放学后茶会

Please don't say "You are lazy"

游戏开发笔记

浅析游戏开发中的寻路算法

Apr 05, 2022 阅读时长: 12 分钟

游戏开发中，只要有地图，基本上就避免不了要实现寻路。哪怕没有给玩家提供寻路功能，游戏中的 AI 实现也需要借助寻路来移动。

地图建模

不管是什么形式的地图，拿到它的第一步应该是设法为其建模，一份无法建模的地图数据是不能进行计算的，建模的目的是为了得到一个方便寻路算法使用的模型。

常用的寻路建模方式有划分格子（Grid），放置路点（WayPoint），生成导航网格（NavMesh）这三种。

划分格子

将地图分割成紧密相邻的若干个大小形状完全相同的正多边形，一般会使用是正方形，偶尔会用正六边形，基本没有别的形状的实现。

一般会以地图上的对象可以通行的最小宽度为单位来划分格子，使用二维数组即可方便保存和读取建模后的数据。使用不同的数字可以标识出地图格子不同的状态，比如可以简单的将 1 作为可以行走的格子，0 作为不可行走的格子。

```
1 local map = {
2     {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
```

```
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1},
    {1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1},
    {1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1},
    {1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1},
    {1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
}
```

划分格子一般用在二维地图上，它的优点是实现比较简单，最直观，也不需要额外的工具辅助修改。缺点是内存开销会比较大，内存占用是随着地图大小线性增长的。

放置路点

路点顾名思义就是路上的点，在地图上根据需求在所有需要寻路的区域放置路点。路点一般是需要人工选择位置放置的，不过不用担心，一般会有策划同学负责搞这个，233。

路点的总集合在 Lua 中一般可以使用一个 Table 来记录，每个路点包含了两份数据，自己的位置和从自己出发可以去的点，可达点还需要记录到它的路径开销，这个路径开销并不一定是简单的坐标距离计算，可以是考虑了地图点优先级的开销。

```
local waypoint = {
    a = {
        {1, 1},
        {{"b", 10}}
    },
    b = {
        {3, 5},
        {{"a", 10}, {"c", 5}}
    },
    c = {
        {12, 20},
        {{"b", 5}}
    }
}
```

路点寻路的缺点是点需要手工配置，经常会有一些点因为配的不太好，比如连通性配置有问题之类的，导致寻路的效果很奇怪，比如会莫名其妙绕路。而且因为对象一开始大概率不会正好在一个路点上，所以还要解决怎么从当前位置去到离得最近的路点这个问题。一般如果地图的路点不多的话，可以遍历路点，找到一个离自己近且中间无阻挡的路点作为起始

