Introduction to A\*

在计算机中，移动一个物体是很简单的，但是路径搜索确是一个复杂的过程，就像路由器转发数据报一样，生成路由表需要经过复杂的计算，但是转发的过程确很简单。路径搜索的例子在游戏中极为常见，我们不仅是在尝试寻找最短可达路径，同样也要考虑实际情况下不同区域赋予不同探索权值时的最终路径开销。

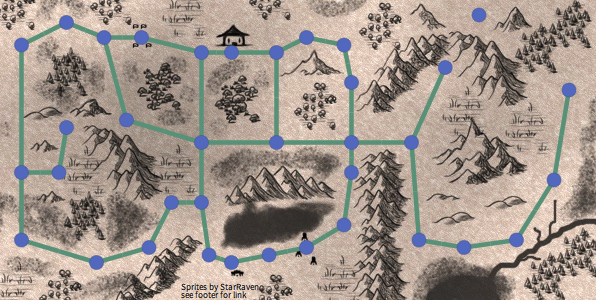


图1：地图抽象为简单图

在游戏中，我们首先要对地图进行抽象，把输入的地图抽象成一张简单的图，这样才能用于计算机的计算。对于路径搜索算法，我们主要可以从输入(input)和输出(output)两部分内容。

输入：将简单图的抽象作为输入内容，其中节点(node，图中为蓝色圆点)表示可以到达的位置，用边(edge，图中为绿色连线)表示节点对之间的联系。

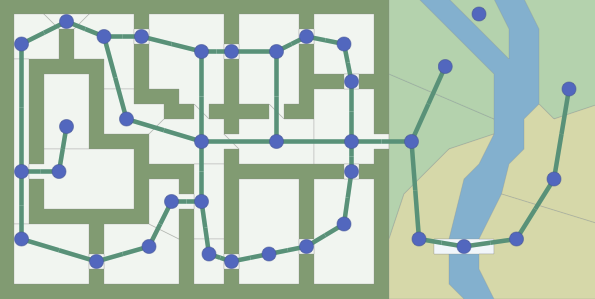


图2：简单图

输出：路径搜索的结果依旧是节点和路径，节点和路径都是原图的子集。对算法而言，它所知道的信息仅仅是一个抽象的图而非具体的实物，如图3和图4就拥有相同的节点和路径，但显然二者在地图中对应的并不是同一个图。所以游戏中，路径搜索算法只会告诉我们从哪个节点移动到哪个节点，并不知道该如何移动过去。

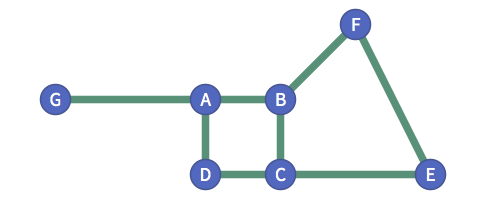


图3：示例1

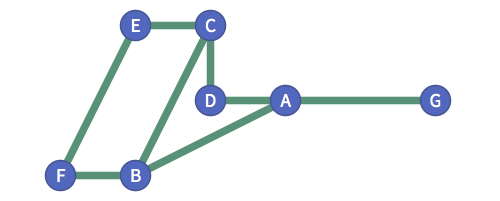


图4：示例2

游戏中，除了障碍物之外，地图中几乎所有的点都是可达的，为了更加简单并且能够可视化地展示寻路过程，我们会将地图抽象成更加简单直接的网格。有一点需要明确的就是，地图的表达方式会对路径搜索算法的性能和路径的质量产生巨大影响。通常情况下在途中越多的点被表达，运算的速度就会越慢。当我们使用网格来表示地图的时候，如果两条路径的长度相差k倍的话，我们需要搜索的节点的数量将会呈平方级增长。所以，实际上当整个图都被表示成网格的时候，效率是非常低下的。

