

# Computer Animation and Game Design

Advanced Computer Game Design

Kun Zeng

63212114@qq.com

A\* 算法  
alpha-beta剪枝算法  
极大极小算法

# Background

## 意义

- A\*搜寻算法，俗称A星算法。这是一种在二维平面上，对于多个节点的路径，求出最低通过代价的算法。常用于游戏中的NPC的移动计算，或在线游戏的BOT的移动计算上。

## 中国象棋的搜索策略

- 极大极小算法
- alpha-beta 剪枝算法

# 1. 相关背景

## 1. 启发式搜索

- 启发式搜索就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。这样可以省略大量无畏的搜索路径，提到了效率。在启发式搜索中，对位置的估价是十分重要的。

## 2. 估价函数

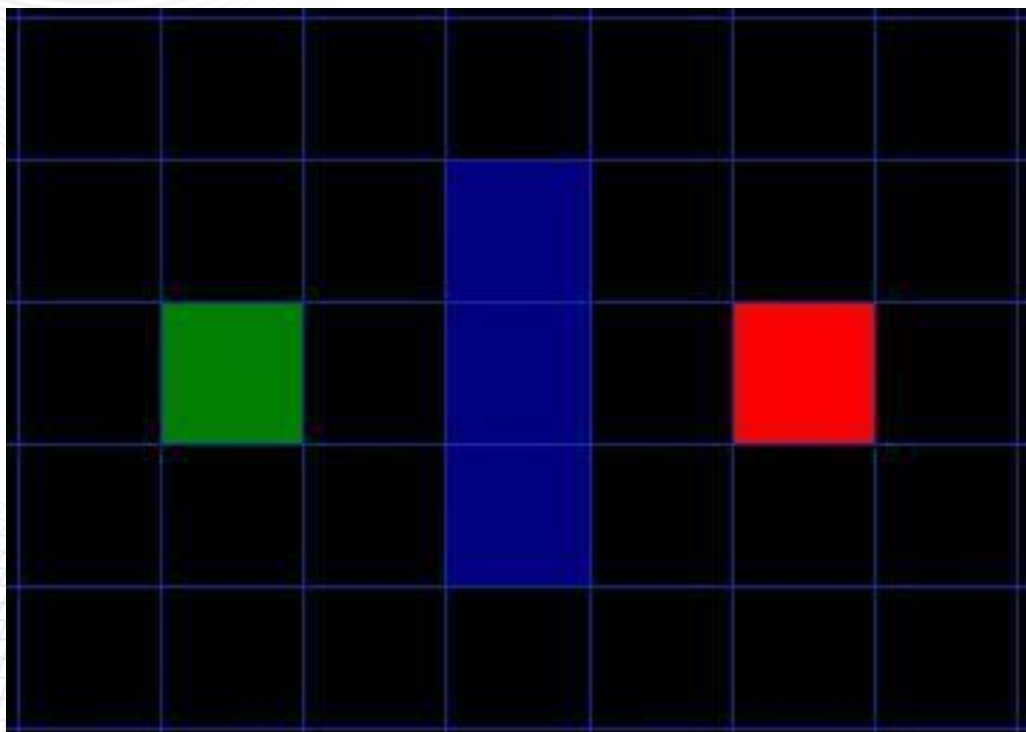
- 从当前节点移动到目标节点的预估费用；这个估计就是启发式的。在寻路问题和迷宫问题中，我们通常用曼哈顿（manhattan）估价函数（下文有介绍）预估费用。