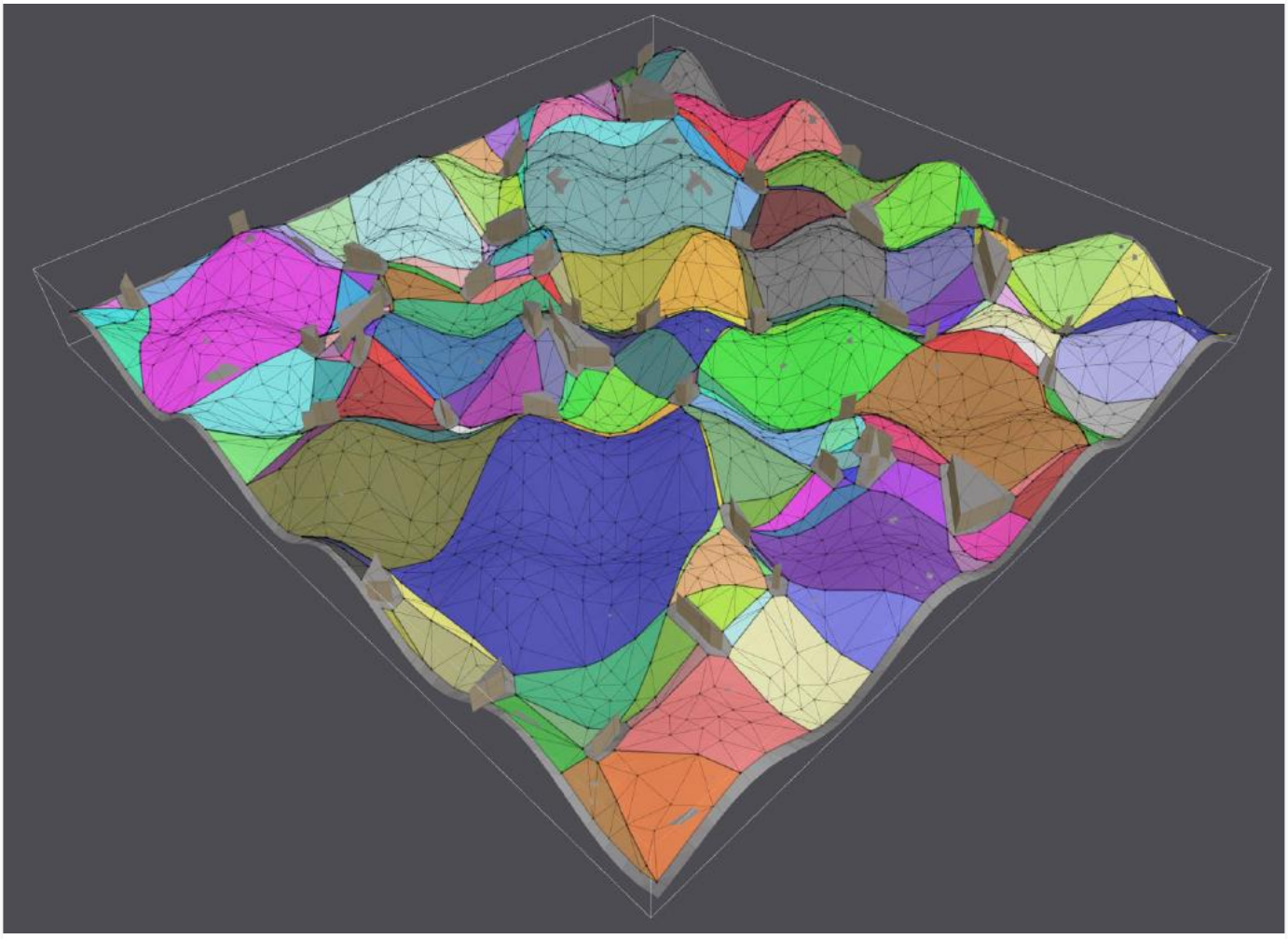
点，直线跑过去以后开始后续的寻路。

路点的优点是相比划分格子，因为路点的数量会少得多，所以它占用的内存会少，而且因为点少，所以它在后续配合算法寻路时效率也更高。还有一个优点是有时候策划想要对象在寻路时按照他配置的线路跑，比如在城市中寻路，有的策划喜欢让角色在路上跑时可以跑在路中间，而不是沿着建筑物边缘跑最更近的路程。

导航网格

导航网格是这几种里面最复杂的，使用了非常多的前端技术来处理地图。有一个非常广泛使用的开源库 `recastnavigation`，基本上是业界标杆水平。

导航网格的核心是将地图分为凸多边形网格（Poly Mesh），之所以这样做是因为凸多边形有一个特性，在它内部包括边缘任意取两个点连线都不出超出它本身的范围。Poly 是不带高度的，每个 Poly 会进一步划分成若干个三角形，每个三角形可以保证是一个平直面。划分的三角形可以保存下来用于寻路。整个处理过程有一篇文章写的很好，游戏的寻路 导航 1：导航网格。



recastnavigation

导航网格的优点是它可以很好的用在 3D 地图上。缺点是概念比较难以理解，这导致难以做一些需求定制，同时处理很大的地图时生成结果的过程非常慢。

寻路算法

虽然标题是寻路算法，但是现在才真正开始讲寻路算法。

寻路算法是需要基于上述几种寻路数据建模的结果来计算的。理论上任何一种寻路算法都可以跟上面的任何一种建模方式配合使用。

一些概念

为了更好的了解寻路算法，首先要简单说几个概念。

曼哈顿距离，是一种在大城市估算城市距离的方法，它假设开始点和结束点直接都是高楼大厦，无法对角线移动。它的计算很简单，在 2D 坐标轴中两个点的曼哈顿距离就是它们在 x 轴上的距离 + 它们在 y 轴上的距离。

欧几里得距离，就是两个点之间的真实距离，它不做任何移动方向上的限制，直接使用勾股定理计算两个点的直线距离。

启发式函数，用来进行一些寻路中需要参考的值的预估，在算法中使用不同的启发式函数，可以得到不同的寻路结果。

开放集合（OpenList），保存的是目前需要关注的点，一般是当前寻路点周围的点。在寻路计算中经常需要从开放集合中取出经过启发式函数计算的估值最小的点，所以开放集合一般使用有序的数据结构实现。

封闭集合（CloseList），保存的是所有已经被启发式函数计算过估值的点，当一个点在寻路计算中被处理完毕以后，它会被放在这个集合中。因为经常需要判断一个点是否在封闭集合中，所以它一般使用哈希表之类的数据结构实现。

