编码，你用1比特表示每一步，2比特表示拐弯——[NORTH 0 0 0 0 0 0 10 0 0 0 11 0 0 0 0 0 10 0 0]——一共24比特。如果初始方向和每次拐弯对应1步，则每次拐弯都节省了一个比特，结果只需要20比特保存这条路径。然而，用变长编码保存更长的路径时需要更多的空间。序列(向北直走200步)用run length encoding表示是[(NORTH, 200)]，总共需要10比特。用变长编码表示同样的序列则是[NORTH 0 0 ...]，一共需要202比特。

6.3 计算导航点

一个导航点（waypoint）是路径上的一个结点。与保存路径上的每一步不同，在进行路径搜索之后，一个后处理（post-processing）的步骤可能会把若干步collapse（译者：不好翻译，保留原单词）为一个简单的导航点，这经常发生在路径上那些方向发生改变的地方，或者在一个重要的（major）位置如城市。然后运动算法将在两个导航点之间运行。

6.4 极限路径长度

当地图中的条件或者秩序会发生改变时，保存一条长路径是没有意义的，因为在从某些点开始，后边的路径已经没有用了。每个物体都可以保存路径开始时的特定几步，然后当路径已经没用时重新计算路径。这种方法虑及了（allows for）对每个物体使用数据的总量的管理。

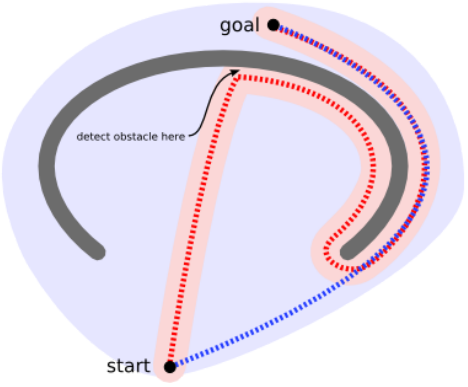
6.5 总结

在游戏中，路径潜在地花费了许多存储空间，特别是当路径很长并且有很多物体需要寻路时。路径压缩，导航点和beacons通过把多个步骤保存为一个较小数据从而减少了空间需求。Waypoints rely on straight-line segments being common so that we have to store only the endpoints, while beacons rely on there being well-known paths calculated beforehand between specially marked places on the map.（译者：此处不好翻译，暂时保留原文）如果路径仍然用了许多存储空间，可以限制路径长度，这就回到了经典的时间-空间折衷法：为了节省空间，信息可以被丢弃，稍后才重新计算它。

原文地址：*http://theory.stanford.edu/~amitp/GameProgramming/*

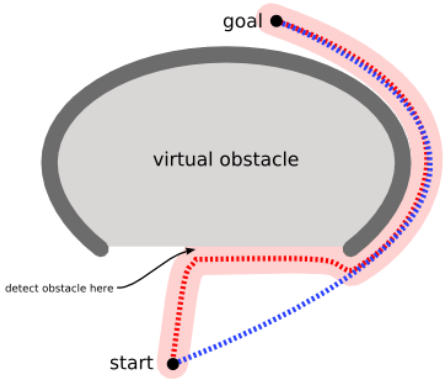
1 导言

移动一个简单的物体（*object*）看起来是容易的。而路径搜索是复杂的。为什么涉及到路径搜索就产生麻烦了？考虑以下情况：



物体（*unit*）最初位于地图的底端并且尝试向顶部移动。物体扫描的区域中*(*粉红色部分*)*没有任何东西显示它不能向上移动，因此它持续向上移动。在靠近顶部时，它探测到一个障碍物然后改变移动方向。然后它沿着*U*形障碍物找到它的红色的路径。相反的，一个路径搜索器（*pathfinder*）将会扫描一个更大的区域（淡蓝色部分），但是它能做到不让物体*(unit)*走向凹形障碍物而找到一条更短的路径*(*蓝色路径）。

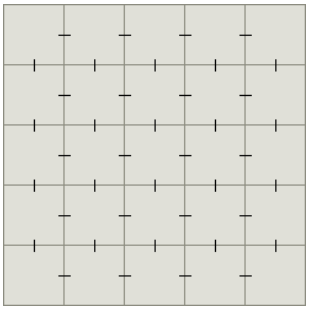
然而你可以扩展一个运动算法，用于对付上图所示的障碍物。或者避免制造凹形障碍，或者把凹形出口标识为危险的*(*只有当目的地在里面时才进去*)*:



比起一直等到最后一刻才发现问题，路径搜索器让你提前作出计划。不带路径搜索的运动*(movement)*可以在很多种情形下工作，同时可以扩展到更多的情形，但是路径搜索是一种更常用的解决更多问题的方法。

1.1 算法

计算机科学教材中的路径搜索算法在数学视角的图上工作——由边联结起来的结点的集合。一个基于图块*(tile)*拼接的游戏地图可以看成是一个图，每个图块*(tile)*是一个结点，并在每个图块之间画一条边：

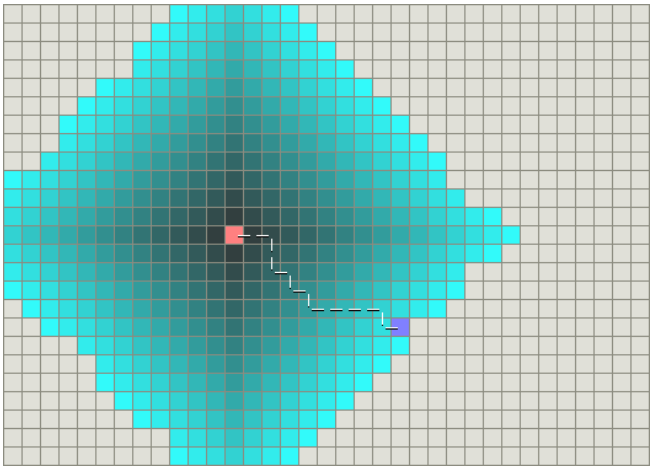


目前，我会假设我们使用二维网格*(grid)*。稍后我将讨论如何在你的游戏之外建立其他类型的图。

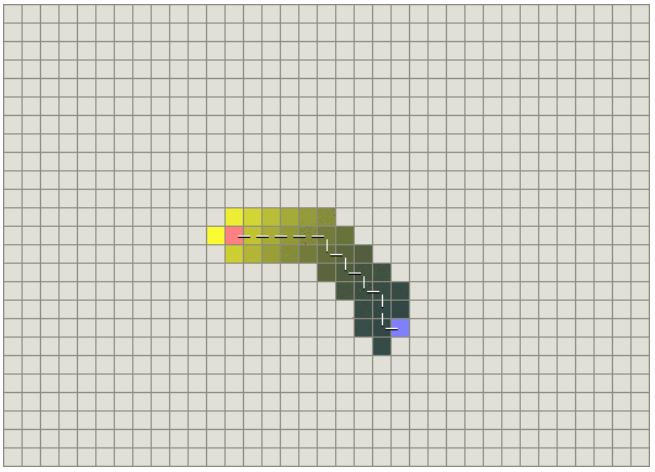
许多*AI*领域或算法研究领域中的路径搜索算法是基于任意*(arbitrary)*的图设计的，而不是基于网格*(grid-based)*的 图。我们可以找到一些能使用网格地图的特性的东西。有一些我们认为是常识，而算法并不理解。例如，我们知道一些和方向有关的东西：一般而言，如果两个物体 距离越远，那么把其中一个物体向另一个移动将花越多的时间；并且我们知道地图中没有任何秘密通道可以从一个地点通向另一个地点。（我假设没有，如果有的 话，将会很难找到一条好的路径，因为你并不知道要从何处开始。）

1.2 *Dijkstra*算法与最佳优先搜索

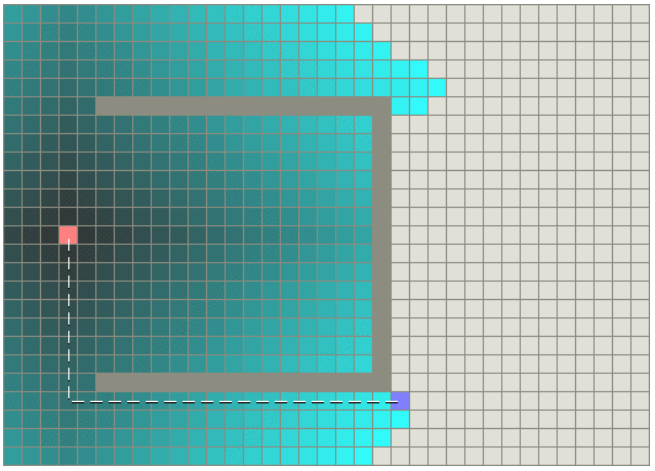
*Dijkstra*算法从物体所在的初始点开始，访问图中的结点。它迭代检查待检查结点集中的结点，并把和该结点最靠近的尚未检查的结点加入待检查结点集。该结点集从初始结点向外扩展，直到到达目标结点。*Dijkstra*算法保证能找到一条从初始点到目标点的最短路径，只要所有的边都有一个非负的代价值。（我说“最短路径”是因为经常会出现许多差不多短的路径。）在下图中，粉红色的结点是初始结点，蓝色的是目标点，而类菱形的有色区域（注：原文是*teal areas*）则是*Dijkstra*算法扫描过的区域。颜色最淡的区域是那些离初始点最远的，因而形成探测过程（*exploration*）的边境（*frontier*）：