## 3. 优缺点

A\*算法在理论上是时间最优的，但是也有缺点：它的空间增长是指数级别的。

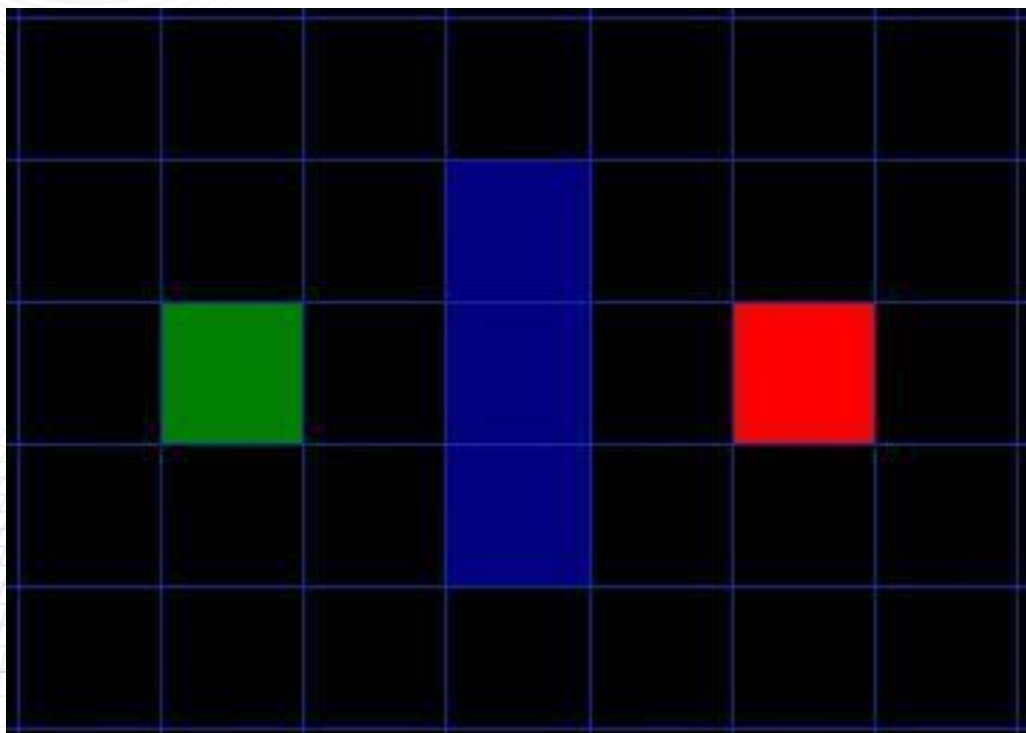
## 2. A\*算法基本原理

**目标：** 假设有人想从A点移动到一墙之隔的B点，如下图，绿色的是起点A，红色是终点B，蓝色方块是中间的墙。



## 2.1 初始化网格

搜索区域被我们划分成了方形网格。第一步简化搜索区域，把搜索区域简化成了一个二维数组。数组的每一个元素是网格的一个方块，方块被标记为可通过的和不可通过的。路径被描述为从A到B我们经过的方块的集合。一旦路径被找到，我们的人就从一个方格的中心走向另一个，直到到达目的地。





## 2.2 开始搜索

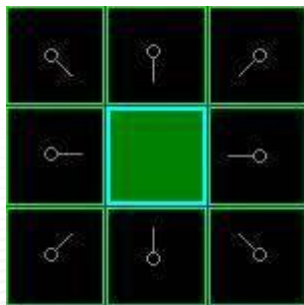
搜索区域初始化为节点，下一步就是去引导一次找到最短路径的搜索。

在A\*寻路算法中，我们通过从点A开始，检查相邻方格的方式，向外扩展直到找到目标。步骤如下：

1. 从点A开始，并且把它作为待处理点存入一个“开启列表”。开启列表就像一张购物清单。尽管现在列表里只有一个元素，但以后就会多起来。总路径可能会通过它包含的方格，也可能不会。
2. 寻找起点周围所有可到达或者可通过的方格，跳过有墙，水，或其他无法通过地形的方格。也把他们加入开启列表。为所有这些方格保存点A作为“父方格”。当我们想描述路径的时候，父方格的资料是十分重要的。后面会解释它的具体用途。

## 2.2 开始搜索

- 从开启列表中删除点A，把它加入到一个“关闭列表”，列表中保存所有不需要再次检查的方格。在这一点，你应该形成如图的结构。在图中，暗绿色方格是你起始方格的中心。它被用浅蓝色描边，以表示它被加入到关闭列表中了。所有的相邻格现在都在开启列表中，它们被用浅绿色描边。每个方格都有一个灰色指针反指他们的父方格，也就是开始的方格。