贪心算法

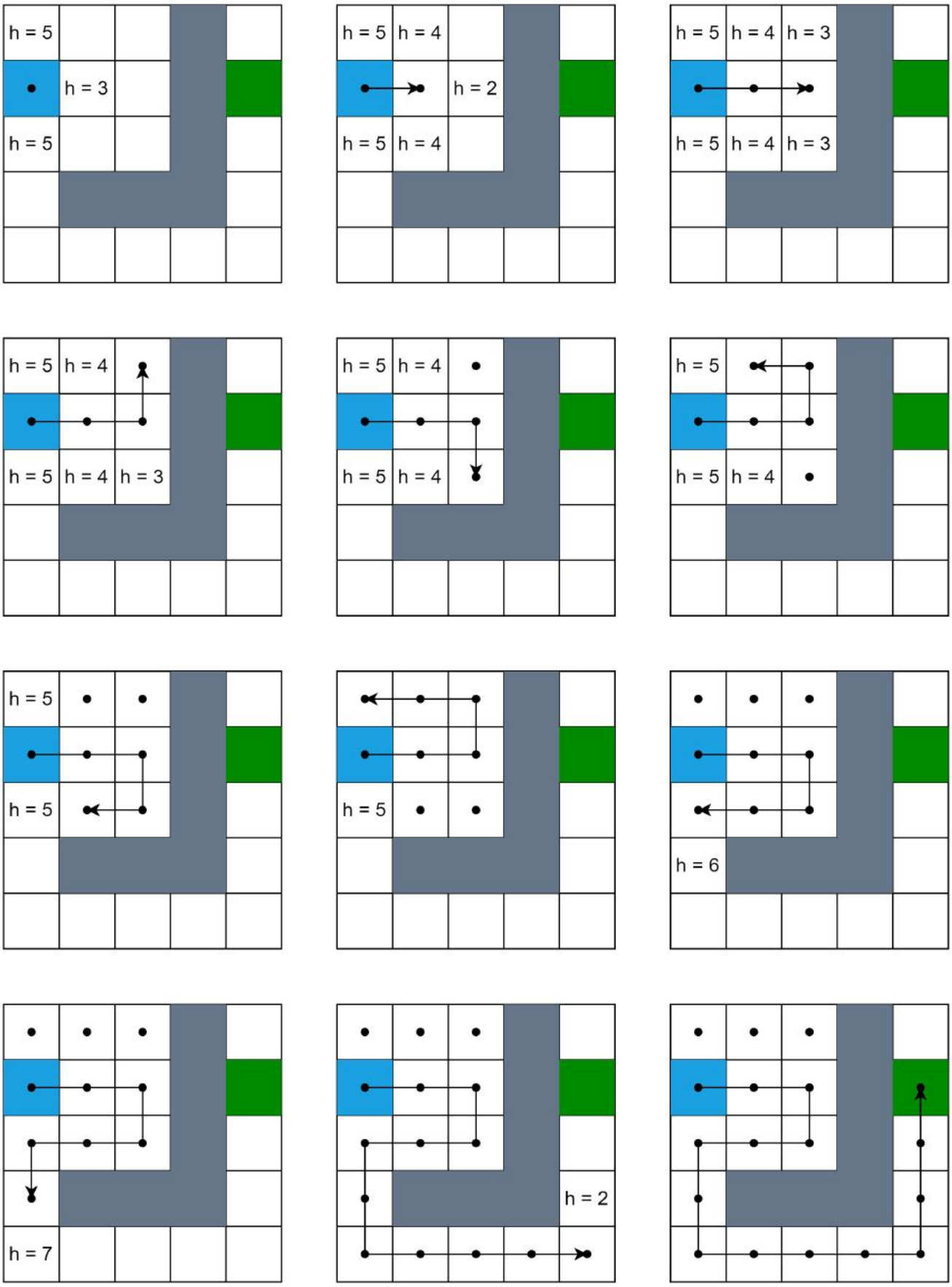
贪心两个字是指算法不做长期的的规划，而仅仅着眼于眼前。它每次选择下一步的点时，都直接选择待选点中通过启发式函数计算开销最低的点。如果在没有任何阻挡的情况下，那么这样是路径最短的，但是如果有了阻挡以后，这样做往往可能会选择出一条非最优路线。

贪心算法的步骤如下：

1. 首先将起点设为当前点，并且将它加入封闭集合中。

2．依次处理当前点的所有可达点中不在封闭集合里的点，将其前置点设为当前点，然后检查它是否在开放集合中，如果不在则计算其预估开销，并且将其加入开放集合中。
3．检查开放集合，如果为空，则地图搜索已经结束了，跳至步骤 5。
4．如果开放集合不为空，则取出其中预估开销最小一个点设为当前点，比较它的位置是否与目标终点一致，如果一致，则搜索结束，否则将它加入到封闭集合中，重复步骤 2。
5．当搜索结束时，如果当前点的位置不是目标终点，那么寻路失败。否则从当前点也就是终点开始一直找前置点，这条线路的逆序就是寻路的结果。

接下来看一个贪心算法的搜索步骤例子，例子使用曼哈顿距离作为期望开销，在图中使用中间有点的格子代表了该格子已经被加入到了封闭列表中，有路径数值的格子代表了当前在开放列表中的格子。


h = 5
h = 3
h = 5
h = 5
h = 4
h = 2
h = 5
h = 4
h = 5
h = 4
h = 3
h = 5
h = 4
h = 3
h = 5
h = 4
h = 5
h = 4
h = 3
h = 5
h = 4
h = 5
h = 4
h = 5
h = 4
h = 5
h = 5
h = 6
h = 7
h = 2