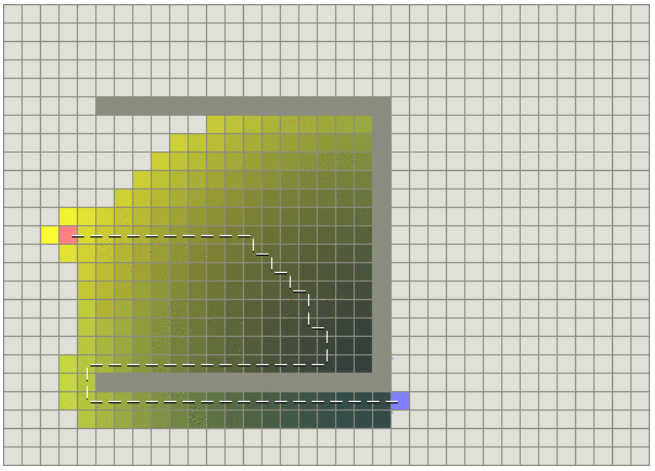
最佳优先搜索（*BFS*）算法按照类似的流程运行，不同的是它能够评估（称为启发式的）任意结点到目标点的代价。与选择离初始结点最近的结点不同的是，它选择离目标最近的结点。*BFS*不能保证找到一条最短路径。然而，它比*Dijkstra*算法快的多，因为它用了一个启发式函数（*heuristic*）快速地导向目标结点。例如，如果目标位于出发点的南方，*BFS*将趋向于导向南方的路径。在下面的图中，越黄的结点代表越高的启发式值（移动到目标的代价高），而越黑的结点代表越低的启发式值（移动到目标的代价低）。这表明了与*Dijkstra*算法相比，*BFS*运行得更快。



然而，这两个例子都仅仅是最简单的情况——地图中没有障碍物，最短路径是直线的。现在我们来考虑前边描述的凹型障碍物。*Dijkstra*算法运行得较慢，但确实能保证找到一条最短路径：



另一方面，*BFS*运行得较快，但是它找到的路径明显不是一条好的路径：



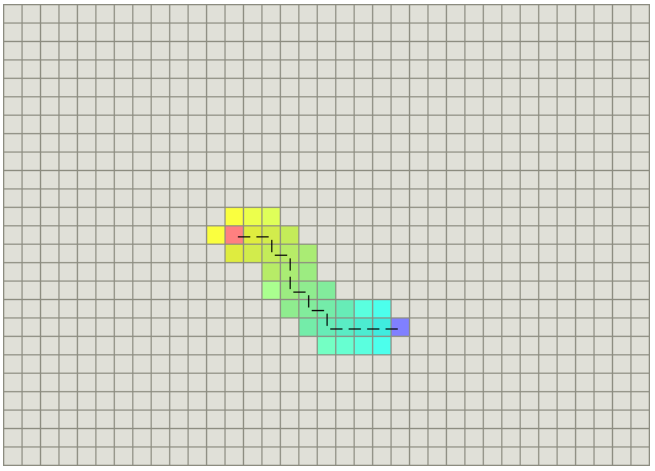
问题在于*BFS*是基于贪心策略的，它试图向目标移动尽管这不是正确的路径。由于它仅仅考虑到达目标的代价，而忽略了当前已花费的代价，于是尽管路径变得很长，它仍然继续走下去。

结合两者的优点不是更好吗？*1968*年发明的*A**算法就是把启发式方法（*heuristic approaches*）如*BFS*，和常规方法如*Dijsktra*算法结合在一起的算法。有点不同的是，类似*BFS*的启发式方法经常给出一个近似解而不是保证最佳解。然而，尽管*A**基于无法保证最佳解的启发式方法，*A**却能保证找到一条最短路径。

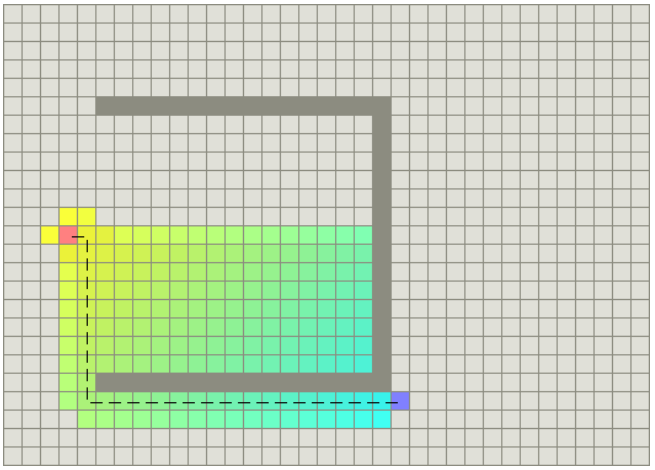
1.3 *A**算法

我将集中讨论*A**算法。*A**是路径搜索中最受欢迎的选择，因为它相当灵活，并且能用于多种多样的情形之中。

和其它的图搜索算法一样，*A**潜在地搜索图中一个很大的区域。和*Dijkstra*一样，*A**能用于搜索最短路径。和*BFS*一样，*A**能用启发式函数（注：原文为*heuristic*）引导它自己。在简单的情况中，它和*BFS*一样快。



在凹型障碍物的例子中，*A**找到一条和*Dijkstra*算法一样好的路径：



成功的**秘诀**在于，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**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*）。在上图中，*yellow(h)*表示远离目标的结点而*teal(g)*表示远离初始点的结点。当从初始点向目标点移动时，*A**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)*。

2 启发式算法

启发式函数*h(n)*告诉*A**从任意结点*n*到目标结点的最小代价评估值。选择一个好的启发式函数是重要的。

2.1 A*对启发式函数的使用

启发式函数可以控制A*的行为：

- 一种极端情况，如果h(n)是0，则只有g(n)起作用，此时A*演变成Dijkstra算法，这保证能找到最短路径。
- 如果h(n)经常都比从n移动到目标的实际代价小（或者相等），则A*保证能找到一条最短路径。h(n)越小，A*扩展的结点越多，运行就得越慢。
- 如果h(n)精确地等于从n移动到目标的代价，则A*将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等（译者：指让h(n)精确地等于实际值）。只要提供完美的信息，A*会运行得很完美，认识这一点很好。
- 如果h(n)有时比从n移动到目标的实际代价高，则A*不能保证找到一条最短路径，但它运行得更快。
- 另一种极端情况，如果h(n)比g(n)大很多，则只有h(n)起作用，A*演变成BFS算法。

所以我们得到一个很有趣的情况，那就是我们可以决定我们想要从A*中获得什么。理想情况下（注：原文为*At exactly the right point*），我们想最快地得到最短路径。如果我们的目标太低，我们仍会得到最短路径，不过速度变慢了；如果我们的目标太高，那我们就放弃了最短路径，但A*运行得更快。

在游戏中，A*的这个特性非常有用。例如，你会发现在某些情况下，你希望得到一条好的路径（*"good" path*）而不是一条完美的路径（*"perfect" path*）。为了权衡g(n)和h(n)，你可以修改任意一个。

注:在学术上，如果启发式函数值是对实际代价的低估，A*算法被称为简单的A算法（原文为*simply A*）。然而，我继续称之为A*，因为在实现上是一样的，并且在游戏编程领域并不区别A和A*。

2.2 速度还是精确度？

A*改变它自己行为的能力基于启发式代价函数，启发式函数在游戏中非常有用。在速度和精确度之间取得折衷将会让你的游戏运行得更快。在很多游戏中，你并不真正需要得到最好的路径，仅需要近似的就足够了。而你需要什么则取决于游戏中发生着什么，或者运行游戏的机器有多快。

假设你的游戏有两种地形，平原和山地，在平原中的移动代价是1而在山地则是3。*A* is going to search three times as far along flat land as it does along mountainous land.* 这是因为有可能有一条沿着平原到山地的路径。把两个邻接点之间的评估距离设为1.5可以加速A*的搜索过程。然后A*会将3和1.5比较，这并不比把3和1比较差。*It is not as dissatisfied with mountainous terrain, so it won't spend as much time trying to find a way around it. Alternatively, you can speed up up A*'s search by decreasing the amount it searches for paths around mountains-just tell A* that the movement cost on mountains is 2 instead of 3. Now it will search only twice as far along the flat terrain as along mountainous terrain. Either approach gives up ideal paths to get something quicker.*

速度和精确度之间的选择前不是静态的。你可以基于CPU的速度、用于路径搜索的时间片数、地图上物体（*units*）的数量、物体的重要性、组（*group*）的大小、难度或者其他任何因素来进行动态的选择。取得动态的折衷的一个方法是，建立一个启发式函数用于假定通过一个网格空间的最小代价是1，然后建立一个代价函数（*cost* ）用于测量（*scales*）：

$$g'(n) = 1 + alpha * (g(n) - 1)$$

如果alpha是0，则改进后的代价函数的值总是1。这种情况下，地形代价被完全忽略，A*工作变成简单地判断一个网格可否通过。如果alpha是1，则最初的代价函数将起作用，然后你得到了A*的所有优点。你可以设置alpha的值为0到1的任意值。

你也可以考虑对启发式函数的返回值做选择：绝对最小代价或者期望最小代价。例如，如果你的地图大部分地形是代价为2的草地，其它一些地方是代价为1的道路，那么你可以考虑让启发式函数不考虑道路，而只返回2*距离。

