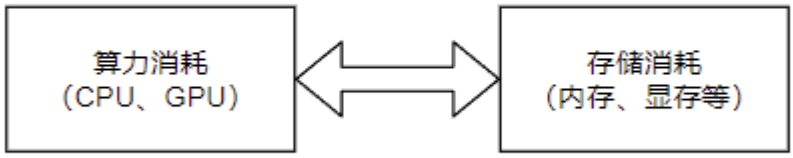
前言

算法是一个极其复杂的综合知识体系，包含计算机思想、数学思想，在漫长的游戏开发职业生涯中，或多或少的会接触到算法，本人在职业生涯中，越来越感受到一些算法的魅力以及重要性，因此花了一些时间，写出这篇博客，作为一些浅显的总结和归纳。由于算法知识体系庞大，即使是游戏开发领域的算法，也无法只通过一篇博客就概况清楚，因此整篇博客无法做到面面俱到，希望能对被算法困扰的游戏开发提供帮助，帮助大家对算法有个基础认识。本人算法水平有限，若有错漏之处，还望不吝赐教。

空间换时间、时间换空间

在算法中，常常会提到“空间换时间”与“时间换空间”，大致的意思就是指通过特别的手段来降低时间复杂度、空间复杂度，而代价就是稍微牺牲一点空间或时间。如下图所示：



空间换时间

空间换时间的例子随处可见，例如：

- 离线生成Shadow、Light;
- 对象池;
- 甚至是GetComponent也可以通过变量存储引用，防止再次取的消耗;

以上这些优化手段大多都是利用存储的方式，把CPU或GPU的运算节省一些。

时间换空间

现代计算机的发展，有个特点：CPU、GPU的运算力的增长速度不如存储设备的增长速度快。 或许存储设备的扩展只是在主机机箱中腾出一个足够大的位置，另外存储设备元件体积也越做越小。 因此对于多数中小型游戏来说，存储空间更加廉价，游戏客户端相对在空间中就有了更小的压力，可以相对放肆的使用。 当然也存在部分低端机，存储空间完全不够，所以往往也需要一些手段也优化。

在早些年，很多游戏，例如《超级马里奥》。当时的游戏卡带及其的小，所以多游戏开发也压榨到了极致，游戏内有许多资源是通过翻转变色，从而重复利用。游戏内的音乐也是几个基础音节按照特定数据描述，周期的播放，也就有了背景音乐。

例如 《我的世界》 大世界生成，采取的方法是程序化生成世界，得出的随机数据，转化为世界的构建。这使得游戏包体在硬盘中只占据极小的空间。

常规算法

在聊游戏算法之前，这里先介绍接触比较多的几个算法：动态规划、分治、贪心、回溯算法。这几个算法是比较有代表性的算法，能针对性的解决特定类型的问题。这也算的上是复杂算法的基础思想，通过基础思想的组合，去解决一个个的复杂问题。

动态规划

用一个不太恰当的公式来表示动态规划：

当前子问题/状态的解 = 某个子问题/状态的决策 + 当前决策方式。

动态规划大体上划分了两种思路：自顶向下与自底向上。 可以概况为从两种角度出发去解决问题。

动态规划

自顶向下

当我们要求解 问题n 时，已知 问题n的求解是在问题n-1基础上的扩展，也就是说我们向下探索，先求问题n-1，以此类推，我们向下探索到了问题1。

这里以斐波拉契数列为例，求解第n项。 已知：

- 第n项 = 第n-1项 + 第n-2项 （n>2）
- n = 1时, f(n) = 1;
- n = 2时，f(n) = 1;

自顶向下可以这样求解，如下代码：

```
public static int Fibonacci(int n)
{
        if(n<=0)
            return n;
        int []Memo=new int[n+1];
        for(int i=0;i<=n;i++)
            Memo[i]=-1;
        return fib(n, Memo);
}

public static int fib(int n,int []Memo)
{

        if(Memo[n]!=-1)
            return Memo[n];
    //如果已经求出了fib（n）的值直接返回，否则将求出的值保存在Memo备忘录中。
        if(n<=2)
            Memo[n]=1;

        else Memo[n]=fib( n-1,Memo)+fib(n-2,Memo);

        return Memo[n];
}
```

自底向上

当我们要求解 问题n 时，我们以最初起始问题开始 问题1 -> 问题2 -> ... 问题n-1 -> 问题n。我们先求解子问题，再最后扩展到最终的目标问题。

同样，以斐波拉契数列为例，求解第n项。代码如下：

```
public static int fib(int n)
{
        if(n<=0)
            return n;
        int []Memo=new int[n+1];
        Memo[0]=0;
        Memo[1]=1;
        for(int i=2;i<=n;i++)
        {
            Memo[i]=Memo[i-1]+Memo[i-2];
        }
        return Memo[n];
}
```