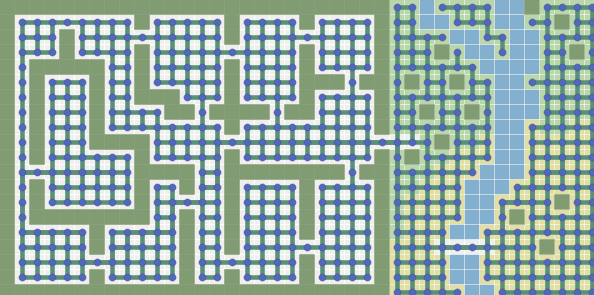


图5：网格图

创建网格的工作非常简单，只需要标记好每个节点的横、纵坐标即可。

all\_nodes = []

for x in range(20):

for y in range(10):

all\_nodes.append([x, y])

在将地图表示成网格之后，首先规定一下在网格图中每次移动的标准。

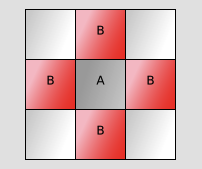


图6：可移动范围

假定目前所在的位置为A，可以从A直接移动到的位置为图中标记为红色的B区域。这里我们并没有允许可以在网格中到达对角区域，即可只能由当前位置往上下左右四个方向移动一格，我们可以用以下的函数得到网格图中具体某个节点可达的所有节点的坐标。

def neighbors(node):

dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]

result = []

for dir in dirs:

neighbor = [node[0] + dir[0], node[1] + dir[1]]

if 0 <= neighbor[0] < 20 and 0 <= neighbor[1] < 10:

result.append(neighbor)

return result

在完成了上面的定义工作之后，我们就可以真正开始讲解路径搜索算法了。虽然本文是在介绍A\*算法，但为了更好地理解它，需要先比较在相同条件下Breadth First Search(广度优先搜索)、Best First Search(最佳优先搜索)和A\*算法的执行过程及优劣。

Breadth First Search(广度优先算法)：

广度优先算法最大的特点就是试图探索所有方向上相连的节点。在网格图中会菱形状一层一层向外圈扩散。

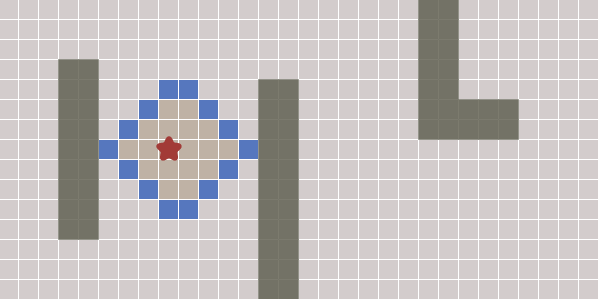


图7：广度优先搜索示例1

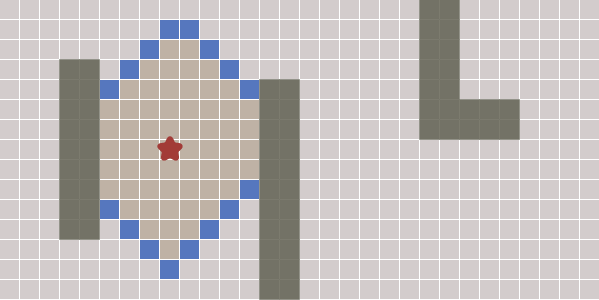


图8：广度优先搜索示例2

广度优先搜索的实现如下：

1. 从队列Frontier中移除一个节点。
2. 将移除的节点标记为已访问(之后不会再访问该节点)
3. 扩展被移除节点的所有邻居节点，将其所有不在队列Frontier中且未被访问过得节点加进队列中。

4. 重复上面的步骤直到所有的节点都被访问到。

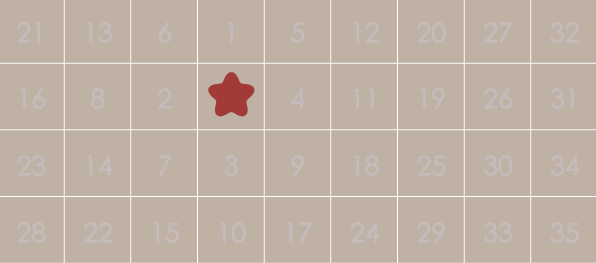


图9：BFS在网格图中搜索顺序

实现上述过程的代码描述：

frontier = Queue()