速度和精确度之间的选择并不是 全局的。在地图上的某些区域，精确度是重要的，你可以基于此进行动态选择。例如，假设我们可能在某点停止重新计算路径或者改变方向，则在接近当前位置的地 方，选择一条好的路径则是更重要的，因此为何要对后续路径的精确度感到厌烦？或者，对于在地图上的一个安全区域，最短路径也许并不十分重要，但是当从一个 敌人的村庄逃跑时，安全和速度是最重要的。（译者注：译者认为这里指的是，在安全区域，可以考虑不寻找精确的最短路径而取近似路径，因此寻路快；但在危险 区域，逃跑的安全性和逃跑速度是重要的，即路径的精确度是重要的，因此可以多花点时间用于寻找精确路径。）

2.3 衡量单位

A*计算f(n) = g(n) + h(n)。为了对这两个值进行相加，这两个**值必须**使用相同的衡量单位。如果g(n)用小时来衡量而h(n)用米来衡量，那么A*将会认为g或者h太大或者太小，因而你将不能得到正确的路径，同时你的A*算法将运行得更慢。

2.4 精确的启发式函数

如果你的启发式函数精确地等于实际最佳路径（*optimal path*），如下一部分的图中所示，你会看到此时A*扩展的结点将非常少。A*算法内部发生的事情是：在每一结点它都计算f(n) = g(n) + h(n)。当h(n)精确地和g(n)匹配（译者注：原文为*match*）时，f(n)的值在沿着该路径时将不会改变。不在正确路径（*right path*）上的所有结点的f值均大于正确路径上的f值（译者注：正确路径在这里应该是指最短路径）。如果已经有较低f值的结点，A*将不考虑f值较高的结点，因此它肯定不会偏离最短路径。

2.4.1 预计算的精确启发式函数

构造精确启发函数的一种方法是预先计算任意一对结点之间最短路径的长度。在许多游戏的地图中这并不可行。然后，有几种方法可以近似模拟这种启发函数：

- *Fit a coarse grid on top of the fine grid. Precompute the shortest path between any pair of coarse grid locations.*
- *Precompute the shortest path between any pair of waypoints. This is a generalization of the coarse grid approach.*

（译者：此处不好翻译，暂时保留原文）

然后添加一个启发函数h′用于评估从任意位置到达邻近导航点（*waypoints*）的代价。（如果愿意，后者也可以通过预计算得到。）最终的启发式函数可以是：

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

或者如果你希望一个更好但是更昂贵的启发式函数，则分别用靠近结点和目标的所有的w1,w2对对上式进行求值。（译者注：原文为*or if you want a better but more expensive heuristic, evaluate the above with all pairs w1, w2 that are close to the node and the goal, respectively.*）

2.4.2 线性精确启发式算法

在特殊情况下，你可以不通过预计算而让启发式函数很精确。如果你有一个不存在障碍物和slow地形，那么从初始点到目标的最短路径应该是一条直线。

如果你正使用简单的启发式函数（我们不知道地图上的障碍物），则它应该和精确的启发式函数相符合（译者注：原文为*match*）。如果不是这样，则你会遇到衡量单位的问题，或者你所选择的启发函数类型的问题。

2.5 网格地图中的启发式算法

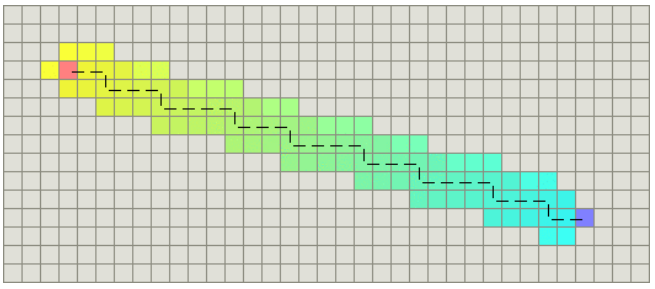
在网格地图中，有一些众所周知的启发式函数。

2.5.1 曼哈顿距离

标准的启发式函数是曼哈顿距离（*Manhattan distance*）。考虑你的代价函数并找到从一个位置移动到邻近位置的最小代价*D*。因此，我的游戏中的启发式函数应该是曼哈顿距离的*D*倍：

```
H(n) = D * (abs ( n.x – goal.x ) + abs ( n.y – goal.y ) )
```

你应该使用符合你的代价函数的衡量单位。



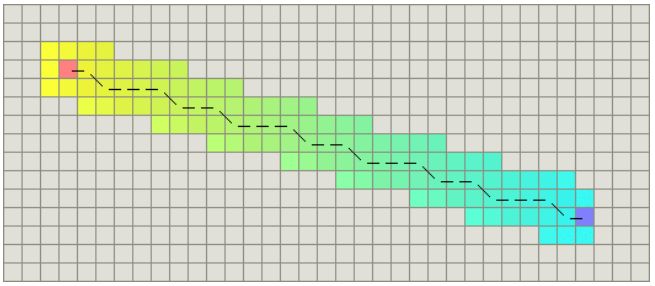
(Note: the above image has a tie-breaker added to the heuristic.)

（译者注：曼哈顿距离——两点在南北方向上的距离加上在东西方向上的距离，即*D*（*I*，*J*）*=|XI-XJ|+|YI-YJ|*。对于一个具有正南正北、正东正西方向规则布局的城镇街道，从一点到达另一点的距离正是在南北方向上旅行的距离加上在东西方向上旅行的距离因此曼哈顿距离又称为出租车距离，曼哈顿距离不是距离不变量，当坐标轴变动时，点间的距离就会不同——百度知道）

2.5.2 对角线距离

如果在你的地图中你允许对角运动那么你需要一个不同的启发函数。（*4 east, 4 north*）的曼哈顿距离将变成*8*D*。然而，你可以简单地移动（*4 northeast*）代替，所以启发函数应该是*4*D*。这个函数使用对角线，假设直线和对角线的代价都是*D*：

```
h(n) = D * max(abs(n.x - goal.x), abs(n.y - goal.y))
```



如果对角线运动的代价不是*D*，但类似于*D2 = sqrt(2) * D*，则上面的启发函数不准确。你需要一些更准确（原文为*sophisticated*）的东西：

```
h_diagonal(n) = min(abs(n.x - goal.x), abs(n.y - goal.y))

h_straight(n) = (abs(n.x - goal.x) + abs(n.y - goal.y))

h(n) = D2 * h_diagonal(n) + D * (h_straight(n) - 2*h_diagonal(n)))
```

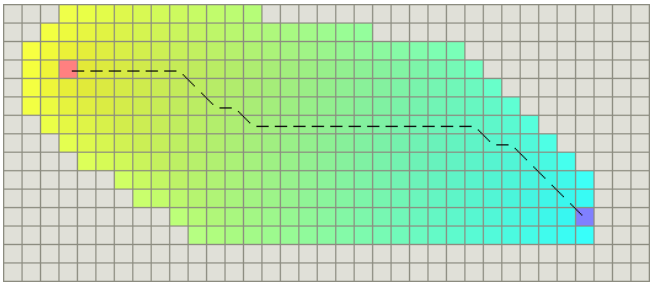
这里，我们计算*h_diagonal(n)*：沿着斜线可以移动的步数；*h_straight(n)*：曼哈顿距离；然后合并这两项，让所有的斜线步都乘以*D2*，剩下的所有直线步*(*注意这里是曼哈顿距离的步数减去*2*倍的斜线步数*)*都乘以*D*。

2.5.3 欧几里得距离

如果你的单位可以沿着任意角度移动（而不是网格方向），那么你也许应该使用直线距离：

```
h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)
```

然而，如果是这样的话，直接使用*A**时将会遇到麻烦，因为代价函数*g*不会*match*启发函数*h*。因为欧几里**得距离**比曼哈顿距离和对角线距离都短，你仍可以得到最短路径，不过*A**将运行得更久一些：

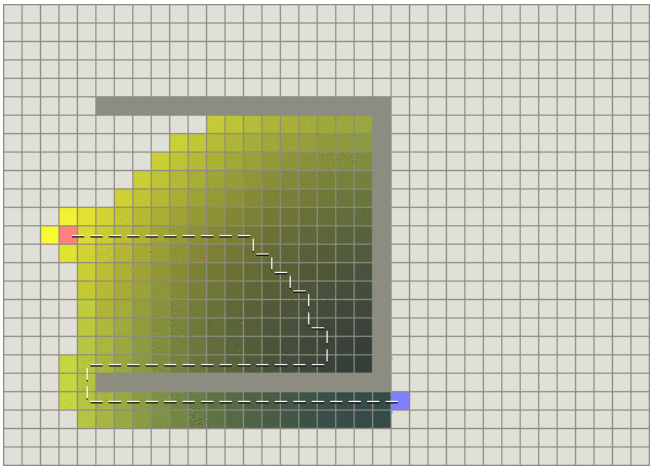


2.5.4 平方后的欧几里得距离

我曾经看到一些*A**的网页，其中提到让你通过使用距离的平方而避免欧几里**得距离**中昂贵的平方根运算：

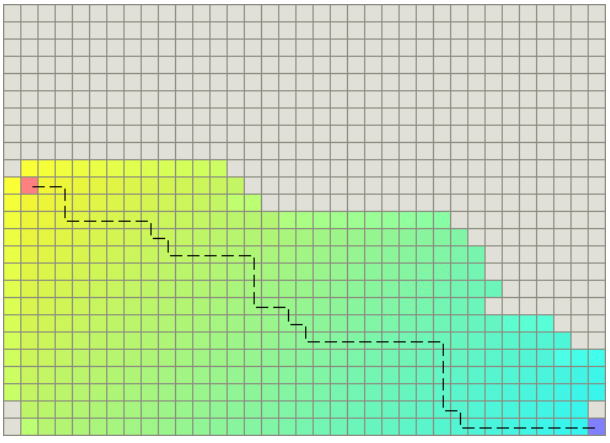
*h(n) = D * ((n.x-goal.x)^2 + (n.y-goal.y)^2)*