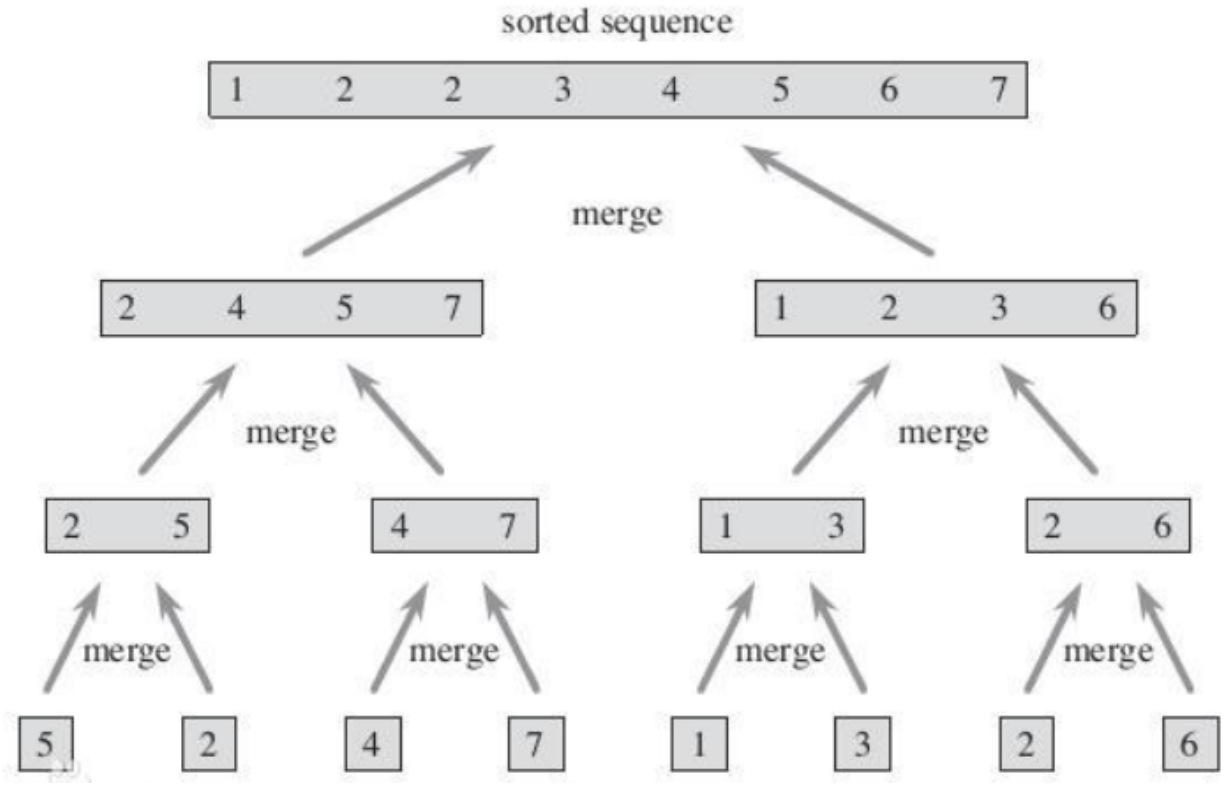
分治

分治算法是将一个相对复杂的问题，分解为相似的更小的子问题，而最终的子问题是相对可以更简单的求解。

分治算法与动态规划的区别是，分治的子问题相对独立，没有直接关联，不直接影响各自的决策。

注意：只有子问题的解可以合并，才能选择用分治。

归并排序，把原本复杂的问题，拆成一个个子问题，最后把子问题的解合并，得到最终的解，如下图：



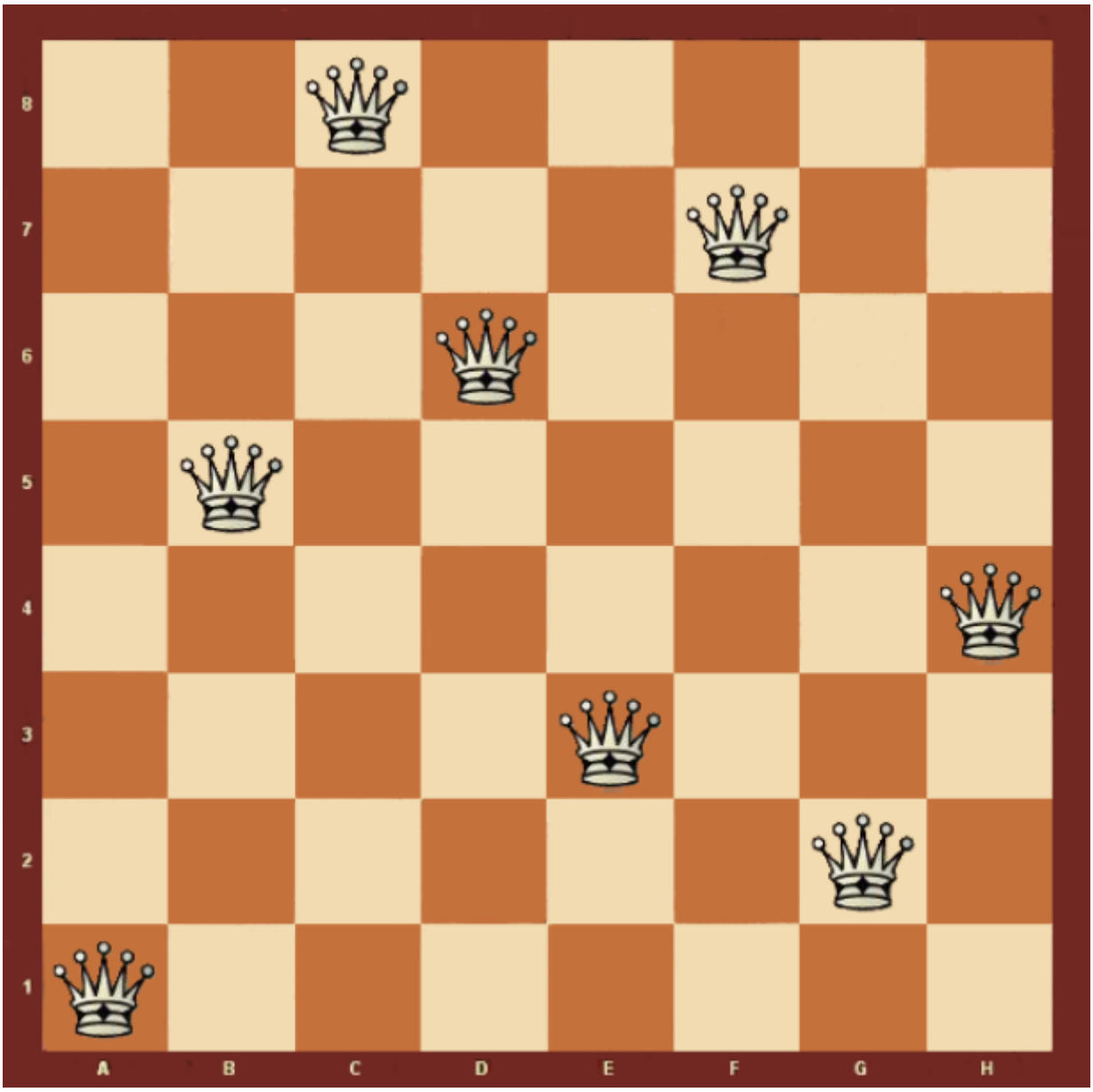
分治

回溯

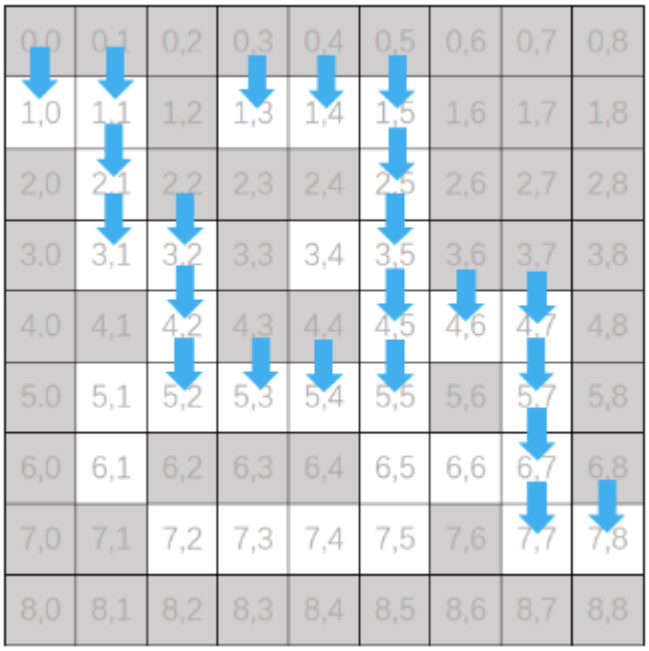
回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。 回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

例如迷宫、数独等游戏，都可以通过回溯算法来解。当然也不是唯一解决办法。

n 皇后问题，如下图，八皇后：


8
7
6
5
4
3
2
1
A
B
C
D
E
F
G
H

迷宫问题，如下图：


0,0 0,1 0,2 0,3 0,4 0,5 0,6 0,7 0,8
1,0 1,1 1,2 1,3 1,4 1,5 1,6 1,7 1,8
2,0 2,1 2,2 2,3 2,4 2,5 2,6 2,7 2,8
3,0 3,1 3,2 3,3 3,4 3,5 3,6 3,7 3,8
4,0 4,1 4,2 4,3 4,4 4,5 4,6 4,7 4,8
5,0 5,1 5,2 5,3 5,4 5,5 5,6 5,7 5,8
6,0 6,1 6,2 6,3 6,4 6,5 6,6 6,7 6,8
7,0 7,1 7,2 7,3 7,4 7,5 7,6 7,7 7,8
8,0 8,1 8,2 8,3 8,4 8,5 8,6 8,7 8,8

数独问题，如下图：

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

回溯

贪心

贪心算法强调局部最优解的选择，通过局部最优来达到全局最优。如上面提到的，如任意步骤的选择，不影响下一步骤的可选策略范围，那么意味做，当前步骤做任何决策，下一步的决策范围也是一样的，这就代表每一步选择最优，全局就应当是整体最优。只有这样的贪心算法才是效率最高的。

例如：有一个骰子，投6次，什么情况下，6次的点数之和最大。 可以这么理解：每一次投掷，出现点数6时，就是局部最优解，所以按照贪心算法的思路，如果每一回合都是局部最优，那么意味着每一回合都是点数6。最后点数 6*6 就是全局最优解。

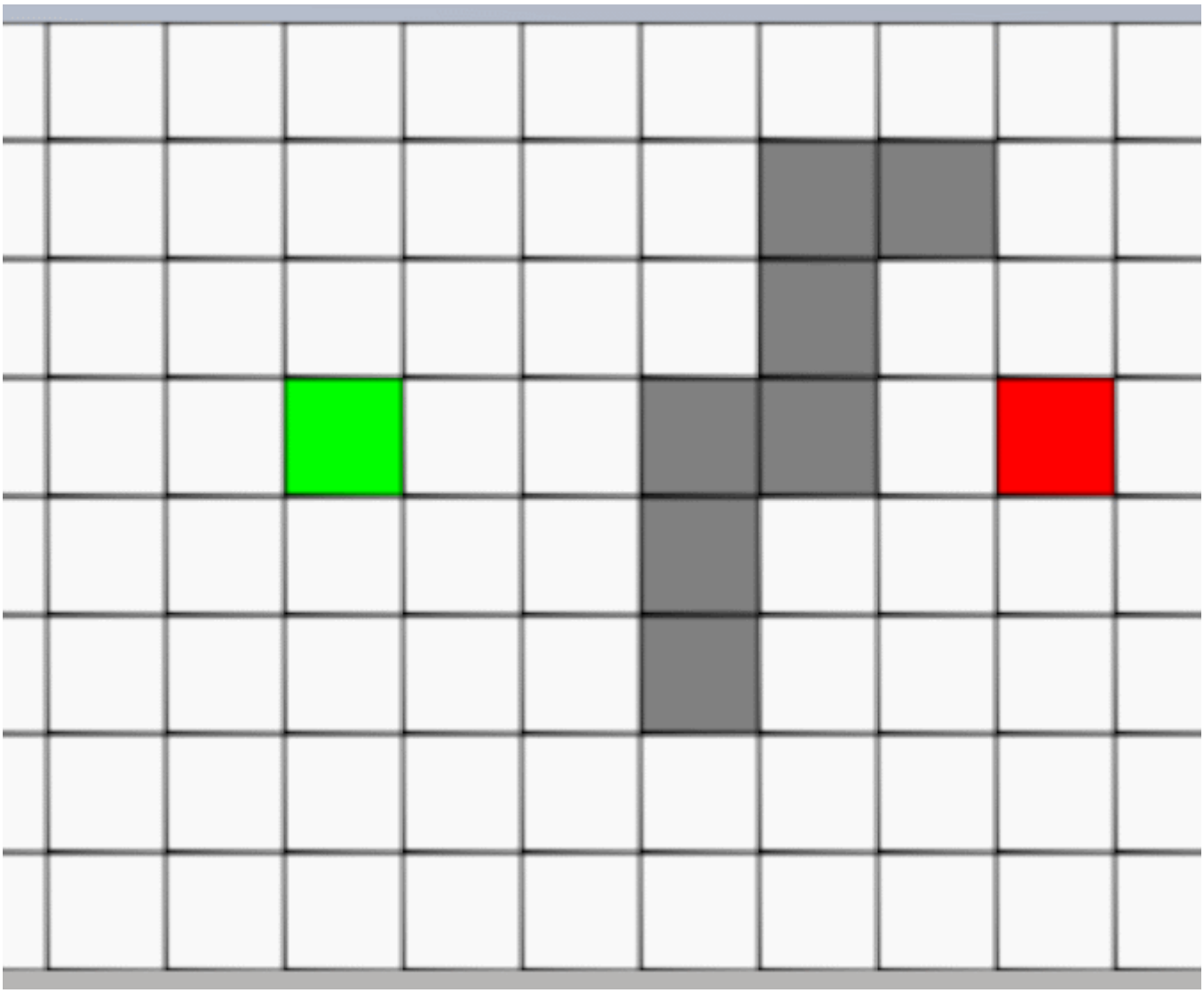
贪心

寻路算法

在很多导航需求，AI移动需求方面，我们常常需要用上寻路算法，业内常规的寻路算法有：A星、JPS等等。

A-Star

A*（A-Star)算法是一种静态路网中求解最短路有效的方法。公式表示为：f(n)=g(n)+h(n)，其中f(n)是节点n从初始点到目标点的估价函数，g(n)是在状态空间中从初始节点到n节点的实际代价，h(n)是从n到目标节点最佳路径的估计代价。网上能找到不少优秀的文章，这里不再过多总结。

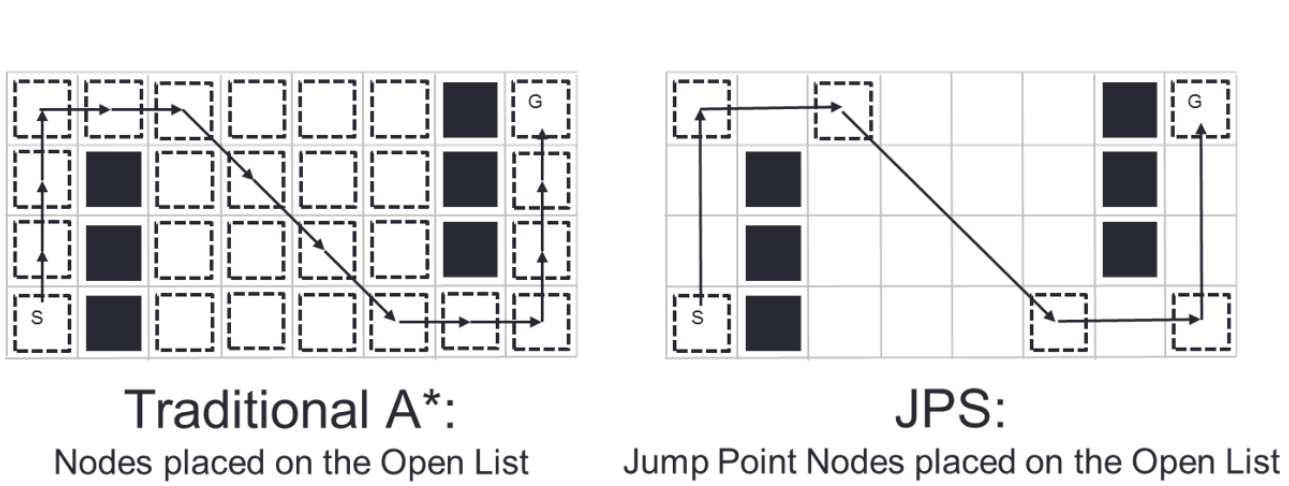


以前写的A星：https://zhuanlan.zhihu.com/p/80707067

当图结构特别庞大时，需要检索的节点会特别多，导致树的广度特别大，这是因为OpenList内太多节点需要进行比较，这时我们需要减少OpenList内的节点。

JPS(跳点寻路算法)