frontier.put(start )

visited = {}

visited[start] = True

while not frontier.empty():

current = frontier.get()

for next in graph.neighbors(current):

if next not in visited:

frontier.put(next)

visited[next] = True

上面这段代码循环虽然简单且效率很低，但却是很多图算法的本质。但是上面的代码只是告诉我们如何访问图中所有的节点，并没有发现最短路径以及构建路径。只需要在邻居节点被添加到Frontier时将它们的上一个节点(记录在Came\_from中)记录为刚刚被访问的节点即可。

frontier = Queue()

frontier.put(start )

came\_from = {}

came\_from[start] = None

while not frontier.empty():

current = frontier.get()

for next in graph.neighbors(current):

if next not in came\_from:

frontier.put(next)

came\_from[next] = current

但是刚刚的算法只是将途中所有的点都遍历了一遍，如果需要搜索某一条特定两点间路径的话，还需要设置一个终点，当程序执行过程时遇到终点时则立即结束程序。

frontier = Queue()

frontier.put(start )

came\_from = {}

came\_from[start] = None

while not frontier.empty():

current = frontier.get()

if current == goal:

break

for next in graph.neighbors(current):

if next not in came\_from:

frontier.put(next)

came\_from[next] = current

上面的例子中，广度优先遍历展示了在往上下左右四个方向移动一格代价均为1的网格图中寻找路径的方法。但是如果每步代价不完全相同，且要求寻找最小代价路径时，广度优先遍历就不能奏效了，这时就需要引入Dijksrtra算法来解决了。

Dijkstra算法：

1. 从优先队列PriorityQueue中选择具有最小代价节点。
2. 将该节点标记为已访问(以后不会再访问)。
3. 在所有尚未访问的节点中，如果从源节点经新加入节点中转再到另一节点的代价小于直接从源节点到该节点的代价，则更新最小代价值。
4. 重复以上过程直到访问到终点截至。

在网格图中，若每移动一格的代价恒为1的话，那么Dijkstra算法将会退化成广度优先遍历算法，在网格中会呈菱形的方式逐渐接近终点并得到一条最短路径。

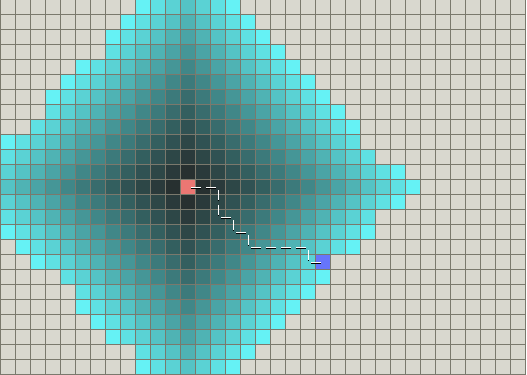


图10：Dijkstra算法在等权网格图中寻路

假定有网格图11，其中灰色部分表示平原，在平原上每移动一格的代价为1，绿色部分表示森林，穿越森林时每移动一格的代价为5。下图对比了使用广度有先搜索算法和Dijkstra算法时路径搜索过程，虽然二者都是从起点逐层向外推进的方式进行探索，但显然广度优先遍历算法只是寻找了一条连接起点和终点的路径，直接穿越了代价值极高的森林区域，并没有考虑其中的代价。不过Dijkstra算法通过计算避开了森林区域，选择了一条从起点到终点代价最低的路径。

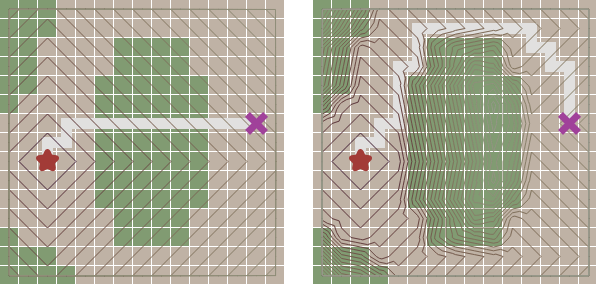


图11：权值不等时BFS和Dijkstra寻路对比

虽然Dijkstra算法在寻找最短路径时总能找到代价最小的路径，但由于每次都要选择当前权值最小的节点，所以整体算法复杂度还是比较高的。这时我们可以考虑最佳优先算法来完成最短路径的选择。