不要这样做！这明显地导致衡量单位的问题。当*A**计算*f(n) = g(n) + h(n)*，距离的**平方将**比*g*的代价大很多，并且你会因为启发式函数评估值过高而停止。对于更长的距离，这样做会靠近*g(n)*的极端情况而不再计算任何东西，*A**退化成*BFS*：



2.5.5 Breaking ties

导致低性能的一个原因来自于启发函数的*ties*（注：这个词实在不知道应该翻译为什么）。当某些路径具有相同的*f*值的时候，它们都会被搜索（*explored*），尽管我们只需要搜索其中的一条：



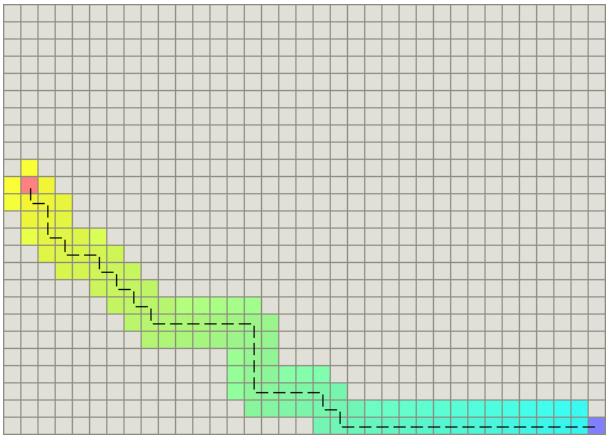
Ties in f values.

为了解决这个问题，我们可以为启发函数添加一个附加值（译者注：原文为*small tie breaker*）。附加值对于结点必须是确定性的（也就是说，不能是随机的数），而且它必须让*f*值体现区别。因为*A**对*f*值排序，让*f*值不同意味着只有一个*"equivalent"*的*f*值会被检测。

一种添加附加值的方式是稍微改变（译者注：原文为*nudge*）*h*的衡量单位。如果我们减少衡量单位（译者注：原文为*scale it downwards*），那么当我们朝着目标移动的时候*f*将逐渐增加。很不幸，这意味着*A**倾向于扩展到靠近初始点的结点，而不是靠近目标的结点。我们可以增加衡量单位（译者注：原文为*scale it downwards scale h upwards slightly*）（甚至是*0.1%*），*A**就会倾向于扩展到靠近目标的结点。

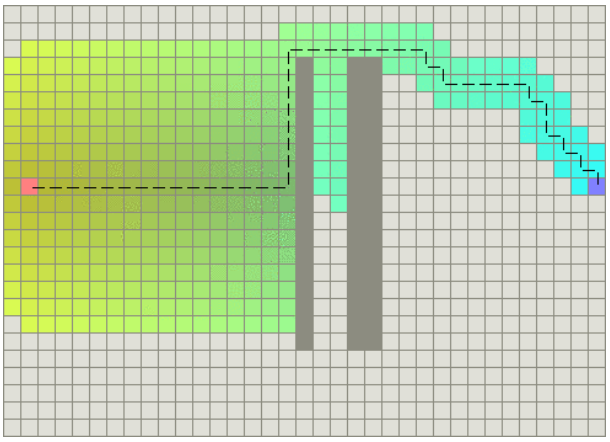
*heuristic *= (1.0 + p)*

选择因子*p*使得*p* < 移动一步（*step*）的最小代价 / 期望的最长路径长度。假设你不希望你的路径超过*1000*步（*step*），你可以使*p = 1 / 1000*。添加这个附加值的结果是，*A**比以前搜索的结点更少了。



Tie-breaking scaling added to heuristic.

当存在障碍物时，当然仍要在它们周围寻找路径，但要意识到，当绕过障碍物以后，*A**搜索的区域非常少：



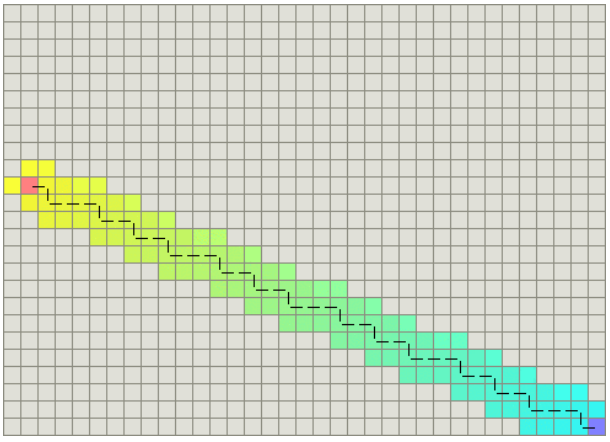
Tie-breaking scaling added to heuristic, works nicely with obstacles.

*Steven van Dijk*建议，一个更直截了当的方法是把*h*传递到比较函数（*comparison*）。当*f*值相等时，比较函数检查*h*，然后添加附加值。

一个不同的添加附加值的方法是，倾向于从初始点到目标点的连线（直线）：

```
dx1 = current.x - goal.x

dy1 = current.y - goal.y

dx2 = start.x - goal.x

dy2 = start.y - goal.y

cross = abs(dx1*dy2 - dx2*dy1)

heuristic += cross*0.001
```

这段代码计算初始-目标向量（*start to goal vector*）和当前-目标向量（*current point to goal vector*）的向量叉积（*vector cross-product*）。*When these vectors don't line up the cross product will be larger.*结果是，这段代码选择的路径稍微倾向于从初始点到目标点的直线。当没有障碍物时，*A**不仅搜索很少的区域，而且它找到的路径看起来**非常**棒：



Tie-breaking cross-product added to heuristic, produces pretty paths.

然而，因为这种附加值倾向于从初始点到目标点的直线路径，当出现障碍物时将会出现奇怪的结果（注意这条路径仍是最佳的，只是看起来很奇怪）：

Tie-breaking cross-product added to heuristic, less pretty with obstacles.

为了交互地研究这种附加值方法的改进，请参考James Macgill的A*确applet（http://www.ccg.leeds.ac.uk/james/aStar/）[如果链接无效，请使用这个镜像（http://www.vision.ee.ethz.ch/~buc/astar/AStar.html）]（译者注：两个链接均无效）。使用“Clear”以清除地图，选择地图对角的两个点。当你使用“Classic A*”方法，你会看到附加值的效果。当你使用“Fudge”方法，你会看到上面给启发函数添加叉积后的效果。

然而另一种添加附加值的方法是，小心地构造你的A*优先队列，使新插入的具有特殊f值的结点总是比那些以前插入的具有相同f值的旧结点要好一些。

你也许也想看看能够更灵活地（译者注：原文为sophisticated）添加附加值的AlphA*算法（http://home1.stofanet.dk/breese/papers.html），不过用这种算法得到的路径是否能达到最佳仍在研究中。AlphA*具有较好的适应性，而且可能比我在上面讨论的附加值方法运行得都要好。然而，我所讨论的附加值方法非常容易实现，所以从它们开始吧，如果你需要得到更好的效果，再去尝试AlphA*。

2.5.6 区域搜索

如果你想搜索邻近目标的任意不确定结点，而不是某个特定的结点，你应该建立一个启发函数h’(x)，使得h’(x)为h1(x), h2(x), h3(x)。。。的最小值，而这些h1, h2, h3是邻近结点的启发函数。然而，一种更快的方法是让A*仅搜索目标区域的中心。一旦你从OPEN集合中取得任意一个邻近目标的结点，你就可以停止搜索并建立一条路径了。

3 Implementation notes

3.1 概略

如果不考虑具体实现代码，A*算法是相当简单的。有两个集合，OPEN集和CLOSED集。其中OPEN集保存待考查的结点。开始时，OPEN集只包含一个元素：初始结点。CLOSED集保存已考查过的结点。开始时，CLOSED集是空的。如果绘成图，OPEN集就是被访问区域的边境（frontier）而CLOSED集则是被访问区域的内部（interior）。每个结点同时保存其父结点的指针因此我们可以知道它是如何被找到的。

在主循环中重复地从OPEN集中取出最好的结点n（f值最小的结点）并检查之。如果n是目标结点，则我们的任务完成了。否则，结点n被从OPEN集中删除并加入CLOSED集。然后检查它的邻居n’。如果邻居n’在CLOSED集中，那么它是已经被检查过的，所以我们不需要考虑它*；如果n’在OPEN集中，那么它是以后肯定会被检查的，所以我们现在不考虑它*。否则，把它加入OPEN集，把它的父结点设为n。到达n’的路径的代价g(n’)，设定为g(n) + movementcost(n, n’)。

(*)这里我忽略了一个小细节。你确实需要检查结点的g值是否更小了，如果是的话，需要重新打开（re-open）它。

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): **
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

(**) This should never happen if you have an admissible heuristic. However in games we often have inadmissible heuristics.