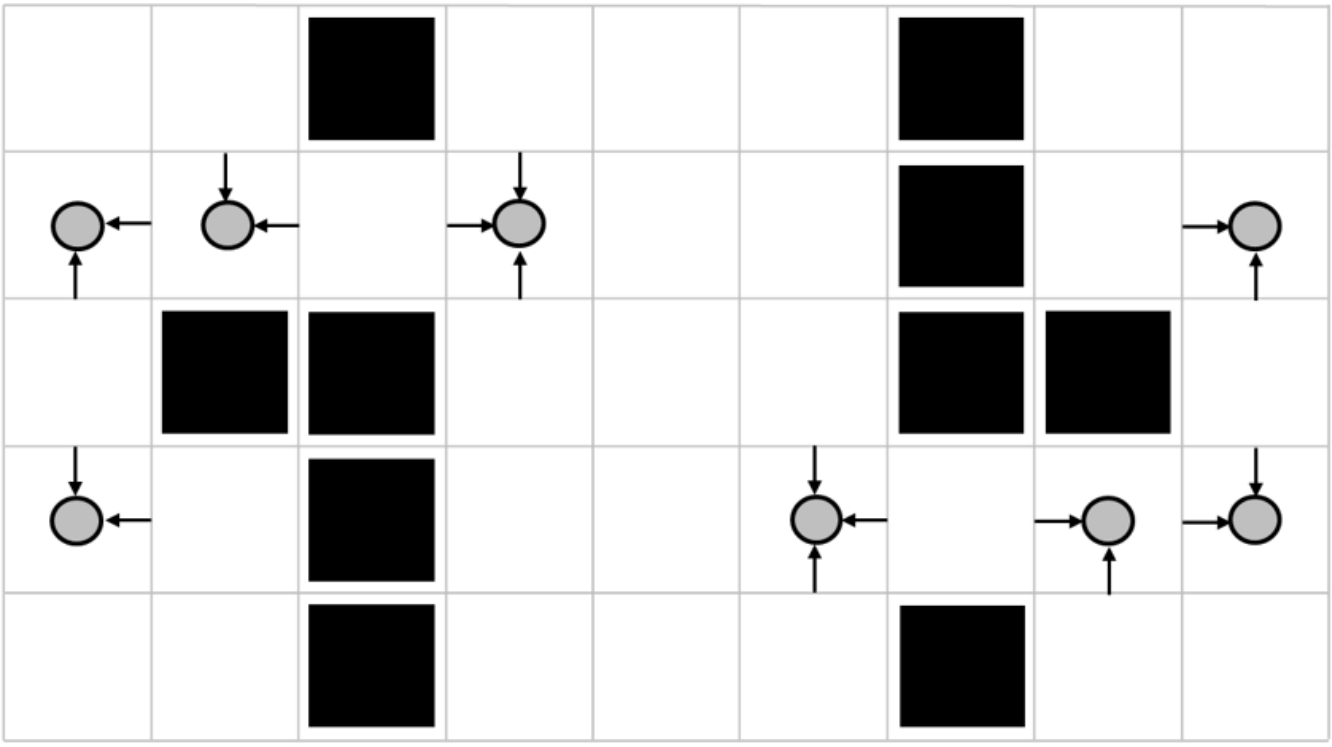
当我们了解了A星之后，我们注意到OpenList过大一定会导致节点的比较次数增大，另外我们发现 Grid 类型的世界，很多节点是可以不用考虑到寻路内的。



JPS 预处理

注意，预处理只能应用于静态地图，因为只有静态地图才是一种确定性的搜索。

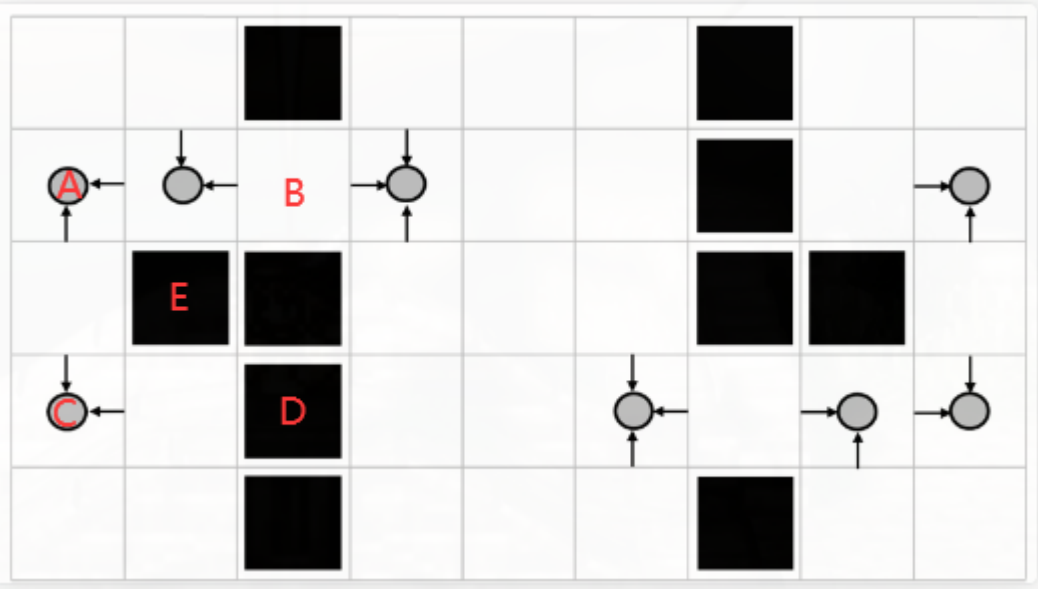
情况一，对角的连通判断，例如每个障碍物，存在四个对角节点，若节点满足：节点非障碍物，并且有连通性。如图：



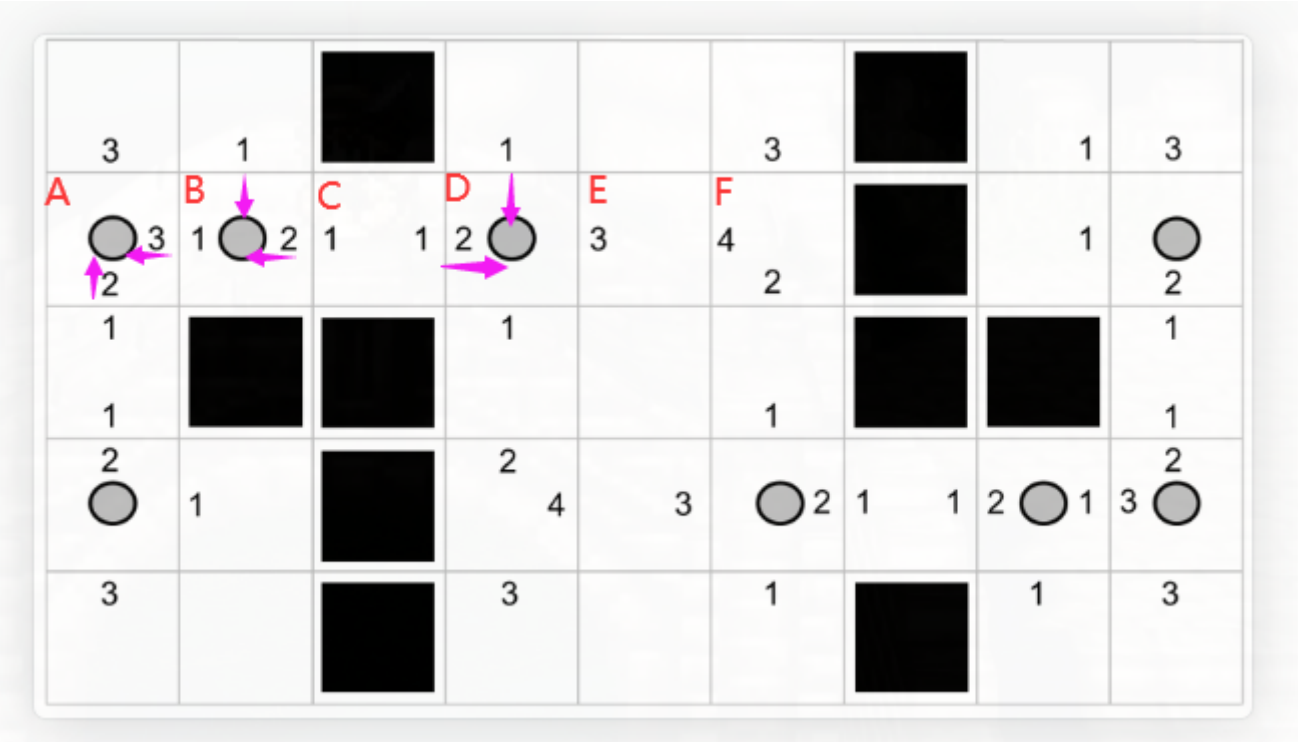
解读如下：例如 E 是障碍物，那么 E 存在四个对角：A、B、C、D。

如下图，分析情况：

- D 节点是障碍物，所以 D不是跳点。
- B 节点没有连通性，B 可以从西到东，但是B不能从北到南，因为B的南向是障碍物，所以B不是跳点。
- A 节点不是障碍物，并且可以从南到北，可以从东到西，有连通性，所以是跳点。
- C 节点不是障碍物，并且可以从北到南，可以从东到西，有连通性，所以是跳点。



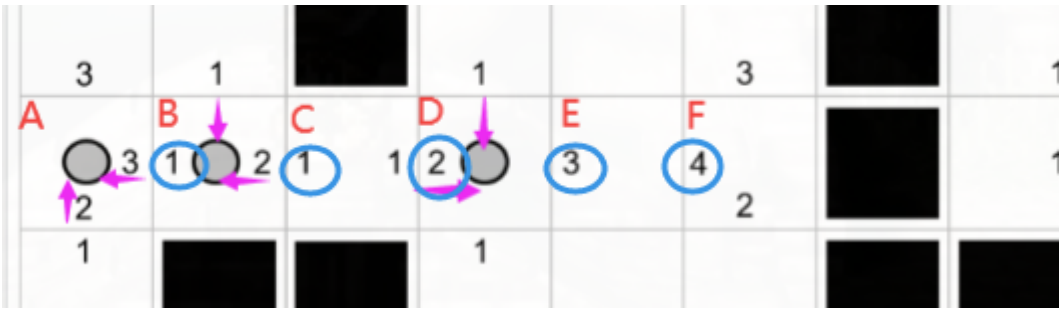
直线方向上的判断。我们先从简单的情况开始，我们从左到右检查，按照ABCDEF的顺序检索下去。如下图所示：