navi_greedy

可以看到贪心算法导致了从起点开始没有直接向下，而是一路向右，直到碰到了阻挡又绕了一圈过去，最终找到了目标点，但是没有得到最佳的路线。

A* 算法

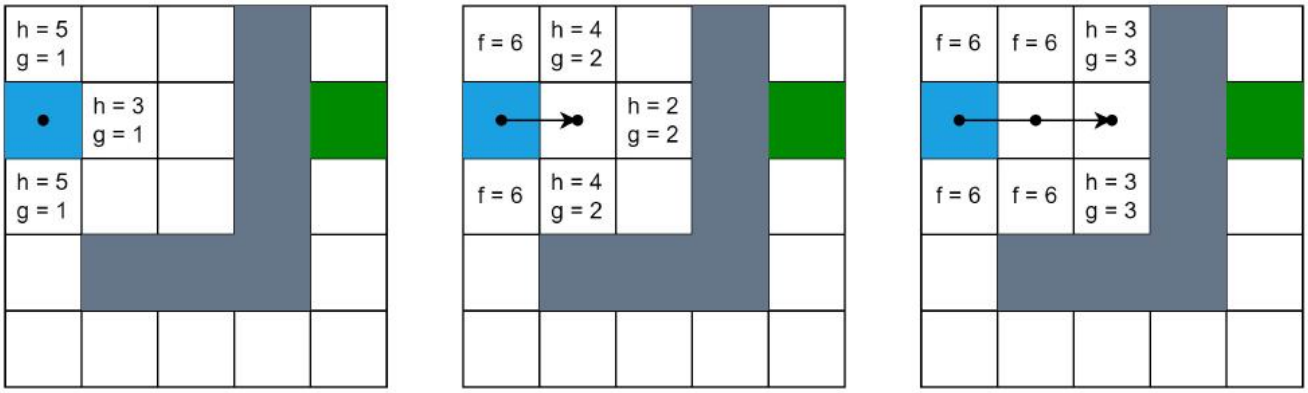
A* 算法应该是使用最广泛的寻路算法了，它兼顾了性能与路径质量，同时实现也比较简单，只比贪心复杂了一点点。

A* 算法的步骤如下：

1. 首先将起点设为当前点，并且将它加入封闭集合中。
2. 依次处理当前点的所有可达点中不在封闭集合里的点，检查其是否在开放集合中，如果不在则将其前置点设为当前点，计算其总开销，并且加入开放集合中。如果它本来就在开放集合中，则以当前点作为它的前置点再计算一次它的总开销，如果比之前的少，则将它的前置点改为当前点并且更新它的总开销。
3. 检查开放集合，如果为空，则地图搜索已经结束了，跳至步骤 5。
4. 如果开放集合不为空，则取出其中总开销最小的一个点设为当前点，比较它的位置是否与目标终点一致，如果一致，则搜索结束，否则将它加入到封闭集合中，重复步骤 2。
5. 当搜索结束时，如果当前点的位置不是目标终点，那么寻路失败。否则从当前点也就是终点开始一直找前置点，这条线路的逆序就是寻路的结果。

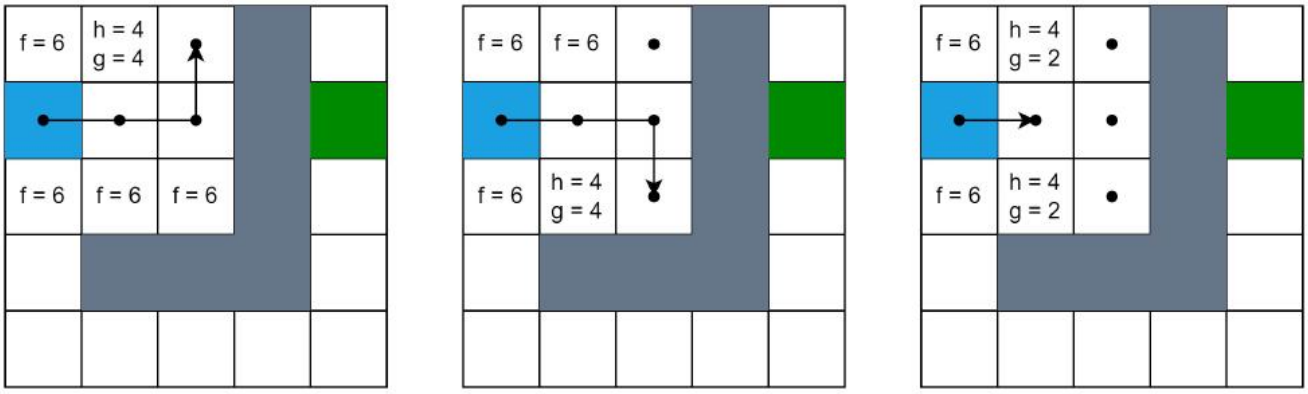
可以看到，A* 算法与贪心算法的区别都在步骤 2 里，主要有两点，首先 A* 算法中是使用的两个路径和来计算的，其次 A* 算法在处理已经在开放集合中的可达点时，会比较它当前的总开销与以自己为前置点的总开销，当以自己为前置点的总开销小的时候才会更新可达点的前置点。

接下来使用 A* 算法处理刚刚贪心算法处理过的那个问题，图中的 h 为预估开销，g 为前置路径开销，f 为总开销，也就是 h 和 g 的总和。当计算当前点的可达点时，会计算出以自己为前置点的开销，列出 h 和 g，不是当前点的可达点的点只列出它目前的开销和，方便比较。