3.2 源代码

我自己的（旧的）C++A*代码是可用的：path.cpp (http://theory.stanford.edu/~amitp/ GameProgramming/path.cpp)和path.h (http://theory.stanford.edu/~amitp/GameProgramming/ path.h)，但是不容易阅读。还有一份更老的代码（更慢的，但是更容易理解），和很多其它的A*实现一样，它在Steve

*Woodcock*的游戏*AI*页面（*http://www.gameai.com/ai.html*）。

在网上，你能找到*C*，*C++*，*Visual Basic*，*Java(http://www.cuspy.com/software/pathfinder/ doc/)*，*Flash/Director/Lingo, C# (http://www.codeproject.com/csharp/CSharpPathfind.asp), Delphi, Lisp, Python, Perl,* 和*Prolog* 实现的*A**代码。一定的阅读*Justin Heyes-Jones*的*C++*实现（*http://www.geocities.com/jheyesjones/astar.html*）。

3.3 集合的表示

你首先想到的用于实现*OPEN*集和*CLOSED*集的数据结构是什么？如果你和我一样，你可能想到“数组”。你也可能想到“链表”。我们可以使用很多种不同的数据结构，为了选择一种，我们应该考虑我们需要什么样的操作。

在*OPEN*集上我们主要有三种操作：主循环重复选择最好的结点并删除它；访问邻居结点时需要检查它是否在集合里面；访问邻居结点时需要插入新结点。插入和删除**最佳是**优先队列（*http://members.xoom.com/killough/heaps.html*）的典型操作。

选择哪种数据结构不仅取决于操作，还取决于每种操作执行的次数。检查一个结点是否在集合中这一操作对每个被访问的结点的每个邻居结点都执行一次。删除最佳操作对每个被访问的结点都执行一次。被考虑到的绝大多数结点都会被访问；不被访问的是搜索空间边缘（*fringe*）的结点。当评估数据结构上面的这些操作时，必须考虑*fringe(F)*的最大值。

另外，还有第四种操作，虽然执行的次数相对很少，但还是必须实现的。如果正被检查的结点已经在*OPEN*集中（这经常发生），并且如果它的*f*值比已经在*OPEN*集中的结点要好（这很少见），那么*OPEN*集中的**值必须**被调整。调整操作包括删除结点（*f*值不是最佳的结点）和重插入。这两个步骤必须被最优化为一个步骤，这个步骤将移动结点。

3.3.1 未排序数组或链表

最简单的数据结构是未排序数组或链表。集合关系检查操作（*Membership test*）很慢，扫描整个结构花费*O(F)*。插入操作很快，添加到末尾花费*O(1)*。查找最佳元素（*Finding the best element*）很慢，扫描整个结构花费*O(F)*。对于数组，删除最佳元素（*Removing the best element*）花费*O(F)*，而链表则是*O(1)*。调整操作中，查找结点花费*O(F)*，改变值花费*O(1)*。

3.3.2 排序数组

为了加快删除最挂操作，可以对数组进行排序。集合关系检查操作将变成*O(log F)*，因为我们可以使用折半查找。插入操作会很慢，为了给新元素腾出空间，需要花费 *O(F)*以移动所有的元素。查找最佳元素操作会很快，因为它已经在末尾了所以花费是*O(1)*。如果我们保证最佳排序至数组的尾部（*best sorts to the end of the array*），删除最佳元素操作花费将是*O(1)*。调整操作中，查找结点花费*O(logF)*，改变值/位置花费*O(F)*。

3.3.3 排序链表

在排序数组中，插入操作很慢。如果使用链表则可以加速该操作。集合关系检查操作很慢，需要花费*O(F)*用于扫描链表。插入操作是很快的，插入新元素只花费*O(1)*时间，但是查找正确位置需要花费*O(F)*。查找最佳元素很快，花费*O(1)*时间，因为最佳元素已经在表的尾部。删除最佳元素也是*O(1)*。调整操作中，查找结点花费*O(F)*，改变值/位置花费*O(1)*。

3.3.4 排序跳表

在未排序链表中查找元素是很慢的。如果用跳表（*http://en.wikipedia.org/wiki/Skip_list*）代替链表的话，可以加速这个操作。在跳表中，如果有排序键（*sort key*）的话，集合关系检查操作会很快：*O(log F)*。如果你知道在何处插入的话，和链表一样，插入操作也是*O(1)*。如果排序键是*f*，查找最佳元素很快，达到*O(1)*，删除一个元素也是*O(1)*。调整操作涉及到查找结点，删除结点和重插入。

如果我们用地图位置作为跳表的排序键，集合关系检查操作将是*O(log F)*。在完成集合关系检查后，插入操作是*O(1)*。查找最佳元素是*O(F)*，删除一个结点是*O(1)*。因为集合关系检查更快，所以它比未排序链表要好一些。

如果我们用*f*值作为跳表的排序键，集合关系检查操作将是*O(F)*。插入操作是*O(1)*。查找最佳元素是*O(1)*，删除一个结点是*O(1)*。这并不比排序链表好。

3.3.5 索引数组

如果结点的集合有限并且数目是适当的，我们可以使用直接索引结构，索引函数*i(n)*把结点*n*映射到一个数组的索引。未排序与排序数组的长度等于*OPEN*集的最大值，和它们不同，对所有的*n*，索引数组的长度总是等于*max(i(n))*。如果你的函数是密集的（没有不被使用的索引），*max(i(n))*将是你地图中结点的数目。只要你的地图是网格的，让索引函数密集就是容易的。

假设*i(n)*是*O(1)*的，集合关系检查将花费*O(1)*，因为我们几乎不需要检查*Array[i(n)]*是否包含任何数据。*Insertion is O(1), as we just ste Array[i(n)];*查找和删除最佳操作是*O(numnodes)*，因为我们必须搜索整个结构。调整操作是*O(1)*。

3.3.6 哈希表

索引数组使用了很多内存用于保存不在*OPEN*集中的所有结点。一个选择是使用哈希表。哈希表使用了一个哈希函数*h(n)*把地图上每个结点映射到一个哈希码。让哈希表的大小等于*N*的两倍，以使发生冲突的可能性降低。假设*h(n)* 是*O(1)*的，集体关系检查操作花费*O(1)*；插入操作花费*O(1)*；删除最佳元素操作花费*O(numnodes)*，因为我们需要搜索整个结构。调整操作花费*O(1)*。

3.3.7 二元堆

一个二元堆（不要和内存堆混淆）是一种保存在数组中的树结构。和许多普通的树通过指针指向子结点所不同，二元**堆使用**索引来查找子结点。*C++ STL*包含了一个二元堆的高效实现，我在我自己的*A**代码中使用了它。

在二元堆中，集体关系检查花费*O(F)*，因为你必须扫描整个结构。插入操作花费*O(log F)*而删除最佳操作花费也是*O(log F)*。调整操作很微妙（*tricky*），花费*O(F)*时间找到节点，并且很神奇，只用*O(log F)*来调整。

我的一个朋友（他研究用于最短路径算法的数据结构）说，除非在你的*fringe*集里有多于*10000*个元素，否则二元堆是很不错的。除非你的游戏地图特别大，否则你不需要更复杂的数据结构（如*multi-level buckets*（*http://www-cs-students.stanford.edu/~csilvers/*））。你应该尽可能不用*Fibonacci* 堆（*http://www.star-lab.com/goldberg/pub/neci-tr-96-062.ps*），因为虽然它的渐近复杂度很好，但是执行起来很慢，除非*F*足够大。

3.3.8 伸展树

堆是一种基于树的结构，它有一个期望的*O(log F)*代价的时间操作。然而，问题是在*A**算法中，通常的情况是，一个代价小的节点被移除（花费*O(log F)*的代价，因为其他结点必须从树的底部向上移动），而紧接着一些代价小的节点被添加（花费*O(log F)*的代价，因为这些结点被添加到底部并且被移动到最顶部）。在这里，堆的操作在预期的情况下和最坏情况下是一样的。如果我们找到这样一种数据结构，最坏情况还是一样，而预期的情况好一些，那么就可以得到改进。

伸展树（*Splay tree*）是一种自调整的树结构。任何对树结点的访问都尝试把该结点推到树的顶部（*top*）。这就产生了一个缓存效果（*"caching" effect*）：很少被使用的结点跑到底部（*bottom*）去了并且不减慢操作（*don't slow down operations*）。你的*splay*树有多大并不重要，因为你的操作仅仅和你的“*cache size*”一样慢。在*A**中，低代价的结点使用得很

多，而高代价结点经常不被使用，所以高代价结点将会移动到树的底部。

使用伸展树后，集体关系检查，插入，删除最佳和调整操作都是期望的*O(log F)*（注：原文为*expected O(log F)*），最坏情况是*O(F)*。然而有代表性的是，缓存过程（*caching*）避免了最坏情况的发生。*Dijkstra*算法和带有低估的启发函数（*underestimating heuristic*）的*A**算法却有一些特性让伸展树达不到最优。特别是对结点*n*和邻居结点*n'*来说，*f(n') >= f(n)*。当这发生时，也许插入操作总是发生在树的同一边结果是使它失去了平衡。我没有试验过这个。