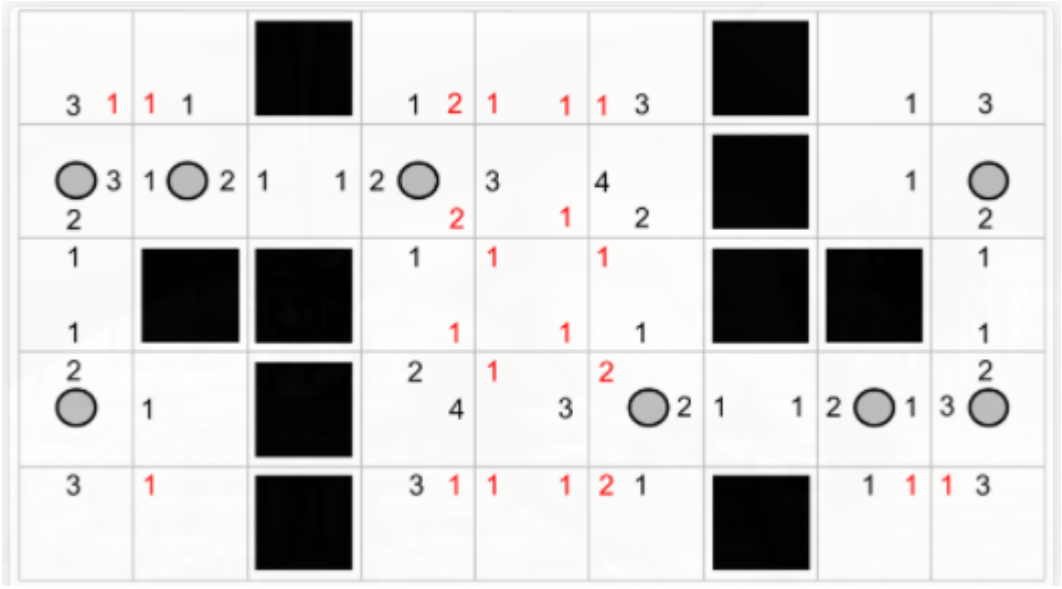
A节点的强制邻居是右和下，所以我们画上的红色箭头。然后依次给B和D也画上。

如下图，我们来分析ABCDEF的向西（West）方向的值，这个值其实就是标注特别的跳点距离。注意是特别的跳点。

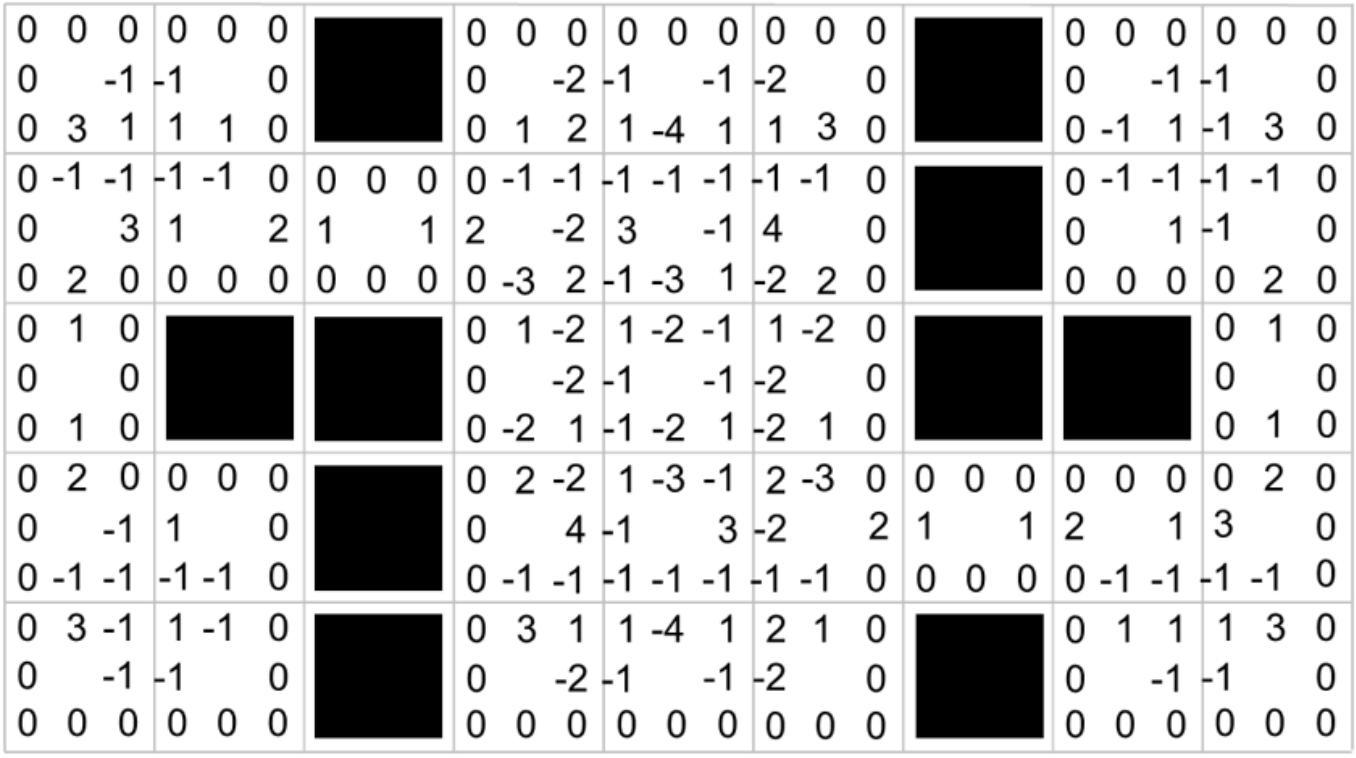
- B.West = 1，因为A.East 是满足强制邻居条件的，B和A的距离是1，所以B.West = 1
- C.West = 1, 因为B.East 也满足条件，C和B的距离是1
- D.West = 2, 因为B比A更近，距离为2
- E.West = 3, 同上理由
- E.West = 4, 同上理由



接着处理四个斜线方向。



然后对于剩余节点处理，记录距离，如下：

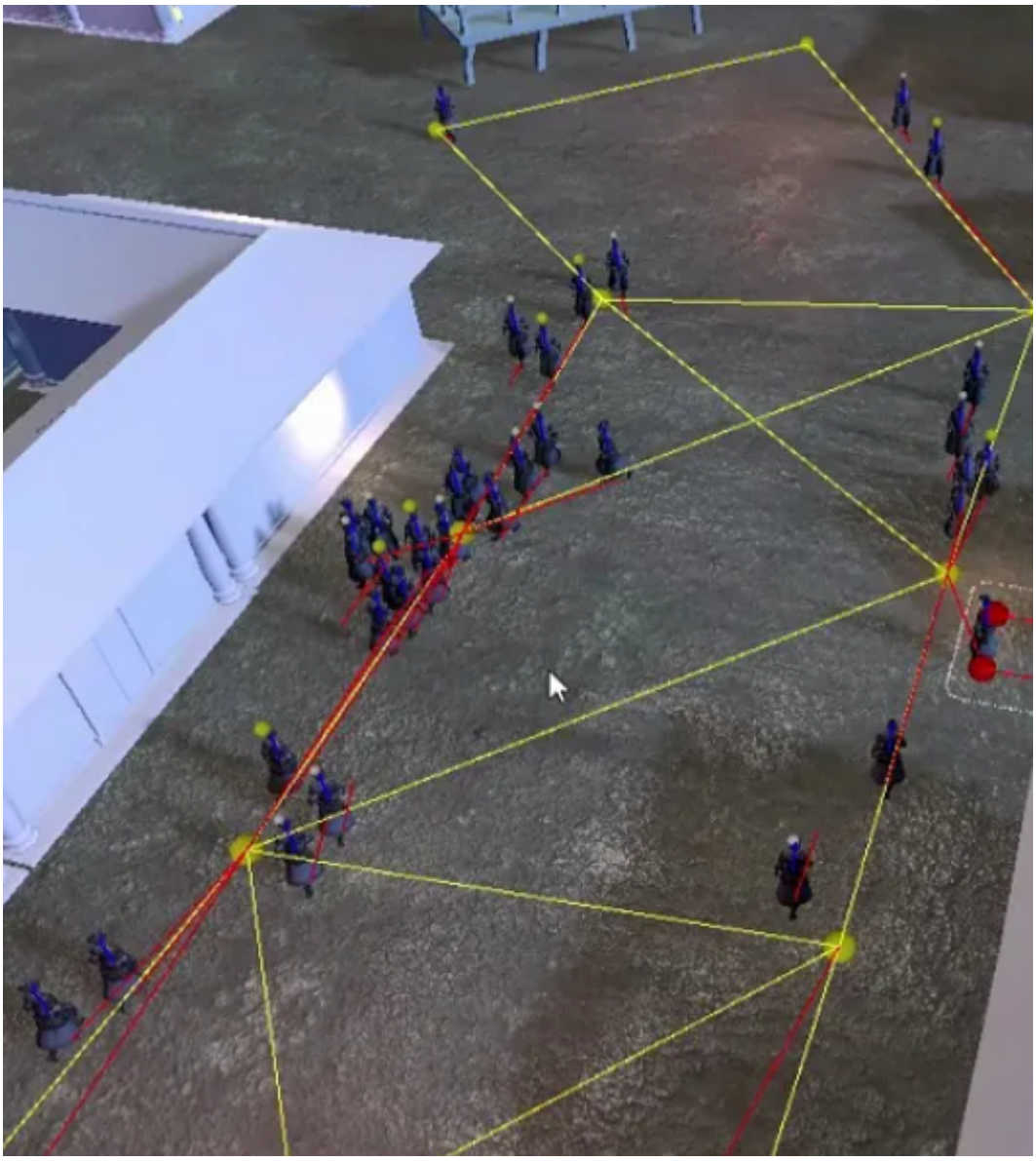


篇幅有限，可以查看其它人写的源码： https://github.com/trgrote/JPS-Unity

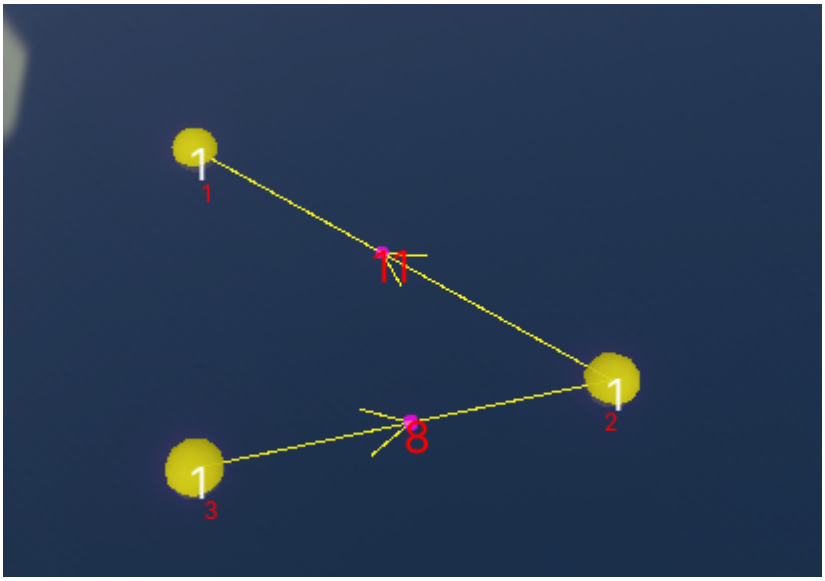
WayPoint（路点寻路）

并不是所有的游戏世界都能用 Grid 来描述的。特别对于一些偏向写实的游戏，更加不可能把整个世界按照大量小格子来做划分，其一，描述的过于简单，不适合描述更加复杂结构。其二，格子的尺寸很影响最终的格子数量，一旦地图过大，格子数量就特别庞大，对于超远距离的寻路一样是不友好的，即使是跳点寻路。

为此，这里采用另外一种寻路方式解决这类问题，路点寻路。如下图，通过Gizmos绘制运行时的寻路情况：



点与线对象构成图结构，此处截图表示一种单向图，因此 Gizmos 绘制了箭头，如下图所示：



整个技术方案如下描述：

- 提供一种Editor下的编辑工具，可以创建节点，可以任意选择两个节点是否相连，如果相连就产生了线，所产生的线可以自定义代价（这里的代价就是寻路的代价，默认情况下，代价越小，寻路越优）。并且产生的线可以选择是否单向或者双向（这意味着是否只是单向的寻路，或者双向的寻路，可以产生人行到靠右边的特殊效果，下面会提到）。最后把编辑好的图数据导出。
- 运行时通过图数据构建寻路数据，然后进行点到点的寻路，得出最近路径。