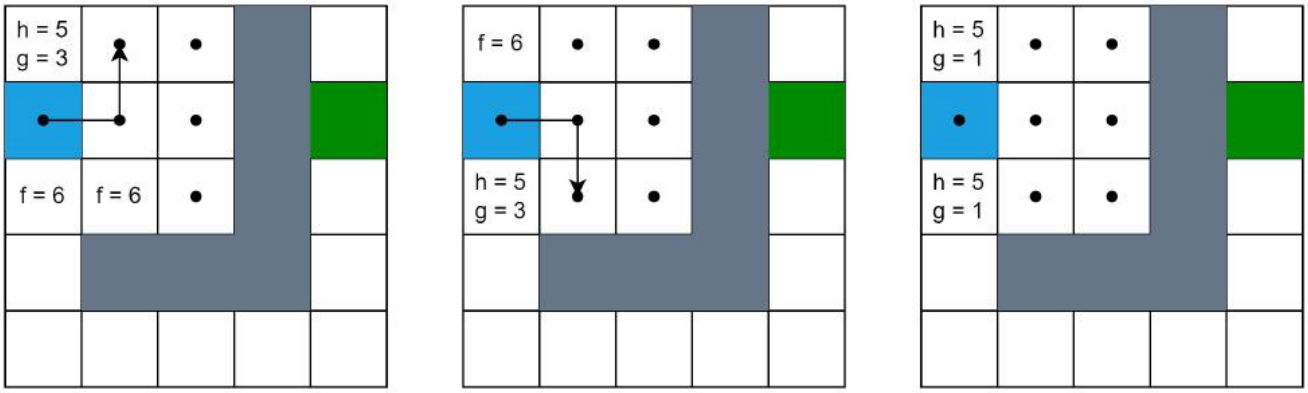
navi_a_star_1

可以看到 A* 寻发在一开始一样是一路向右边走，遇到阻挡以后，此时开放集合中有六个点，在第三张图里分成了三列，每一列上下两个点的父节点都是它们中间那个点，这时六个点的总开销都是 6。



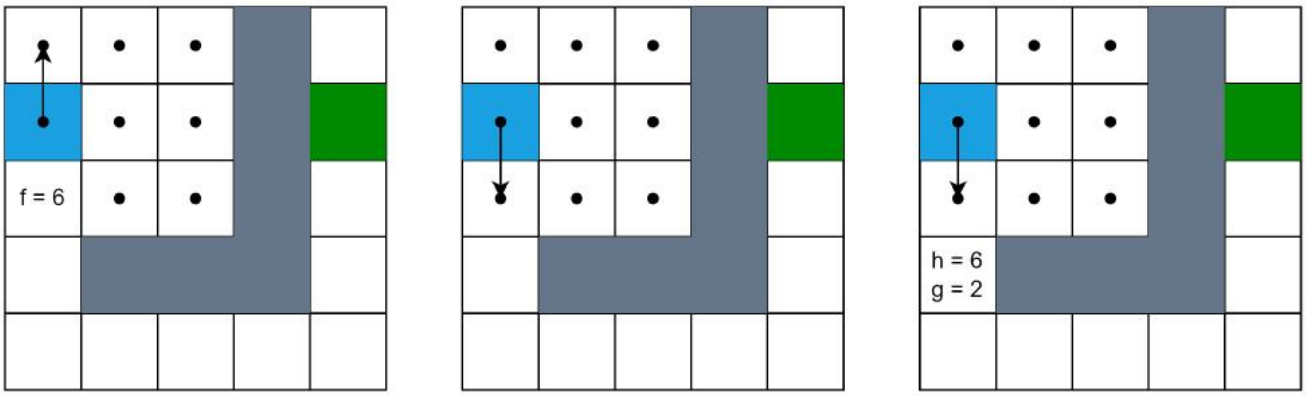
navi_a_star_2

因为开放集合里的点路径相等，所以随机取一个点作为当前点，最左边图中假设取到了右上角的点。可以看到它只有一个相邻点可以走并且不在封闭集合中，但是它从本点走过去的路径和是 8，大于其原来的 6，所以不做更新操作。中间的图假设取到了下面的点，也是一样的情况。现在开放集合中还剩下四个点，并且路径和都为 6。



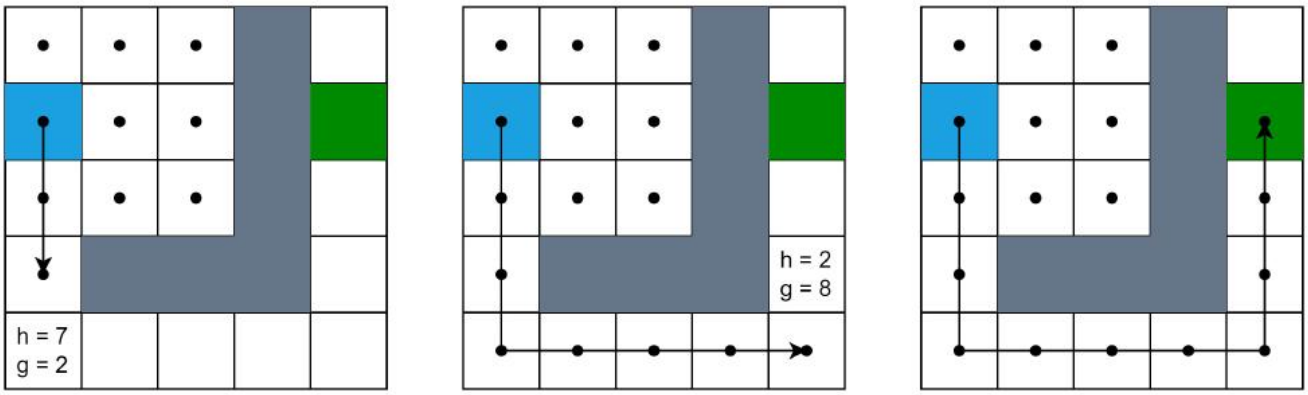
navi_a_star_3

这三张图处理的是开放集合中，处于中间列的两个点的情况，跟上面的情况一样，也找不到更短的路径，不做任何更新，只把自己加入到封闭集合中。



navi_a_star_4

继续看后续的操作，这三种图中，第一张图处理第一个点，它周围已经没有任何不在封闭集合中的点了，无法进行操作，直接把自己放入封闭集合中。然后处理第一列下面的那个点，将它更下面的点加入到开发集合中。



navi_a_star_5

接下来的事情就顺理成章了，开放集合中只有一个点，一直走下去就可以了。当前节点位置等于目标点时，说明寻路结束，反转链表即可得到寻路线路。可以看到 A* 最终搜索到了最优路线。