Best-First-Search(最佳优先算法):

1. 每次从优先队列PriorityQueue中选择启发值最低的节点。
2. 将移除节点标记为已访问。
3. 扩展所有与被选节点相连节点，将所有不在优先队列且未被访问过的节点计算过启发值后加入到队列中。
4. 重复以上过程直到访问到终点截至。

其实， 最佳优先搜索所做的操作和Dijkstra还是很相似的，只不过最佳优先搜索使用了评估值(启发值)作为从某节点决定选择下一个节点，因为启发值只是一个预估值，所以并不能保证选择的一定是最短路径，不过一般情况下其表现还是很不错的。

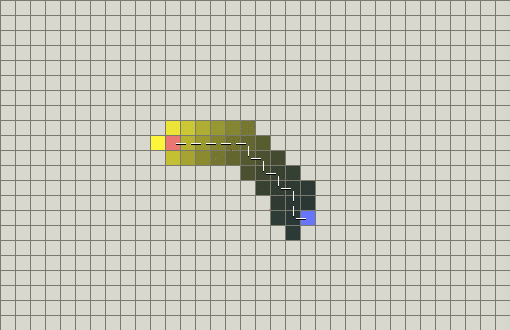


图12：最佳优先算法在等权网格中寻路

上图中各方格颜色表示了启发值的高低，颜色越黄启发值越高(越不会被选择)，颜色越黑表示启发值越低(越易被选择)。所以可见启发值的大小从宏观上来说决定了搜索的方向，因为算法总是选择启发值最低的路径。

但至于什么是启发值，本身启发函数的选取就是一个值得讨论的点，这里暂时不会展开，不过在这里可以将启发值理解成从某个节点到终点的一个估算的代价。

但是在某些具有障碍物的图中，最佳优先算法就不能很好地工作了，举个简单的例子如下图所示。

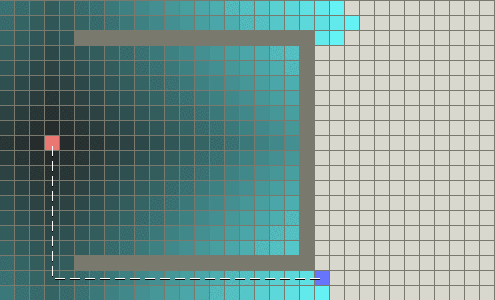


图13：有障碍物(等权)时Dijkstra算法

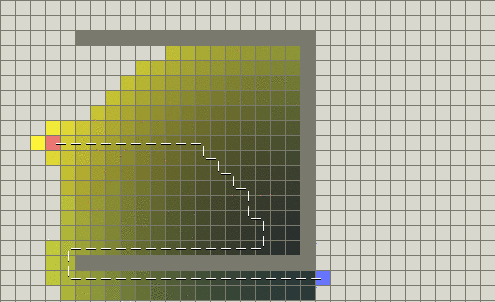


图14：有障碍物时(等权)时最佳优先搜索

显然，使用了贪心策略的最佳优先搜索算法明显选择了一条更长的路径，原因在于它试图考虑移向目标的代价却忽略了当前已花费的代价。虽然深色区域的启发值更低，颜色更深的区域大部分却被包在了障碍物内，这样等最佳优先搜索算法探索到障碍物边缘时，虽然有更低启发值的区域但却被障碍物所阻挡，只能贴着障碍物边缘重新选择绕出障碍的路径。但是使用Dijkstra算法虽然探测了更多的节点，将障碍物所围的所有区域都进行了探测，但是最终得到的路径却绕开了障碍物，选择了最短的路径。可见，虽然最佳优先搜索在启发值的帮助下，在无障碍图中常可以选择出比较不错的路径，但是在有障碍的图中，往往不能得到很好的结果。