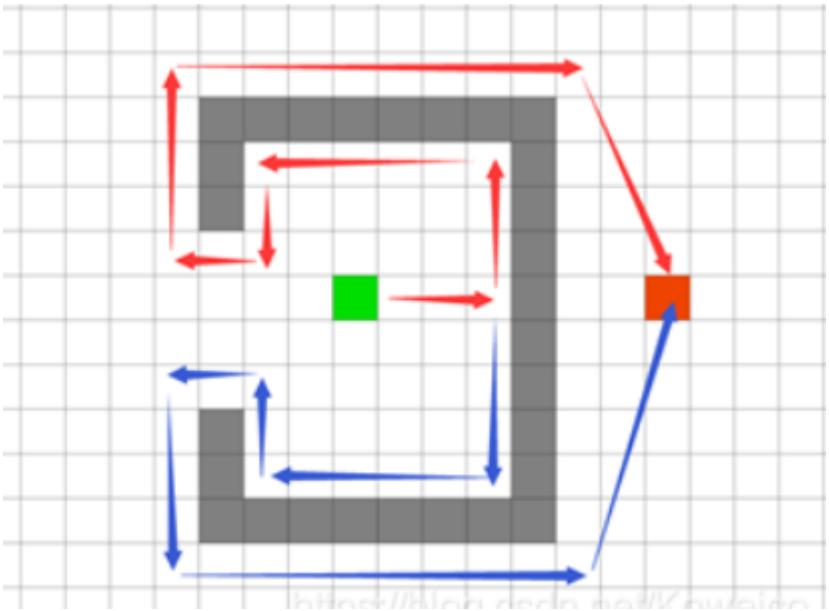
这种方式的寻路，还是有不少好处，对策划同学特别友好。

实现人群靠右行走：https://www.zhihu.com/zvideo/1339900170771976192

几年前，写过一遍粗略的总结：https://zhuanlan.zhihu.com/p/194888259

B-Star

B-Star， BranchStar 即一种分支寻路算法。B-Star 的关键是产生分支，而产生分支的时机就是碰到障碍物（或者说碰到墙）。如下图所示：



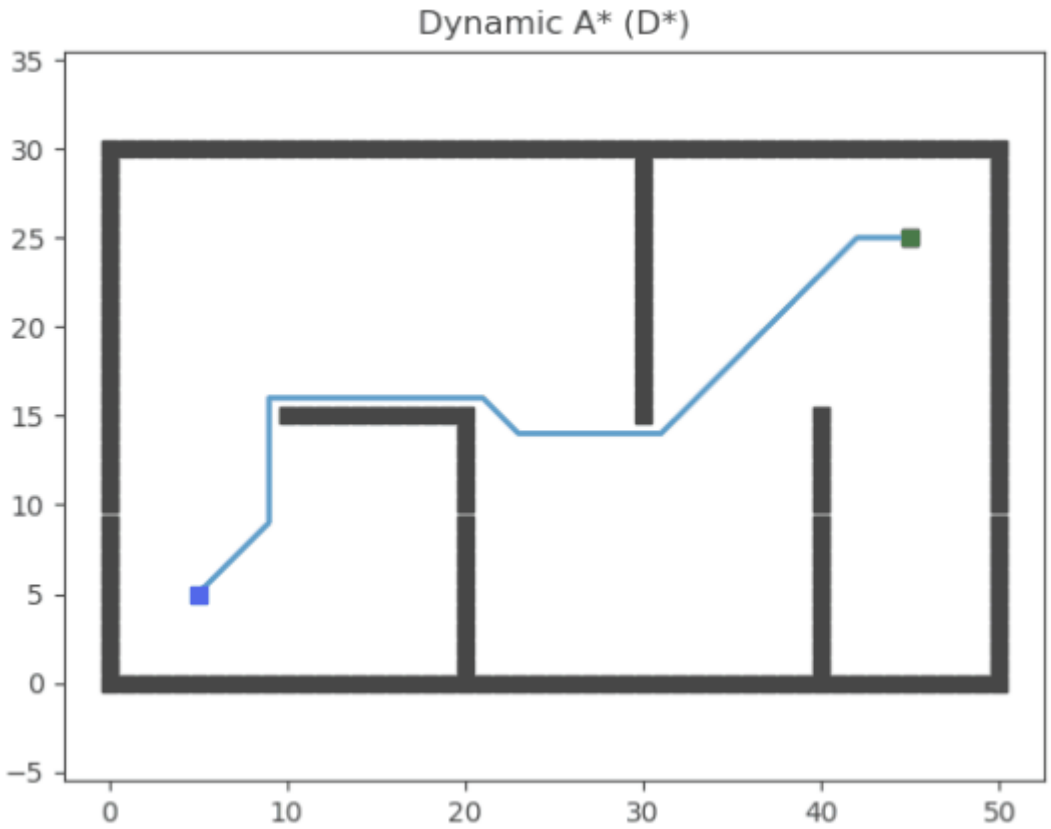
这是一种消耗较低，但是不保证绝对最优，所搜索出来的路径是保证可达终点，但是路径代价几乎高于最优路径很多。这种算法据说是从自然界的现象观察出来的，例如：大量蚂蚁进行指向性的觅食（即从起点出发时，大致知道终点方向）。

算法大致步骤如下描述：

- 1.从起点出发时，按照终点的大致方向移动。
- 2.判断移动的前方是否是障碍，如果有墙就进行分支寻找（即变成两条寻找路径，前提是可以分支时，特殊死角不能分支除外）。
- 3.根据上一步，可能产生了n个分支，n个分支继续寻路，直到找到终点。

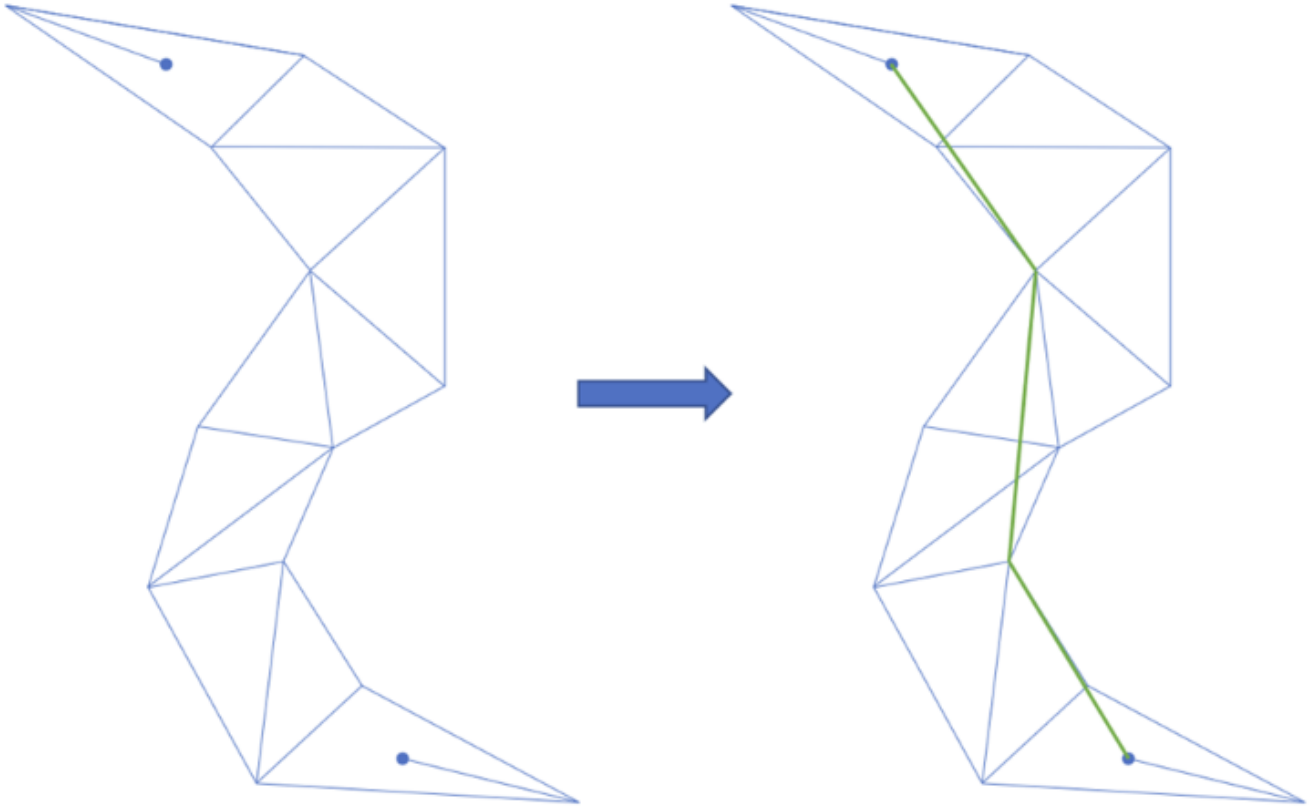
D-Star

D-Star 即 Dynamic A Star，同样要维护一个OpenList，但是与 A Star 不同的是，D-Star 是从终点开始搜索。D-Star 比较适合对动态变化的环境寻路，这样能最小程度的进行再次的小范围寻路，从而修改小范围的路径。



漏斗算法

在Unity 中，引擎提供了一种 Nav-Mesh 的数据，用来描述地图数据，划分可走区域与不可走区域，然后对于需要寻路的对象，挂载 NavAgent，赋予对象寻路能力。其中比较有意思的是 Nav-Mesh 是多个三角形组成的 Mesh，这意味着它的寻路过程是一个三角形到另一个三角形的搜索。这和我们上面提到的 Grid 地图时间是有差异的。



算法的大致流程：

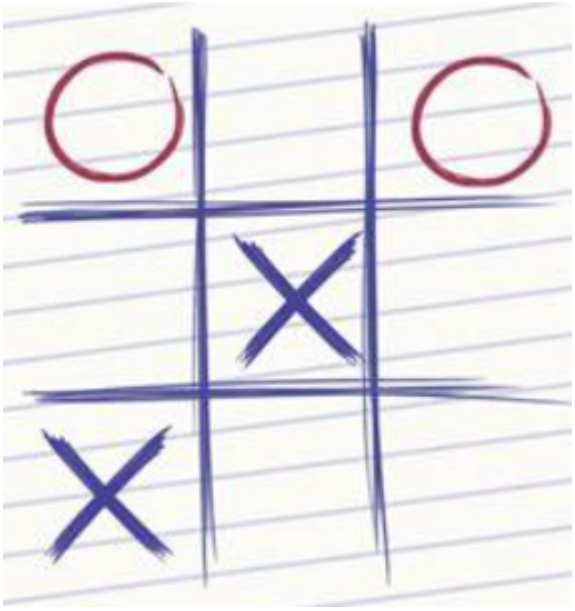
- 起点和终点必然包含在三角形中（否则无法到达终点）
- 找到三角形的共用边。
- 从起点开始，从起点分别连接上共用边的亮点，形成第一个漏斗
- 然后根据漏斗的收敛方向，把漏斗的夹角缩小，直到不能再缩小，漏斗起点推进到下一个点
- 接着从新构建新漏斗，重复上面的步骤，直到到达终点所处的三角形。表示找到终点。路径生成。

更加详细的过程，可以查看这篇博客：https://www.cnblogs.com/pointer-smq/p/11332897.html

博弈论

博弈论又被称为对策论（Game Theory），是现代数学的一个新分支，也是运筹学的一个重要学科。在很多棋牌类游戏中，我们时常遇到这类理论的运用。例如象棋游戏，我们希望在单机模式下，依然有AI与我们共同竞技，为此AI需要具备思考的能力，对于AI所做出的决策也一定是经过运算，从运算结果分析利弊，从而得出结论"这一步棋应该这么下"。