A* 算法被提出已经有五十多年了，至今已经有了诸多针对它的优化算法，但是它依然是使用最广泛的寻路算法。这主要归功于它的适用场景十分广泛，它的改进型都或多或少增加了一些场景限制，如果覆盖不到自己的适用场景，那效率再高也是镜花水月。

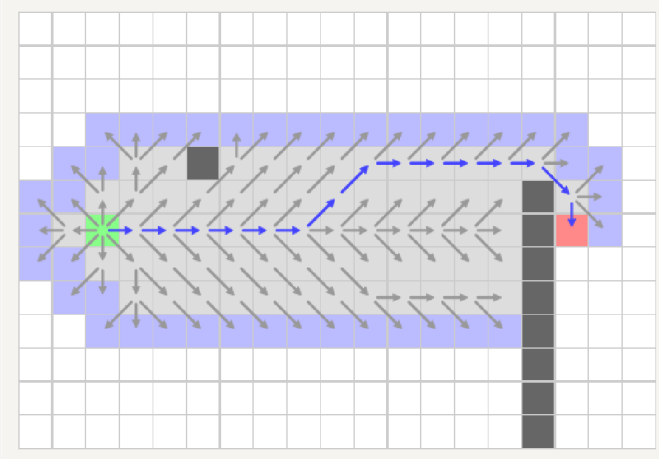
关于 A* 算法有两篇非常好的文章，`Introduction to the A* Algorithm` 讲述了 A* 算法的概念。`Implementation of A*` 给出了 A* 算法的几种实现。

JPS 算法

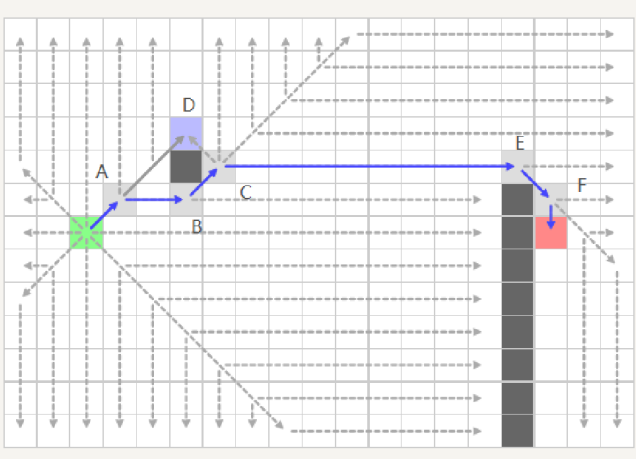
JPS (Jump Point Search) 算法，有时候也翻译成跳点算法。它是 A* 的一种改进方法，实现了效率的大幅度提升。相较于它的老前辈们，JPS 算法的历史非常短，提出至今不过十年左右的时间，但是因为它出色的效率，现在已经基本上成为了 A* 改进型的标杆算法了。

但是因为算法特性问题，它的使用场景比较受限，它只支持规则格子地图上的可用和阻挡两种情况，无法处理更复杂的地形，比如地图上某些地块有更高的优先级这种情况，而且它需要两个点之间可以对角线行走。

JPS 进行的优化在于，它根据当前地图上的状况，把一些点给排除了，不需要加入到开放集合中。开放集合中的点少了，需要计算的次数自然就少了。而可以加入到开放集合中的点被称为跳点，这也就是跳点算法名字的由来。