*3.3.9 HOT*队列

还有一种**比堆好的**数据结构。通常你可以限制优先队列中值的范围。给定一个限定的范围，经常会存在更好的算法。例如，对任意值的排序可以在*O(N log N)*时间内完成，但当固定范围时，桶排序和基数排序可以在*O(N)*时间内完成。

我们可以使用*HOT*（*Heap On Top*）队列*(http://www.star-lab.com/goldberg/pub /neci-tr-97-104.ps*）来利用*f(n') >= f(n)*，其中*n'*是*n*的一个邻居结点。我们删除*f(n)*值最小的结点*n*，插入满足*f(n) <= f(n') <= f(n) + delta*的邻居*n'*，其中*delta <= C*。常数*C*是从一结点到邻近结点代价改变量的最大值。因为*f(n)*是*OPEN*集中的最小*f*值，并且正要被插入的所有结点都小于或等于*f(n) + delta*，我们知道*OPEN*集中的所有*f*值都不超过一个*0..delta*的范围。在桶/基数排序中，我们可以用“桶”（*buckets*）对*OPEN*集中的结点进行排序。

使用*K*个桶，我们把*O(N)*的代价降低到平均*O(N/K)*。通过*HOT*队列，顶端的**桶使用**二元堆而所有其他的桶都是未排序数组。因而，对顶部的桶，集合关系检查代价是预期的*O(F/K)*，插入和删除**最佳是***O(log (F/K))*。对其他桶，集合关系检查是*O(F/K)*，插入是*O(1)*，而删除最佳根本不发生！如果顶端的桶是空的，那么我们必须把下**一个桶即未**排序数组转换为二元堆。这个操作（“*heapify*”）可以在*O(F/K)*时间内完成。在调整操作中，删除是*O(F/K)*，然后插入是*O(log (F/K))*或*O(1)*。

在*A**中，我们加入*OPEN*集中的许多结点实际上根本是不需要的。在这方面*HOT*队列很有优势，因为不需要的元素的插入操作只花费*O(1)*时间。只有需要的元素被*heapified*（代价较低的那些）。唯一**一个**超过*O(1)*的操作是从堆中删除结点，只花费*O(log (F/K))*。

另外，如果*C*比较小，我们可以只让*K = C*，则对于最小的桶，我们甚至不需要一个堆，国为在一个桶中的所有结点都有相同的*f*值。插入和删除**最佳都是***O(1)*时间！有人研究过，*HOT*队列在至多在*OPEN*集中有*800*个结点时和**堆一样**快，并且如果*OPEN*集中至多有*1500*个结点，则**比堆快***20%*。我期望随着结点的增加，*HOT*队列也更快。

*HOT*队列的一个简单的变化是一个二层队列（*two-level queue*）：把好的结点放进一个数据结构（堆或数组）而把坏的结点放进另一个数据结构（数组或链表）。因为大多数进入*OPEN*集中的结点都“坏的”，它们从不被检查，因而把它们放进出一个大数组是没有害处的。

3.3.10 比较

注意有一点很重要，我们并不是仅仅关心渐近的行为（大*O*符号）。我们也需要关心小常数（*low constant*）下的行为。为了说明原因，考虑一个*O(log F)*的算法，和另一个*O(F)*的算法，其中*F*是堆中元素的个数。也许在你的机器上，第一个算法的实现花费*10000*log(F)*秒，而另一个的实现花费*2*F*秒。当*F=256*时，第一个算法将花费*80000*秒而第二个算法花费*512*秒。在这种情况下，“更快”的算法花费更多的时间，而且只有在当*F>200000*时才能运行得更快。

你不能仅仅比较两个算法。你还要比较算法的实现。同时你还需要知道你的数据的大小（*size*）。在上面的例子中，第一种实现在*F>200000*时更快，但如果在你的游戏中，*F*小于*30000*，那么第二种实现好一些。

基本数据结构没有一种是完全合适的。未排序数组或者链表使插入操作很快而集体关系检查和删除操作非常慢。排序数组或者链表使集体关系检查稍微快一些，删除（最佳元素）操作非常快而插入操作非常慢。**二元堆让插入**和删除操作稍微快一些，而集体关系检查则很慢。伸展树让所有操作都快一些。*HOT*队 列让插入操作很快，删除操作相当快，而集体关系检查操作稍微快一些。索引数组让集体关系检查和插入操作非常快，但是删除操作不可置信地慢，同时还需要花费 很多内存空间。哈希表和索引数组类似，但在普通情况下，它花费的内存空间少得多，而删除操作虽然还是很慢，但比索引数组要快。

关于更高级的优先队列的资料和实现，请参考*Lee Killough*的优先队列页面（*http://members.xoom.com/killough/heaps.html*）。

3.3.11 混合实现

为了得到最佳性能，你将希望使用混合数据结构。在我的*A**代码中，我使用一个索引数组从而集合关系检查是*O(1)*的，一个二元堆从而插入操作和删除**最佳都是***O(log F)*的。对于调整操作，我使用索引数组从而花费*O(1)*时间检查我是否真的需要进行调整（通过在索引数组中保存*g*值），然后在少数确实需要进行调整的情况中，我使用二元堆从而调整操作花费*O(F)*时间。你也可以使用索引数组保存堆中每个结点的位置，这让你的调整操作变成*O(log F)*。

3.4 与游戏循环的交互

交互式的（尤其是实时的）游戏对最佳路径的计算要求很高。能够得到一个解决方案比得到最佳方案可能更重要。然而在所有其他因素都相同的情况下，短路径比长路径好。

一般来说，计算靠近初始结点的路径比靠近目标结点的路径更重要一些。立即开始原理（*The principle of immediate start*）：让游戏中的物体尽可能快地开始行动，哪怕是沿着一条不理想的路径，然后再计算一条更好的路径。在实时游戏中，应该更多地关注*A**的延迟情况（*latency*）而不是吞吐量（*throughput*）。

可以对物体编程让它们根据自己的本能（简单行为）或者智力（一条预先计算好的路径）来行动。除非它们的智力告诉它们怎么行动，否则它们就根据自己的本能来行动（这是实际上使用的方法，并且*Rodney Brook*在他的机器人体系结构中也用到）。和立即计算所有路径所不同，让游戏在每一个，两个，或者三个循环中搜索一条路径。让物体在开始时依照本能行动*(*可能仅仅是简单地朝着目标直线前进*)*，然后才为它们寻找路径。这种方法让让路径搜索的代价趋于平缓，因此它不会集中发生在同一时刻。

3.4.1 提前退出

可以从*A**算法的主循环中提前退出来同时得到一条局部路径。通常，当找到目标结点时，主循环就退出了。然而，在此之前的任意结点，可以得到一条到达*OPEN*中当前最佳结点的路径。这个结点是到达目标点的最佳选择，所以它是一个理想的中间结点（原文为*so it's a reasonable place to go*）。

可以提前退出的情况包括检查了一定数量的结点，*A**算法已经运行了几毫秒时间，或者扫描了一个离初始**点有些**距离的结点。当使用路径拼接时，应该给被拼接的路径一个比全路径（*full path*）小的最大长度。

3.4.2 中断算法

如果需要进行路径搜索的物体较少，或者如果用于保存*OPEN*和*CLOSED*集的数据结构较小，那么保存算法的状态是可行的，然后退出到游戏循环继续运行游戏。

3.4.3 组运动

路径请求并不是均匀分布的。即时策略游戏中有一个常见的情况，玩家会选择多个物体并命令它们朝着同样的目标移动。这给路径搜索系统以沉重的负载。

在这种情况下，为某个物体寻找到的路径对其它物体也是同样有用的。一种方法是，寻找一条从物体的中心到目的地中心的路径*P*。对所有物体使用该路径的绝大部分，对每一个物体，前十步和后十步使用为它自己寻找的路径。物体*i*得到一条从它的开始点到*P[10]*的路径，紧接着是共享的路径*P[10..len(P)-10]*，最后是从*P[len(P)-10]*到目的地的路径。

为每个物体寻找的路径是较短的（平均步数大约是*10*），而较长的路径被共享。大多数路径只寻找一次并且为所有物体所共享。然而，当玩家们看到所有的物体都沿着相同的路径移动时，将对游戏失去兴趣。为了对系统做些改进，可以让物体稍微沿着不同的路径运动。一种方法是选择邻近结点以改变路径。

另一种方法是让每个物体都意识到其它物体的存在（或许是通过随机选择一个“领导”物体，或者是通过选择一个能够最好地意识到当前情况的物体），同时仅仅为领导寻路。然后用*flocking*算法让它们以组的形式运动。

然而还有一种方法是利用*A**算法的中间状态。这个状态可以被朝着相同目标移动的多个物体共享，只要物体共享相同的启发式函数和代价函数。当主循环退出时，不要消除*OPEN*和*CLOSED*集；用*A**上一次的*OPEN*和*CLOSED*集开始下一次的循环（下一个物体的开始位置）。（这可以被看成是中断算法和提前退出部分的一般化）

3.4.4 细化

如果地图中没有障碍物，而有不同代价的地形，那么可以通过低估地形的代价来计算一条初始路径。例如，如果草地的代价是*1*，山地代价是*2*，山脉的代价是*3*，那么*A**会考虑通过*3*个草地以避免*1*个山脉。通过把草地看成*1*，山地看成*1.1*，而山脉看成*1.2*来计算初始路径，*A**将会用更少的时间去设法避免山脉，而且可以更快地找到一条路径（这接近于精确启发函数的效果）。一旦找到一条路径，物体就可以开始移动，游戏循环就可以继续了。当多余的*CPU*时间是可用的时候，可以用真实的移动代价去计算更好的路径。