这里先从简单的棋类开始，分析棋类该怎么做出一个有一定思考能力的AI。我们选择用“井字棋”。



井字棋特点：

- 步骤有限，双方总共9步
- 存在和棋

关于井字棋的博弈分享，在3年前写过一篇文章分享：https://zhuanlan.zhihu.com/p/65219446

预测收益

首先我们得告知 AI，游戏规则。游戏规则如下：

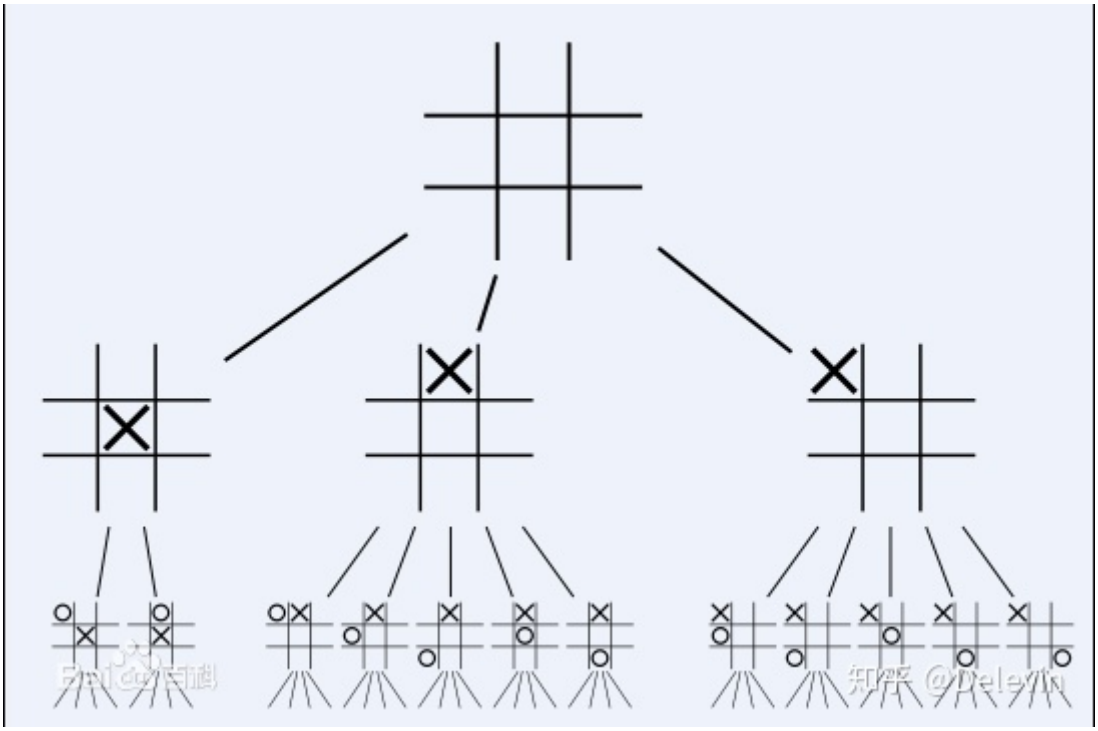
- 回合制
- 每回合轮流各下一子
- 若有一方棋子先同行同列同对角线三子，那么胜出
- 若棋盘下满后，无人胜出，则和棋

当AI知道规则后，我们还需要让AI知道：怎么让AI去判断下子的收益。为此我们需要去定义一些收益的例子，例如：当自己两个棋子已经同行同列或者同对角线（并且没有被围堵）。这种情况下属于接近胜利，我们告诉AI这种时候收益很高。

显然 “收益” 的判断很重要，所以要有准确的 “收益” 制定，让AI知道做哪些事情可以带来收益，最后引导AI走向胜利。 除了 “收益” 外，我们还需要制定另外一种AI思考能力，那就是 “代价” 。

“代价” 可以认为是一种负 “收益”。之所以要提出这个，是因为AI在做出任何决策后，人类都会采取自己的对策，以AI视角来说，人类的“收益”就是AI的“代价”。同理，AI的“收益”就是人类的“代价”。

AI 通过计算当前的可走步子，权衡 “代价” 与 "收益"。可以画出如下的图：



注：由于对称性，所以第一轮的决策，原本有9种选择，被缩减成了三种。

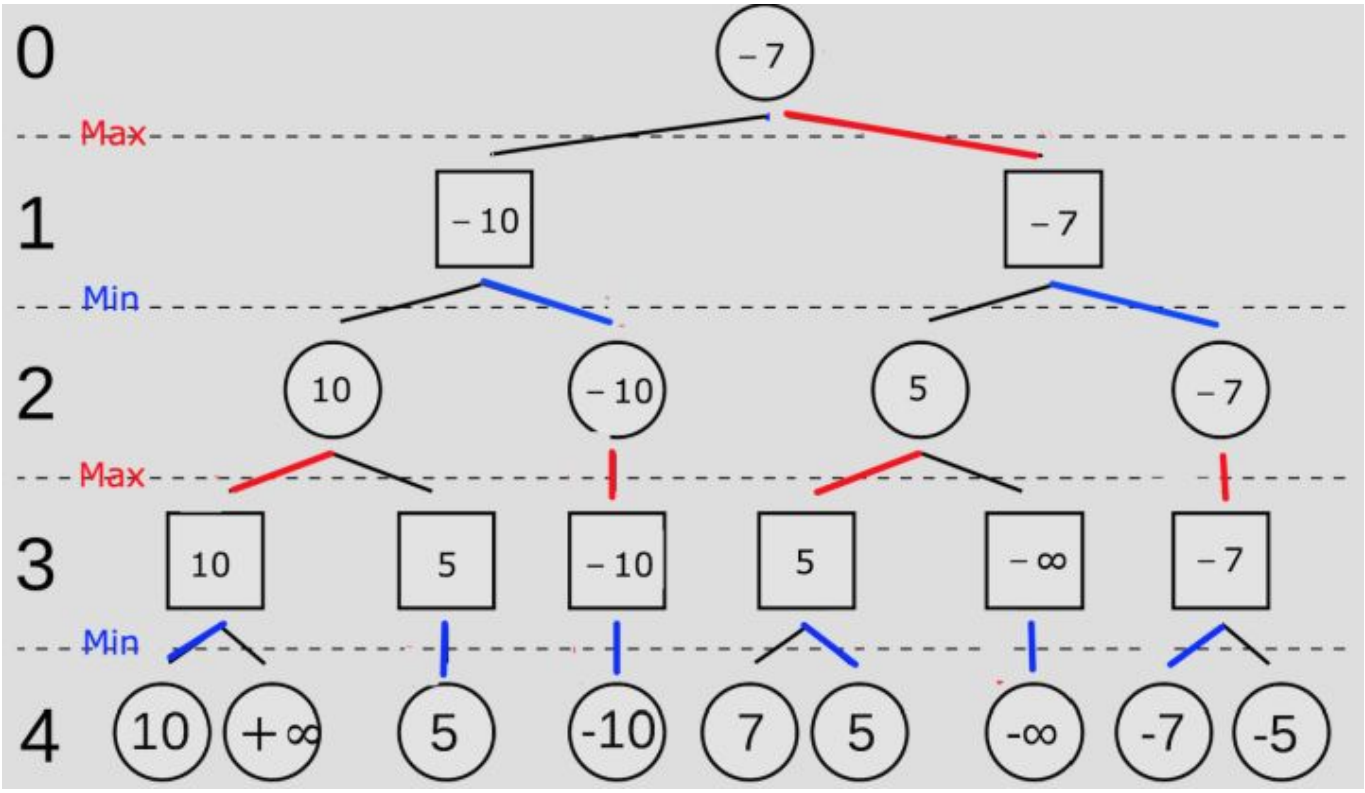
当AI足够聪明时，它总是希望自己利益最大化的，那么这里就要引出一个问题：每一步利益最大化，是否会最终利益最大化。

其实并不会这样，并不是每一步利益最大化，就一定可以最终利益最大化。因为每一步的决策都会导致下一步的环境不一样，这意味着下一步的利益最大化只是针对某种情况而言，也就是说不同决策，后续的利益也无法直接对比，因此局部最优无法表达全局最优的情况。

这里可以列举寻路算法，如果我们选择贪心算法，每一步选择代价最低的步骤，也无法保证是代价最小的最短路径，因为当前步骤的最优，也会导致后续环境的不一样，这也就论证了上面的观点：局部最优无法保证全局最优。

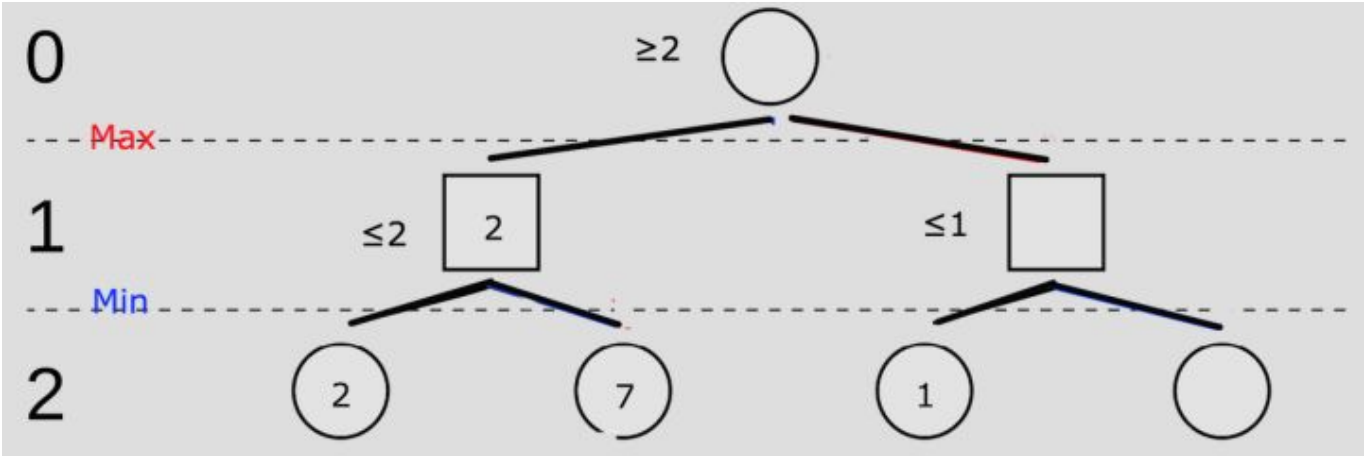
注：只有当每一步决策后，所出状况也会完全相同，那么这种状况下，每一步最优一定可以得到全局最优。