navi_a_star



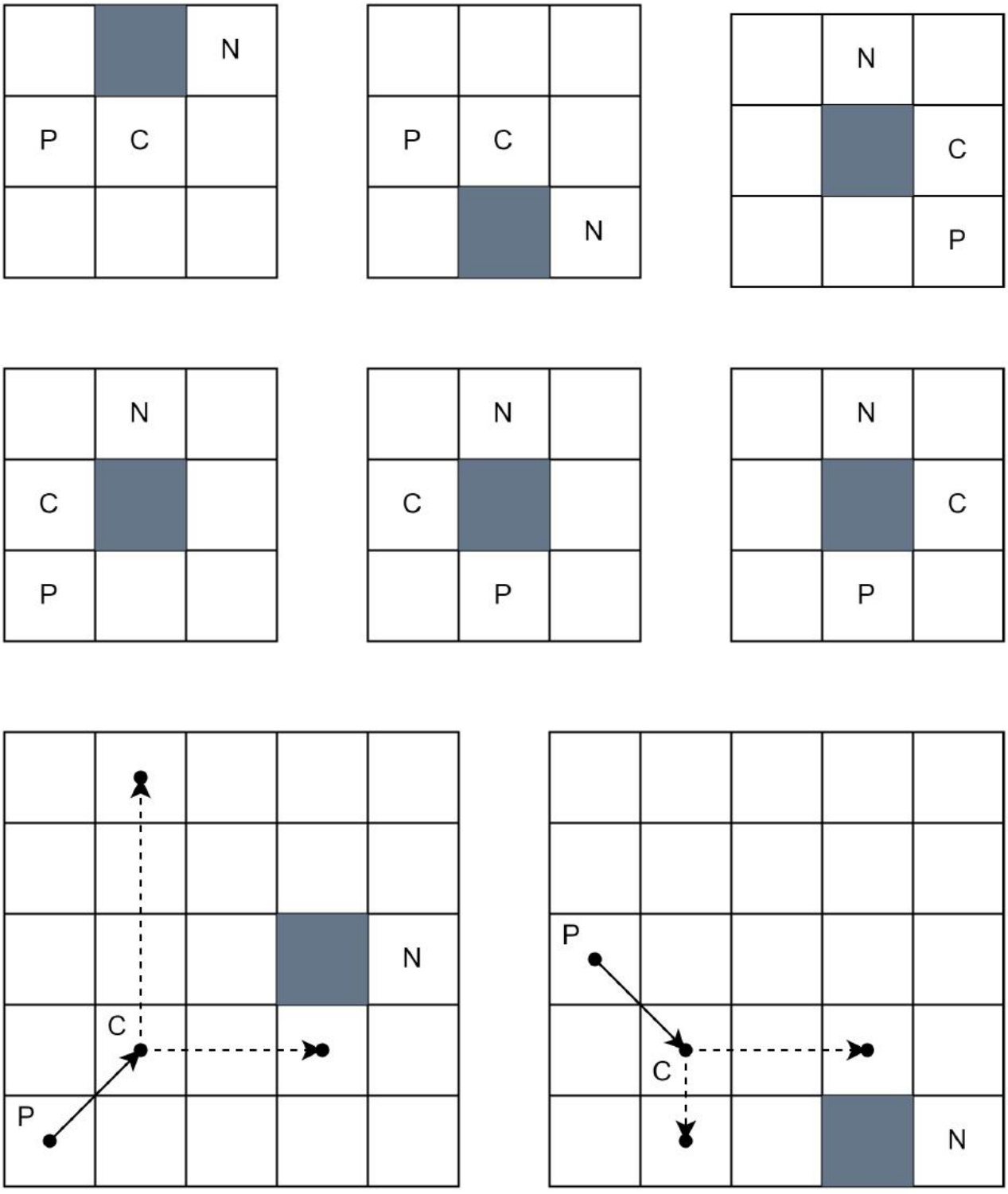
navi_jps

以上两个路径图使用 `A Visual Explanation of Jump Point Search` 页面中的程序实现，图中灰色的点是已经在封闭集合中的，蓝色的点是在开放集合中的。这篇文章不仅

有可自定义的寻路小程序，还对 JPS 算法有非常详细的解释，**强烈推荐阅读**。

在这个例子中可以看到，两个算法找到的路径虽然不一样，但是总开销是一样的，都可以认为是最优解。JPS 处理过的点比 A* 要多，但是绝大部分点因为不符合跳点规则，所以都被忽略了，除开起点和终点，寻路过程中只有六个点被加入到开放列表中处理过，而 A* 在本例中有几十个点都被加入过开放列表，多了大量的运算。

在解释 JPS 的步骤之前，需要明确两个它的独有概念，跳点和强制邻居点。虽然说是两个概念，但是其实是两个强关联的概念。如下图是几种强制邻居点的情况，P 是前置点，C 是当前节点，N 是强制邻居点。如果 N 是 C 的邻居，且 N 的邻居中有阻挡点，并且从 P 到 N 的最短路径一定要过 C，则 N 就是 C 的强制邻居点，而 C 就是跳点。如果前置点到当前点是斜向的话，那么当前点两个分量方向上延申的点如果有强制邻居点的话，那么当前点也是跳点。