*4 A**算法的变种

4.1 beam search

在*A**的主循环中，*OPEN*集保存所有需要检查的结点。*Beam Search*是*A**算法的一个变种，这种算法限定了*OPEN*集的尺寸。如果*OPEN*集变得过大，那些没有机会通向一条好的路径的结点将被抛弃。缺点是你必须让排序你的集合以实现这个，这限制了可供选择的数据结构。

4.2 迭代深化

迭代深化是一种在许多*AI*算法中使用的方法，这种方法从一个近似解开始，逐渐得到更精确的解。该名称来源于游戏树搜索，需要查看前面几步（比如在象棋里），通过查看前面更多步来提高树的深度。一旦你的**解不再**有更多的改变或者改善，就可以认为你已经得到足够好的解，当你想要进一步精确化时，它不会再有改善。在*ID-A**中，深度是*f*值的一个*cutoff*。当*f*的值太大时，结点甚至将不被考虑（例如，它不会被加入*OPEN*集中）。第一次**迭代只**处理很少的结点。此后每一次迭代，访问的结点都将增加。如果你发现路径有所改善，那么就继续增加*cutoff*，否则就可以停止了。更多的细节请参考这些关于*ID-A**的资料：*http://www.apl.jhu.edu/~hall/AI-Programming/IDA-Star.html*。

我本人认为在游戏地图中没有太大的必要使用*ID-A**寻路。*ID*算法趋向于增加计算时间而减少内存需求。然而在地图路径搜索中，“结点”是很小的——它们仅仅是坐标而已。我认为不保存这些结点以节省空间并不会带来多大改进。

4.3 动态衡量

在动态衡量中，你假设在开始搜索时，最重要的**是讯速**移动到任意位置；而在搜索接近结束时，最重要的是移动到目标点。

*f(p) = g(p) + w(p) * h(p)*

启发函数中带有一个权值（*weight*）（*w>=1*）。当你接近目标时，你降低这个权值；**这降低**了启发函数的重要性，同时增加了路径真实代价的相对重要性。

4.4 带宽搜索

带宽搜索（*Bandwidth Search*）有两个对有些人也许有用的特性。这个变种假设*h*是过高估计的值，但不高于某个数*e*。如果这就是你遇到的情况，那么你得到的路径的代价将不会比最佳路径的代价超过*e*。重申一次，你的启发函数设计的越好，最终效果就越好。

另一个特性是，你可以丢弃*OPEN*集中的某些结点。当*h+d*比路径的真实代价高的时候（对于某些*d*），你可以丢弃那些*f*值比*OPEN*集中的最好结点的*f*值**高至少***e+d*的结点。这是一个奇怪的特性。对于好的*f*值你有一个“范围”（*"band"*），任何在这个范围之外的结点都可以被丢弃掉，因为这个结点肯定不会在最佳路径上。

好奇地（*Curiously*），你可以对这两种特性使用不同的启发函数，而问题仍然可以得到解决。使用一个启发函数以保证你得到的路径不会太差，另一个用于检查从*OPEN*集中去掉哪些结点。

4.5 双向搜索

与从开始点向目标点搜索不同的是，你也可以并行地进行两个搜索——一个从开始点向目标点，另一个从目标点**向开始**点。当它们相遇时，你将得到一条好的路径。

这听起来是个好主意，但我不会给你讲很多内容。双向搜索的思想是，搜索过程生成了一棵在地图上散开的树。一棵大树比两棵小树差得多，所以最好是使用两棵较小的搜索树。然而我的试验表明，在*A**中你得不到一棵树，而只是在搜索地图中当前位置附近的区域，但是又不像*Dijkstra*算法那样散开。事实上，这就是让*A**算法运行得如此快的原因——无论你的路径有多长，它并不进行疯狂的搜索，除非路径是疯狂的。它只尝试搜索地图上小范围的区域。如果你的地图很复杂，双向搜索会更有用。

面对面的方法（*The front-to-front variation*）把这两种搜索结合在一起。这种算法选择一对具有最好的*g(start,x) + h(x,y) + g(y,goal)*的结点，而不是选择最好的前向搜索结点——*g(start,x) + h(x,goal)*，或者最好的后向搜索结点——*g(y,goal) + h(start,y)*。

*Retargeting*方法不允许前向和后向搜索同时发生。它朝着某个最佳的中间结点运行前向搜索一段时间，然后再朝这个结点运行后向搜索。然后选择一个后向最佳中间结点，从前向最佳中间结点向后向最佳中间结点搜索。一直进行这个过程，直到两个中间结点碰到一块。

4.6 动态*A**与终身计划*A**

有一些*A**的变种允许当初始路径计算出来之后，世界发生改变。*D**用于当你没有全局所有信息的时候。如果你没有所有的信息，*A**可能会出错；*D**的贡献在于，它能纠正那些错误而不用过多的时间。*LPA**用于代价会改变的情况。在*A**中，**当地图发生**改变时，路径将变得无效；*LPA**可以重新使用之前*A**的计算结果并产生新的路径。然而，*D**和*LPA**都需要很多内存——用于运行*A**并保存它的内部信息（*OPEN*和*CLOSED*集，路径树，*g*值），**当地图发生**改变时，*D**或者*LPA**会告诉你，是否需要就地图的改变对路径作调整。在一个有许多运动着的物体的游戏中，你经常不希望保存所有这些信息，所以*D**和*LPA**在这里并不适用。它们是为机器人技术而设计的，这种情况下只有一个机器人——你不需要为别的机器人寻路而重用内存。如果你的游戏只有一个或者少数几个物体，你可以研究一下*D**或者*LPA**。

- *Overview of D**（*http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html*）
- *D* Paper 1*（*http:// http://www.frc.ri.cmu.edu/~axs/doc/icra94.ps*）
- *D* Paper 2*（*http:// http://www.frc.ri.cmu.edu/~axs/doc/ijcai95.ps*）
- *Lifelong planning overview*（*http://idm-lab.org/project-a.html*）
- *Lifelong planning paper (PDF)*（*http://csci.mrs.umn.edu/UMMCsciwiki/pub/Csci3903s03/KellysPaper/seminar.pdf*）

- *Lifelong planning A* applet*（*http://idm-lab.org/applet.html*）

5 处理运动障碍物

一个路径搜索算法沿着固定障碍物计算路径，但是当障碍物会运动时情况又怎样？当一个物体到达一个特写的位置，原来的障碍物也许不再在那儿了，或者一个新的障碍物也许到达那儿。处理该问题的一个方法是放弃路径搜索而使用运动算法（*movement algorithms*）替代，这就不能*look far ahead*：这种方法会在后面的部分中讨论。这一部分将对路径搜索方法进行修改从而解决运动障碍物的问题。

5.1 重新计算路径

当时间渐渐过去，我们希望游戏世界有所改变。以前搜索到的一条路径到现在也许不再是最佳的了。对旧的路径用新的信息进行更新是有价值的。以下规则可以用于决定什么时候需要重新计算路径：

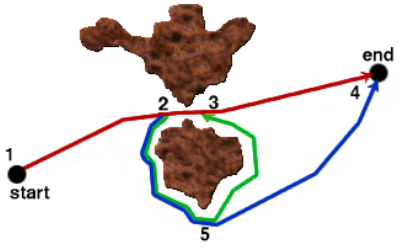
- 每*N*步：这保证用于计算路径的信息不会旧于*N*步。
- 任何可以使用额外的*CPU*时间的时候：这允许动态调整路径的性质；在物体数量多时，或者运行游戏的机器比较慢时，每个物体对*CPU*的使用可得到减少。
- 当物体拐弯或者跨越一个导航点（*waypoint*）的时候。
- 当物体附近的世界改变了的时候。

重计算路径的主要缺点是许多路径信息被丢弃了。例如，如果路径是*100*步长，每*10*步重新计算一次，路径的总步数将是*100+90+80+70+60+50+40+30+20+10 = 550*。对*M*步长的路径，大约需要计算*M^2*步。因此如果你希望有许多很长的路径，重计算不是个好主意。重新使用路径信息比丢弃它更好。

5.2 路径拼接

当 一条路径需要被重新计算时，意味着世界正在改变。对于一个正在改变的世界，对地图中当前邻近的区域总是比对远处的区域了解得更多。因此，我们应该集中于在 附近寻找好的路径，同时假设远处的路径不需要重新计算，除非我们接近它。与重新计算整个路径不同，我们可以重新计算路径的前*M*步：

1. 令*p[1]..p[N]*为路径（*N*步）的剩余部分
2. 为*p[1]*到*p[M]*计算一条新的路径
3. 把这条新路径拼接（*Splice*）到旧路径：把*p[1]..p[M]*用新的路径值代替



4.