我们把自己的收益当作最大值，自己的代价当作最小值。 当我们能评估最大值和最小值时，得出如下图：



剪枝算法

在已知博弈过程中，对弈双方优先选择对自己有利的策略，那么提前预测对方的上限与下限就可以提前知道有些决策是可以不考虑的，也就可以剪枝了。



几何检测算法

三年前，写过一篇基于OpenGL实现的2D碰撞类小游戏： https://zhuanlan.zhihu.com/p/82301894

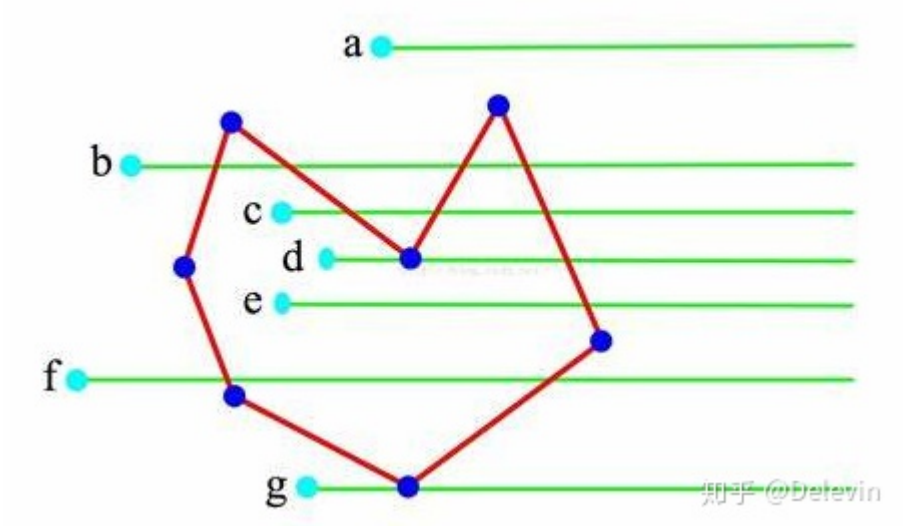
2D 碰撞

三年前在知乎上，总结过一篇多边形碰撞检测： https://zhuanlan.zhihu.com/p/86981378

这里简单提一下。

点与多边形

通过从检测点发射出一道射线，产生交点（如不产生交点，即为0交点）的数量可以作为点是否在多边形内的判断依据，若奇数个点，即在多边形内。若偶数个点，即在多边形外。如下图：

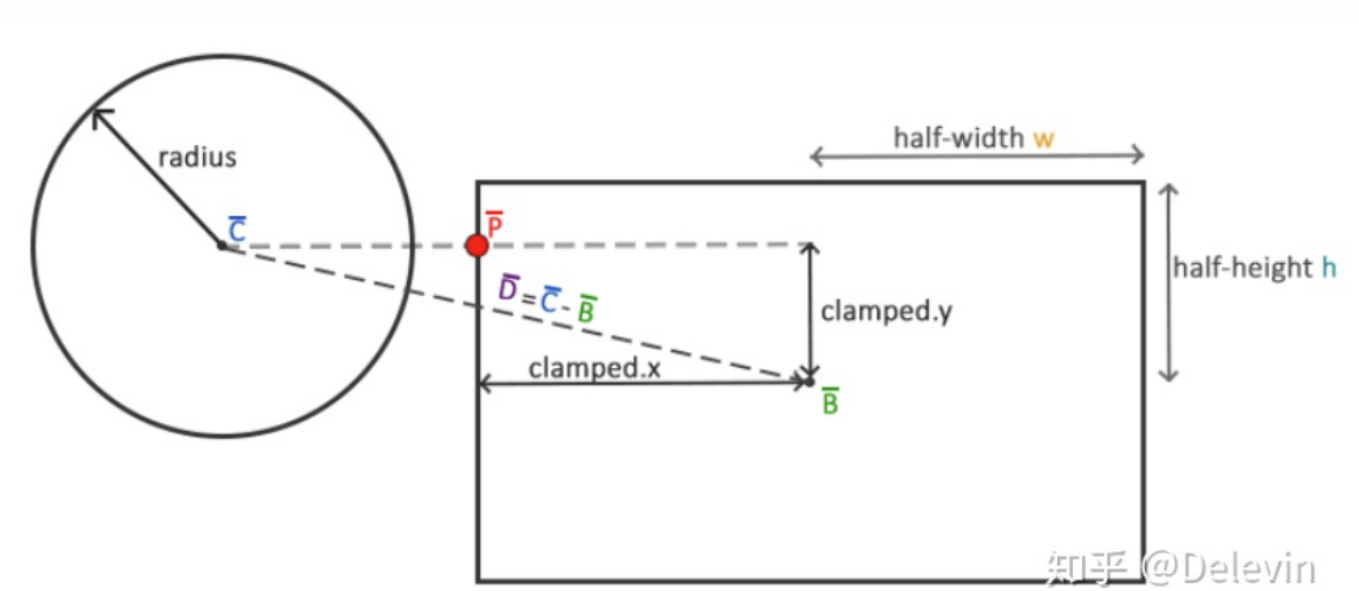


叉乘判断，根据 A x B != B X A 的特性，可以判断出按顺序的叉乘结果，可以作为判断依据，相当于在判断点在向量的左边或者右边，所以如果点在多边形所有边向量的同一边（注意：边向量要讲究顺序），那么点就是在多边形内，所以叉乘结果可以作为方向判断依据。

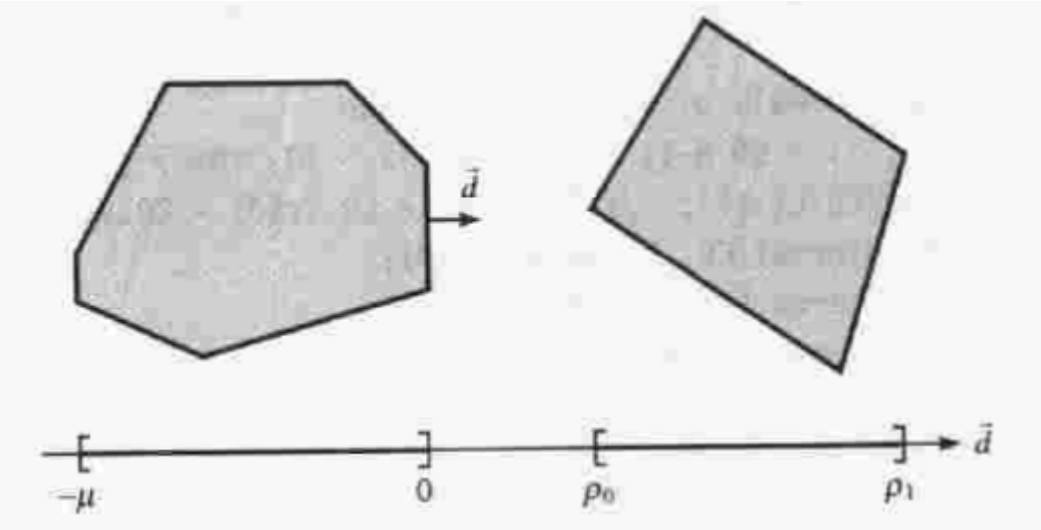
多边形与多边形

先来个开胃菜，判断矩形与圆形是否相交。

如下图所示，判断的关键在于：找到矩形上离圆最近的点P，然后得出点 P 与圆心的距离，再与圆的半径比较。

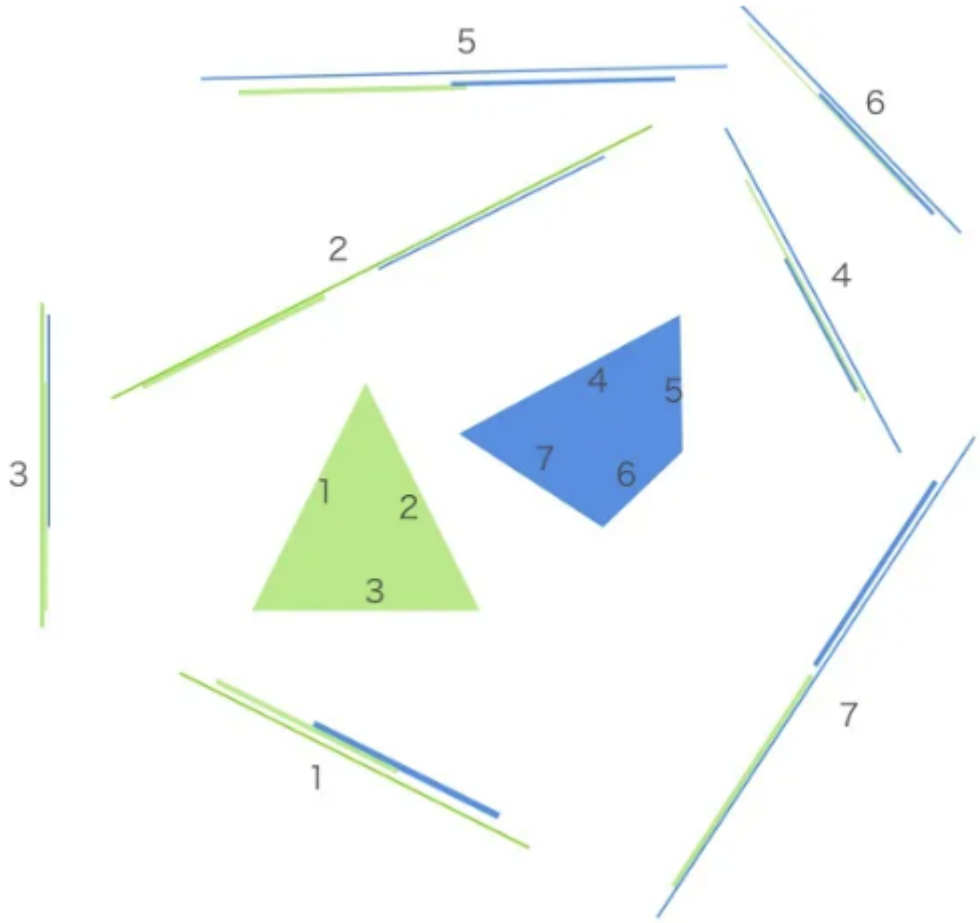


那如果是多边形与多边形。（注意：必须是凸多边形与凸多边形）



(凸包)分离轴定理

在多边形的每条边上取各自的法向量，然后把要比较的两个多边形朝法向量做投影，两个多边形一定会生成各自的投影线段，若存在两个投影线段不相交，那么这两个多变形（凸包）一定不相交。若每个法向量上的投影都相交，那么投影出的多变形（凸包）一定相机。