总结一下上面三种算法，不难发现。

广度优先搜索算法在任意相连节点间路径权值均等时，在有障碍还是无障碍图都能得到结果，但结果未必最优。Dijkstra算法在路径间权相同或不同的图中均能找到从起点到终点的最短路径，但是搜索的节点众多且每次选择时要选出最优结果，时间开销大。最佳优先搜索在路径间权相同或不同的图中，无障碍下表现较好，有障碍时表现较差。

综上，我们其实寻求的理想状态下路径搜索算法实际上是满足：1.相连节点间路径代价可以不等；2.无论有无障碍，得到的路径代价尽可能是最小的；3.处理的无用节点尽可能少，搜索速度尽可能快。实际上，条件一已经将广度优先搜索排除了，虽然它提供了图搜索问题的一般思路。Dijkstra算法满足前两条条件，但是第三点并不能完全满足，最佳优先算法第三点满足但是第二点却有时并不能处理好。

所以，接下来介绍的A\*算法就是将二者的优点结合的一种算法，可以在保证运算速度的基础上也能保证最终代价较小。为了达到这个要求，A\*算法将Dijkstra算法中计算当前开销(从起点到当前节点)和最佳优先搜索中计算预估开销(从当前节点到终点的预估值)的方法结合起来，综合多种因素选取下一个到达的节点。

首先我们需要讨论A\*算法中几个最基本的概念：

g(n)：代表了从初始节点到当前节点的代价。

h(n)：代表了从当前节点到终点的启发值。

f(n)：f(n) = g(n) + h(n)，代表了起点到当前节点的精确值和从当前节点到终点的启发值之和。

Dijkstra算法是通过每次选取g(n)值最小的节点作为被选节点，最佳优先搜索算法是通过选取h(n)值最小的节点作为被选节点，而在主循环中A\*算法每次都选取f(n)值最小的节点， 图15中，当停留在A\*最终路径中任意一个点时，其值都等于前两幅图中g(n)的值和h(n)的值之和，并且最终路径中节点相较附近节点的f(n)值都是最小的。

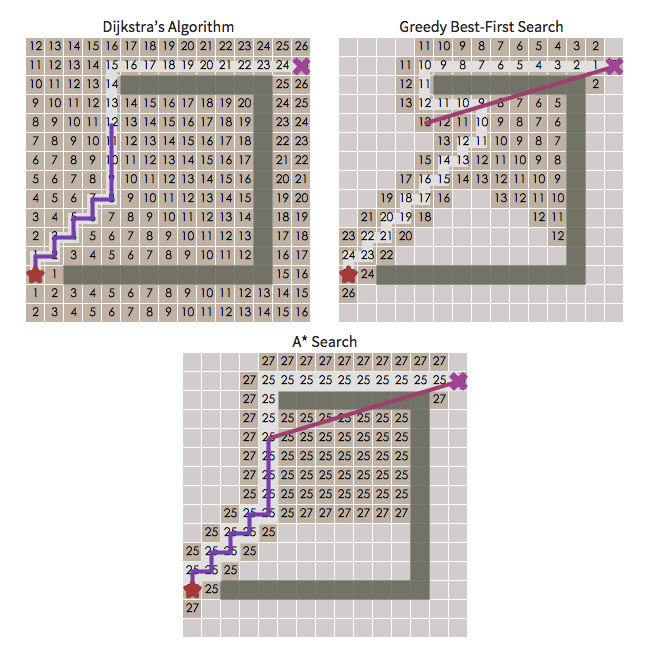


图15：A\*算法f(n) = g(n) + h(n)

实际上g(n)的值是在程序执行过程中当前花费的代价，虽然它对路径的选择也起到非常大的作用，不过我们在编程过程中可以通过改变启发函数h(n)的值来控制A\*算法的行为。下面总结了启发式函数h(n)对A\*算法行为的控制：