因为*p[1]*和*p[M]*比分开的*M*步小（原文：*Since p[1] and p[M] are fewer than M steps apart*），看起来新路径不会很长。不幸的是，新的路径也许很长而且不够好。上面的图显示了这种情况。最初的红色路径是*1-2-3-4*，褐色的是障碍物。如果我们到达*2*并且发现从*2*到达*3*的路径被封锁了，路径拼接技术会把*2-3*用*2-5-3*取代，结果是物体沿着路径*1-2-5-3-4*运动。我们可以看到这不是一条好的路径，蓝色的路径*1-2-5-4*是一条更好的路径。

通常可以通过查看新路径的长度检测到坏的路径。如果这严格大于*M*，就可能是不好的。一个简单的解决方法是，为搜索算法设置一个最大路径长度。如果找不到一条短的路径，算法返回错误代码；这种情况下，用重计算路径取代路径拼接，从而得到路径*1-2-5-4*.。

对于其它情况，对于*N*步的路径，路径拼接会计算*2N*或者*3N*步，这取决于拼接新路径的频率。对于对世界的改变作反应的能力而言，这个代价是相当低的。令人吃惊的是这个代价和拼接的步数*M*无关。*M*不影响*CPU*时间，而控制了响应和路径质量的折衷。如果*M*太大，物体的移动将不能快速对地图的改变作出反应。如果*M*太小，拼接的路径可能太短以致不能正确地绕过障碍物；许多不理想的路径（如*1-2-5-3-4*）将被找到。尝试不同的*M*值和不同的拼接标准*(*如每*3/4 M*步*)*，看看哪一种情况对你的地图最合适。

路径拼接确实比重计算路径要快，但它不能对路径的改变作出很好的反应。经常可以发现这种情况并用路径重计算来取代。也可以调整一些变量，如*M*和寻找新路径的时机，所以可以对该方法进行调整*(*甚至在运行时*)*以用于不同的情况。

Note：*Bryan Stout* 有两个算法，*Patch-One*和*Patch-All*，他从路径拼接中得到灵感，并在实践中运行得很好。他出席了*GDC 2007*（*https://www.cmpevents.com/GD07/a.asp?option =C &V=11& SessID=4608*）；一旦他把资料放在网上，我将链接过去。

***Implementation Note*:**

反向保存路径，从而删除路径的开始部分并用不同长度的新路径拼接将更容易，因为这两个操作都将在数组的末尾进行。本质上你可以把这个数组看成是堆栈因为顶部的元素总是下一个要使用的。

5.3 监视地图变化

与 间隔一段时间重计算全部或部分路径不同的是，可以让地图的改变触发一次重计算。地图可以分成区域，每个物体都可以对某些区域感兴趣（可以是包含部分路径的 所有区域，也可以只是包含部分路径的邻近区域）。当一个障碍物进入或者离开一个区域，该区域将被标识为已改变，所有对该区域感兴趣的物体都被通知到，所以 路径将被重新计算以适应障碍物的改变。

这种技术有许多变种。例如，可以每隔一定时间通知物体，而不是立即通知物体。多个改变可以成组地触发一个通知，因此避免了额外的重计算。另一个例子是，让物体检查区域，而不是让区域通知物体。

监视地图变化允许当障碍物不改变时物体避免重计算路径，所以当你有许多区域并不经常改变时，考虑这种方法。

5.4 预测障碍物的运动

如果障碍物的运动可以预测，就能为路径搜索考虑障碍物的未来位置。一个诸如A*的算法有一个代价函数用以检查穿过地图上一点的代价有多难。A*可以被改进从而知道到达一点的时间需求（通过当前路径长度来检查），而现在则轮到代价函数了。代价函数可以考虑时间，并用预测的障碍物位置检查在某个时刻地图某个位置是否可以通过。这个改进不是完美的，然而，因为它并不考虑在某个点等待障碍物自动离开的可能性，同时A*并不区分到达相同目的地的不同的路径，而是针对不同的目的地，所以还是可以接受的。

6 预计算路径的空间代价

有时，路径计算的限制因素不是时间，而是用于数以百计的物体的存储空间。路径搜索器需要空间以运行算法和保存路径。算法运行所需的临时空间（在A*中是OPEN和CLOSED集）通常比保存结果路径的空间大许多。通过限制在一定的时间计算一条路径，可以把临时空间数量最小化。另外，为OPEN和CLOSED集所选择的数据结构的不同，最小化临时空间的程度也有很大的不同。这一部分聚集于优化用于计算路径的空间代价。

6.1 位置VS方向

一条路径可以用位置或者方向来表示。位置需要更多的空间，但是有一个优点，易于查询路径中的任意位置或者方向而不用沿着路径移动。当保存方向时，只有方向容易被查询；只有沿着整个路径移动才能查询位置。在一个典形的网格地图中，位置可以被保存为两个16位整数，每走一步是32位。而方向是很少的，因此用极少的空间就够了。如果物体只能沿着四个方向移动，每一步用两位就够了；如果物体能沿着6个或者8个方向移动，每一步也只需要三位。这些对于保存路径中的位置都有明显的空间节省。Hannu Kankaanpaa指出可以进一步减少空间需求，那就是保存相对方向（右旋60度）而不是绝对方向（朝北走）。有些相对方向对某些物体来说意义不大。比如，如果你的物体朝北移动，那么下一步朝南移动的可能性很小。在只有六种方向的游戏中，你只有五个有意义的方向。在某些地图中，也许只有三个方向（直走，左旋60度，右旋60度）有意义，而其它地图中，右旋120度是有效的（比如，沿着陡峭的山坡走之字形的路径时）。

6.2 路径压缩

一旦找到一条路径，可以对它进行压缩。可以用一个普通的压缩算法，但这里不进行讨论。使用特定的压缩算法可以缩小路径的存储，无论它是基于位置的还是基于方向的。在做决定之前，考察你的游戏中的路径以确定哪种压缩效果最好。另外还要考虑实现和调试，代码量，*and whether it really matters*.如果你有300个物体并且在同一时刻只有50个在移动，同时路径比较短（100步），内存总需求大概只有不到50k，总之，没有必要担心压缩的效果。

6.2.1 位置存储

在障碍物比地形对路径搜索影响更大的地图中，路径中有大部分是直线的。如果是这种情况，那么路径只需要包含直线部分的终止点（有时叫*waypoints*）。此时移动过程将包含检查下一结点和沿着直线向前移动。

6.2.2 方向存储

保存方向时，有一种情况是同一个方向保存了很多次。可以用简单的方法节省空间。

一种方法是保存方向以及朝着该方向移动的次数。和位置存储的优化不同，当一个方向并不是移动很多次时，这种优化的效果反而不好。同样的，对于那些可以进行位置压缩的直线来说，方向压缩是行不通的，因为这条直线可能没有和正在移动的方向关联。通过相对方向，你可以把“继续前进”当作可能的方向排除掉。*Hannu Kankaanpaa*指出，在一个八方向地图中，你可以去掉前，后，以及向左和向右135度（假设你的地图允许这个），然后你可以仅用两个比特保存每个方向。

另一种保存路径的方法是变长编码。这种想法是使用一个简单的比特（*0*）保存最一般的步骤：向前走。使用一个“*1*”表示拐弯，后边再跟几个比特表示拐弯的方向。在一个四方向地图中，你只能左转和右转，因此可以用“*10*”表示左转，“*11*”表示右转。

变长编码比*run length encoding*更一般，并且可以压缩得更好，但对于较长的直线路径则不然。序列（向北直走6步，左转，直走3步，右转，直走5步，左转，直走2步）用*run length encoding*表示是*[(NORTH, 6), (WEST, 3), (NORTH, 5), (WEST, 2)]*。如果每个方向用2比特，每个距离用8比特，保存这条路径需要40比特。而对于变长编码，你用1比特表示每一步，2比特表示拐弯——*[NORTH 0 0 0 0 0 0 10 0 0 0 11 0 0 0 0 0 0 10 0 0]*——一共24比特。如果初始方向和每次拐弯对应1步，则每次拐弯都节省了一个比特，结果只需要20比特保存这条路径。然而，用变长编码保存更长的路径时需要更多的空间。序列(向北直走200步)用*run length encoding*表示是*[(NORTH, 200)]*，总共需要10比特。用变长编码表示同样的序列则是*[NORTH 0 0 ...]*，一共需要202比特。

6.3 计算导航点

一个导航点（*waypoint*）是路径上的一个结点。与保存路径上的每一步不同，在进行路径搜索之后，一个后处理（*post-processing*）的步骤可能会把若干步*collapse*（译者：不好翻译，保留原单词）为一个简单的导航点，这经常发生在路径上那些方向发生改变的地方，或者在一个重要的（*major*）位置如城市。然后运动算法将在两个导航点之间运行。

6.4 极限路径长度

当地图中的条件或者秩序会发生改变时，保存一条长路径是没有意义的，因为在从某些点开始，后边的路径已经没有用了。每个物体都可以保存路径开始时的特定几步，然后当路径已经没用时重新计算路径。这种方法虑及了（*allows for*）对每个物体使用数据的总量的管理。

6.5 总结

在游戏中，路径潜在地花费了许多存储空间，特别是当路径很长并且有很多物体需要寻路时。路径压缩，导航点和*beacons*通过把多个步骤保存为一个较小数据从而减少了空间需求。*Waypoints rely on straight-line segments being common so that we have to store only the endpoints, while beacons rely on there being well-known paths calculated beforehand between specially marked places on the map.*（译者：此处不好翻译，暂时保留原文）如果路径仍然用了许多存储空间，可以限制路径长度，这就回到了经典的时间-空间折衷法：为了节省空间，信息可以被丢弃，稍后才重新计算它。