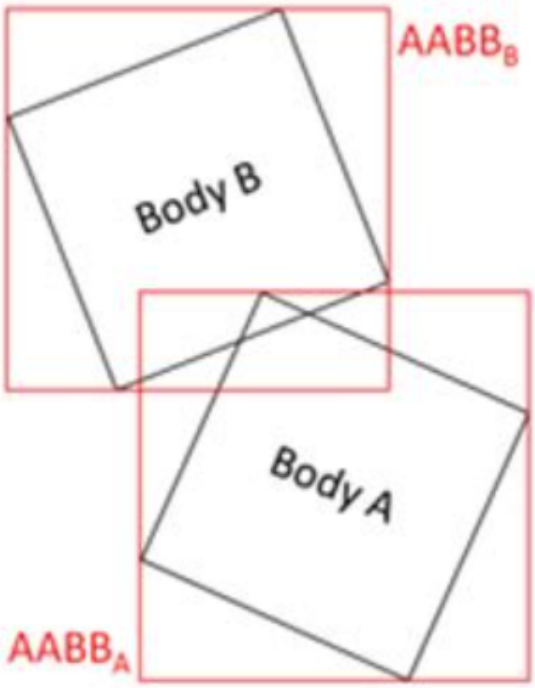


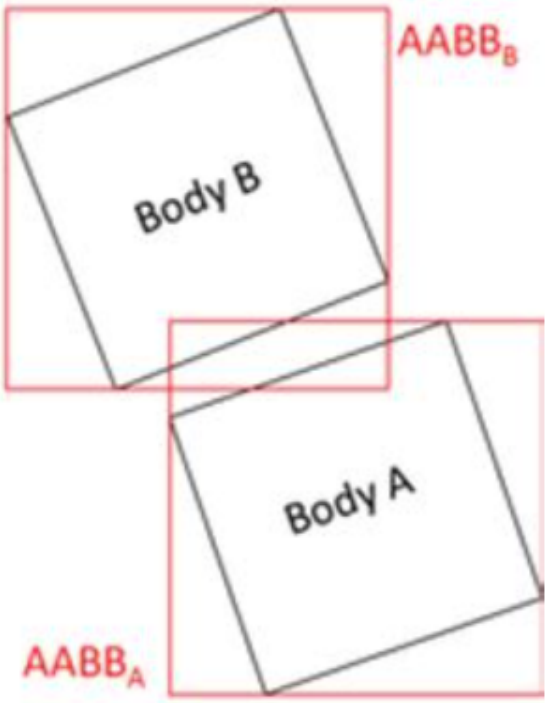
非凸包的多变形怎么办呢？切割成多个凸包！

AABB 包围盒

不管是点与多变形相交，还是多边形与多变形相交，都是细粒度的检测，我们希望能够先用粗粒度的剔除，减少检测判断。

轴对齐包围盒，首先在2D平面内，任意图形都可以用对应尺寸的矩形进行包裹，如果我们包矩形尺寸尽可能匹配，并且保证包围盒轴对齐。那么就可以得出如下结果。如下图：





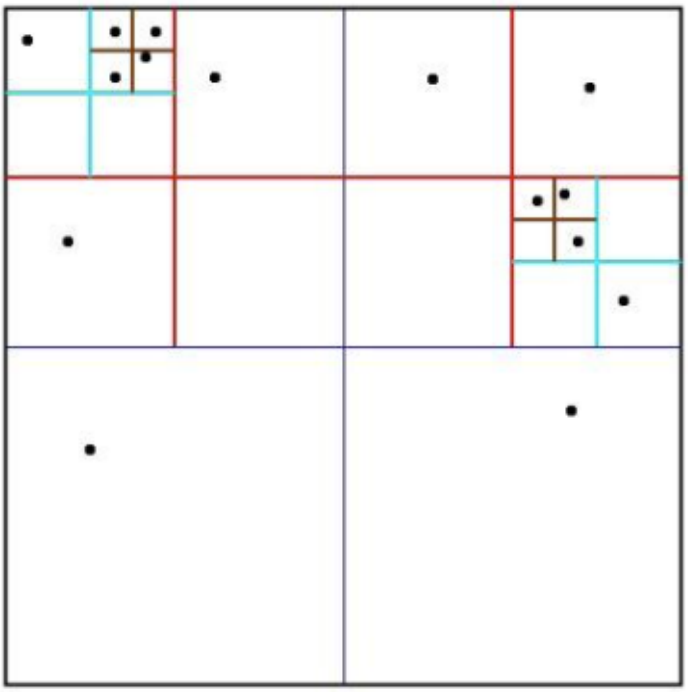
注：之所以要用轴对齐的包围盒，也是希望运算简单，如果轴对齐，那么矩形就可以通过 X(min,max), Y(min,max) 来表示，因为轴对齐，所以矩形的右上角顶点一定是（Xmax,Ymax），矩形的左下角顶点一定是（Xmin,Ymin）。那么当判断点是否在这个轴对齐包围盒时，只需要判断 X 与 Xmin、Xmax,Y 与 Ymin 、Ymax 的关系。

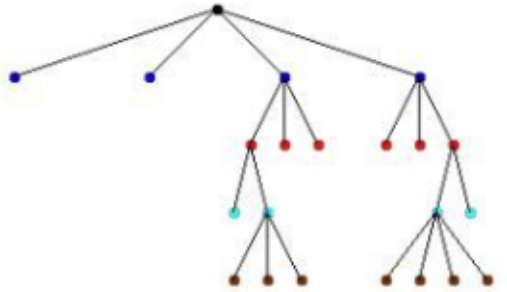
如果不满足轴对齐包围盒的范围，那么都不需要进行细粒度的检测判断，从而节省了复杂的运算。

四叉树优化

另外，在一个有限范围的2D 游戏世界，当存在 n 个游戏对象时，如果不做任何简化，似乎进行两两比较，时间复杂度为： n ^ 2。为此，我们还应当再做更加粗粒度的判断。

我们把2D游戏世界分为四叉树空间。任何游戏对象都属于四叉树划分下的叶子节点下。





可以看下三年前写的这篇文章： https://zhuanlan.zhihu.com/p/83722268

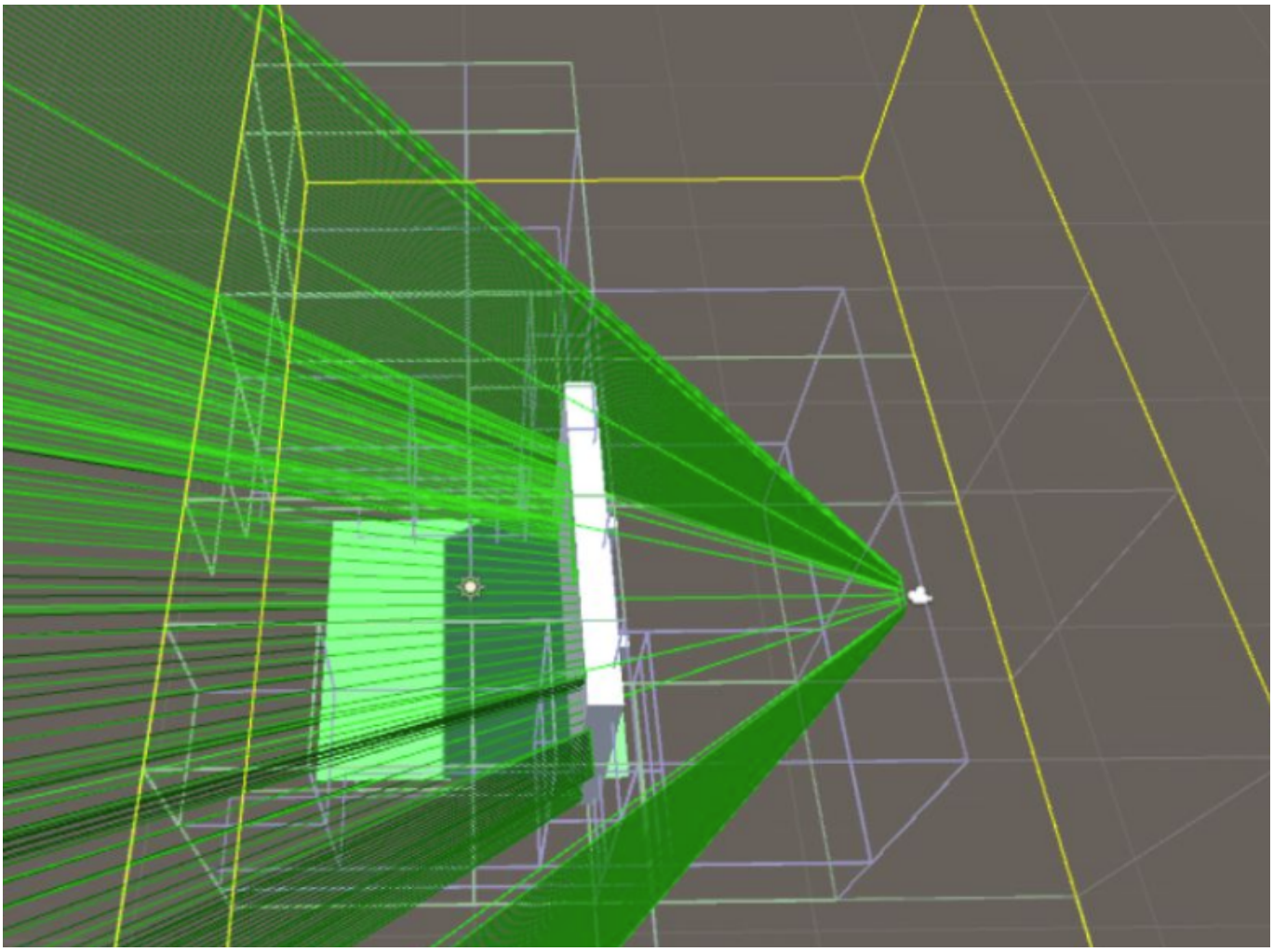
3D 碰撞

同理，我们按照2D碰撞检测的思路，为3D对象构建3D 的 AABB 包围盒，得到(Xmin,Ymin,Zmin) (Xmax,Ymax,Zmax)，然后进行粗粒度的判断。 然后，根据3D对象的面上的法向量，做两个3D对象（注：凸包，下同）的投影，如果存在两个对象投影无交集的情况，那么就说明3D对象不相交。

八叉树优化

将有限的游戏世界划分，这样3D对象被划分到了对应的叶子节点下。也就意味着可以只进行小范围的碰撞检测，减少计算。

另外，八叉数在遮挡剔除中也有运用，这是思想比较契合“空间换时间”。通过离线生成游戏场景的静态八叉树数据，使得在运行时，可以直接通过摄像机的位置与观察方向结合八叉树数据，得到一个遮挡关系，从而剔除掉被遮挡的对象。如下图：



LeetCode

职业生涯中，感受到了算法的乐趣，因此在几年前，进行了一段 LeetCode 刷题经历，其中遇到了不少让我拍案叫绝的算法，因此，再次列举一些，作为分享。

相交链表

https://leetcode.cn/problems/intersection-of-two-linked-lists/

链表本身是一个极其基础的数据结构，如果认识不深刻，那么就无法意识到 node.next 意味着什么。node.next 的地址唯一性，就注定了相交后的所有节点必定也相交。这是解这道题的关键！

只出现一次的数字

https://leetcode.cn/problems/single-number/

这个问题的本质，其他出现2次，只有一个数字出现一次，找到这个数。如果熟悉异或，一点就通了。

环形链表

https://leetcode.cn/problems/linked-list-cycle/

环形链表意味着链表没有终点，node.next 永远不为 null，必定会指向一个节点。并且必定会在某个节点循环，所以问题就变成了，怎么判断链表在循环？如果链表有循环，那么必定会是在某个节点形成环形。比较取巧的办法是快慢指针，如果快慢指针会相遇，那么必定是在环上的节点相遇，也就可以判断链表是否是环形链表。

装最多水的容器

https://leetcode.cn/problems/container-with-most-water/

简化问题，题目表示面积越大，容量越大，只要保证包含的面积最大即可，因此，我们按照横向的距离，从大到小缩放，统计出面积的大小，缩放的规则是比较高度值，趋向于优先剔除短板的高度。这就是解题的大致思路。

References

https://zhuanlan.zhihu.com/p/385733813

https://www.cnblogs.com/KillerAery/p/12242445.html

https://juejin.cn/post/6979895361397587982

https://www.cnblogs.com/674001396long/p/9901811.html

https://www.cnblogs.com/coder-tcm/p/11437045.html

https://zhuanlan.zhihu.com/p/81426117

https://www.cnblogs.com/pointer-smq/p/11332897.html

https://blog.csdn.net/youngyangyang04/article/details/110095497