1. 在极限条件下，如果h(n)的值为0则g(n)起到主导作用，A\*算法变为Dijkstra算法，此时可以保证找到最短路径。
2. 如果h(n)的值总是小于或等于从当前节点移动到终点的实际值，h(n)的值越小，A\*算法需要搜索的节点越多，效率越低，但能保证得到的是最短路径。
3. 如果h(n)精确地等于当前节点移动到终点的实际值，A\*算法将会在不扩展任何别的节点的情况下寻找最短路径。
4. 如果h(n)有时比实际移动到终点的代价高，则A\*不能保证得到的一定是最短路径，但它通常会运行得比较快。
5. 如果h(n)的值远远大于g(n)，此时h(n)起主要作用，A\*算法演变成最佳优先搜索算法。

所以我们就遇到了一个很有趣的情况，我们需要A\*中获得什么，如果启发值低了，最终会得到最短路径但运行速度会比较慢；如果启发值高了，运行速度会提高很多但可能并不能得到最短路径。在游戏中，A\*算法的这一特性非常有用，在某些情况下，我们需要得到的仅仅是较好的路径而并不是最短路径，而这在A\*算法中正好可以通过平衡g(n)和h(n)的值来完成。

下面来讨论一下在网格地图中常用的的启发式函数。

曼哈顿距离：

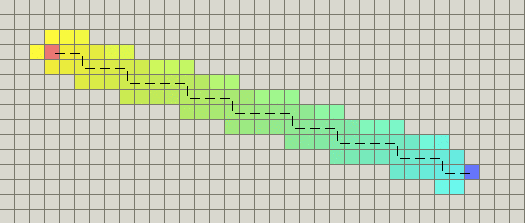


图16：曼哈顿距离

对于方格形网格，标准的启发式函数就是曼哈顿距离，这里的D通常是从一个位置移动到相邻位置的最小代价，在一个没有障碍的、最小移动代价为D地形上，每向目标移动一步，g就增加同时h减少4，因为f = g + h，f的值保持不变，这是启发式函数与代价函数的衡量单位匹配的一个标志。在简单情况下，可以将其设为1。在一个可以向四个方向移动的方向网格中，启发式函数值是曼哈顿距离的D倍。当然了，如果这里将D的值设置为略大于移动到相邻位置的最小代价，此时就会增加h所占比例，从而减少拓展节点数量，加快A\*算法的运行速度。

def heuristic(node):

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

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

return D \* (dx + dy)

对角线距离：

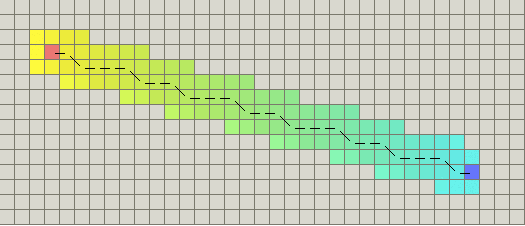


图17：对角线距离

如果允许在地图中沿着对角线移动，那么需要一个不同的启发式函数。在这里我们计算不走对角线所需的步数，然后减去走对角线节约的步数。在对角线上的步数有min(dx, dy)个，其每步的代价为D2，每步可以节约2 \* D的非对角线步数代价。

def heuristic(node):

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

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

return D \* (dx + dy) + (D2 - 2 \* D) \* min(dx, dy)

欧几里得距离：

如果允许任何角度的移动，那么可以考虑欧几里得距离。

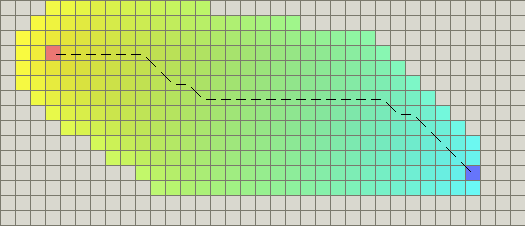


图18：欧几里得距离

这里代价函数g不会匹配启发函数h，由于欧几里得距离比曼哈对距离或对角线距离更短，所以仍然会得到最短路径，不过由于h的启发值比较小，所以A\*的实际运行时间会更长一些。但是由于开平方的开销会比较大，所以要么使用快速平方根逼近欧几里得距离要么使用对角线距离作为欧几里得距离。

def heuristic(node):

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

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

return D \* sqrt(dx \* dx + dy \* dy)