navi_jps_point

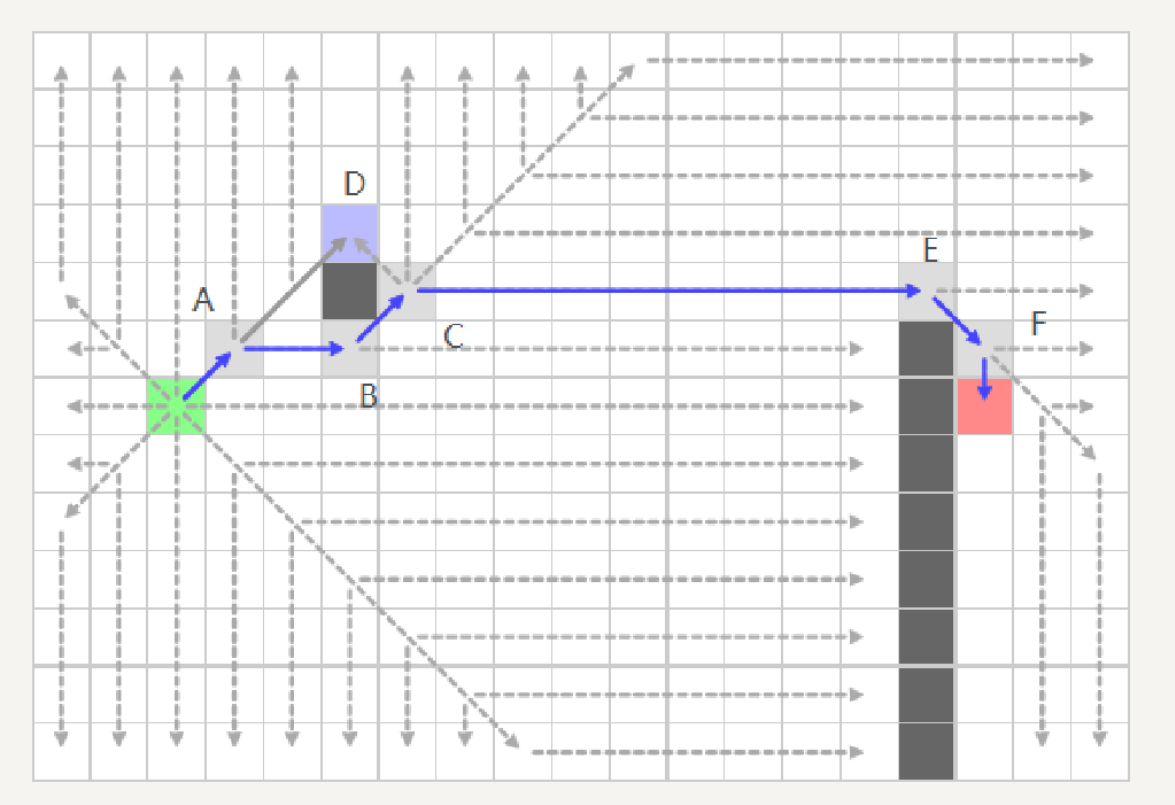
JPS 算法的步骤要复杂一些，大概如下：

1．首先将起点设为当前点，并且将它加入封闭集合中。

2．从当前点开始，检查当前点是否有前置点，如果没有前置点的，说明是起始点，起始点的扩散方向为周围全部的 8 个方向。如果不是起始点的，则会首先判断自己有没有

强制邻居点，如果有的话，将强制邻居点加入到开放列表中，然后根据前置点和自己的位置判断自己所在的方向，如果自己是在平直方向上，则沿着该方向搜索，如果是在斜角方向上，则先向斜角两个向量方向搜索，然后再沿着斜角搜索。

3. 在延展搜索的过程中，如果碰到了地图边界或者是阻挡，则停止这个方向上的搜索。对搜索到的每一个点检查它是否有强制邻居点，如果有，则它就是跳点，停止当前的搜索，并且将该点加入到开放列表中。
4. 检查开放集合，如果为空，则地图搜索已经结束了，跳至步骤 6。
5. 如果开放集合不为空，则取出其中总开销最小的一个点设为当前点，比较它的位置是否与目标终点一致，如果一致，则搜索结束，否则将它加入到封闭集合中，然后跳回步骤 2。
6. 当搜索结束时，如果当前点的位置不是目标终点，那么寻路失败。否则从当前点也就是终点开始一直找前置点，这条线路的逆序就是寻路的结果。



navi_jps

再回过头来复盘一下这个图的寻路过程。首先从起点开始向 8 个方向搜索，搜索右上角到 A 点时，从 A 点向上边和右边扩散，当扩散到 B 点时，发现 B 有强制邻居 C 点，说明 A 是跳点，将 A 的前置点设为起始点并且加入开放列表中，停止继续搜索。

从开放列表中拿取一个开销最小的点也就是 A，根据 A 的前置点，得到它的方向是向右上的。向上会碰到边界，向右到达 B 点的时候，可以发现 B 有一个强制邻居点 C，说明 B 是一个跳点，将 B 的前置点设为 A，然后将 B 加入到开放列表中。同时斜角方向发现

了跳点 D，将其前置点设为 A 并且加入到开放列表中。

从开放列表中拿取一个开销最小的点也就是 B，检查它周围，检查到有一个强制邻居点 C，将 C 的前置点设为 B，B 因为 A 的方向的关系，所以它会沿右方处理，并不会找到新的跳点。

从开放列表中拿取一个开销最小的点也就是 C，检查它的周围，发现一个强制邻居点 D，但是 D 已经在开放列表中了，所以不再处理它。C 因为前置点 B 的方向关系，它会向右上搜索。可以找到 E 有一个强制邻居点 F。将 E 的前置点设为 C，并且将它加入开放列表中。

从开放列表中拿取一个开销最小的点也就是 E，检查它的周围，发现一个强制邻居点 F，将 F 的前置点设为 E，并且将它加入到开放列表中。E 向右扩展，没有新的发现。

从开放列表中拿取一个开销最小的点也就是 F，检查它的周围，发现终点，将终点的前置点设为 F，从终点开始逆序，可以得到 Start → A → B → C → E → F → End 的路线。

总结

寻路算法是一个发展了很久很久的方向，除了上面列出的这三种比较有代表性的算法以外，还有非常非常多的其它算法，不过思路大部分比较类似，都是在前人的基础上做的一些改进。

在游戏开发中寻路是个偏前端的问题，除了 SLG 游戏以外，寻路一般都是在前端实现的，后端只会拿前端得到的路径判断一下点的合法性就行了。但是寻路算法的复杂性还并不是这些算法的理解，而是要搞定策划的需求，很多时候高效简洁的实现可能无法满足策划的想法，要在算法的基础上做各种贴近业务的定制化修改。

寻路

相关文章

浅谈游戏开发中的浮点数与定点数

游戏开发中的存储系统设计

分析游戏开发中 A 现思路