- 接着，我们选择开启列表中的临近方格，根据F值最低的原则选取方格。



## 2.2 开始搜索

### 路径评分

$$F = G + H$$

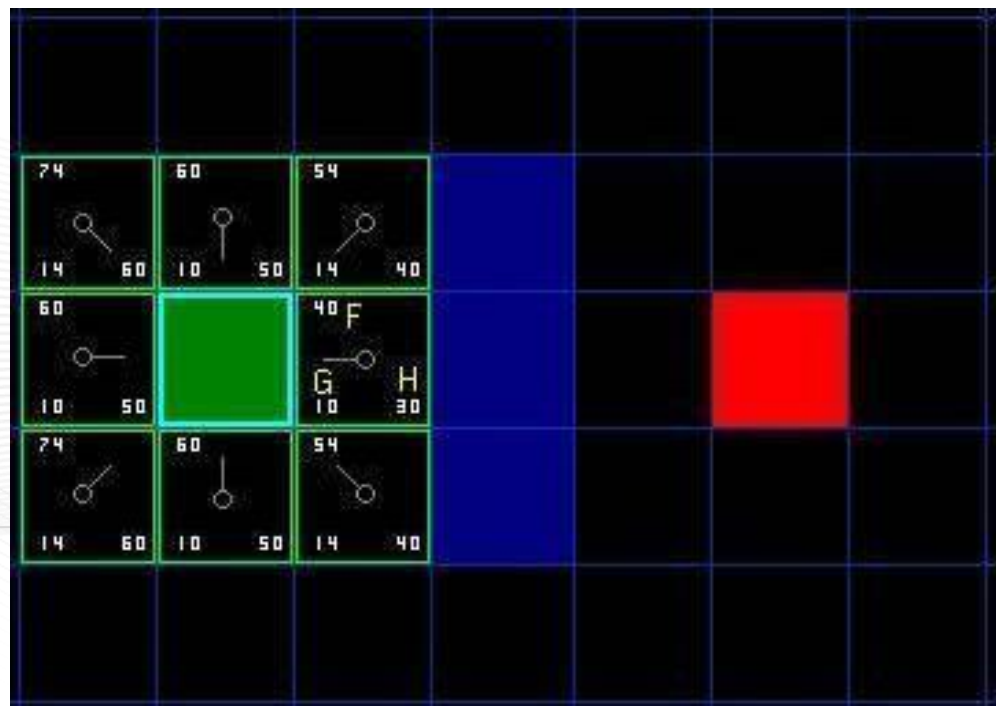
- $G$  = 从起点A，沿着产生的路径，移动到网格上指定方格的移动耗费。
- $H$  = 从网格上那个方格移动到终点B的预估移动耗费。我们没办法事先知道路径的长度，因为路上可能存在各种障碍(墙，水，等等)。虽然本文只提供了一种计算H的方法。

$G$ 表示沿路径从起点到当前点的移动耗费。在这个例子里，我们令水平或者垂直移动的耗费为10，对角线方向耗费为14。我们取这些值是因为沿对角线的距离是沿水平或垂直移动耗费的 $\text{SQRT}(2)$ ，或者约1.414倍。为了简化，我们用10和14近似。

## 2.2 开始搜索

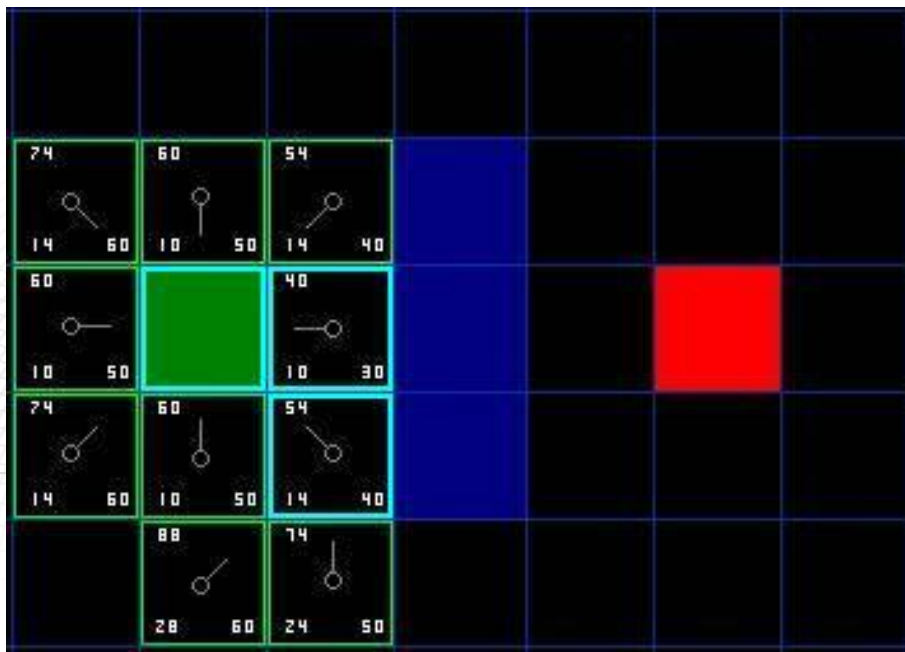
H值可以用不同的方法估算。我们这里使用的方法被称为曼哈顿方法，它计算从当前格到目的格之间水平和垂直的方格的数量总和，忽略对角线方向。然后把结果乘以10。这被成为曼哈顿方法是因为它看起来像计算城市中从一个地方到另外一个地方的街区数，在那里你不能沿对角线方向穿过街区。很重要的一点，我们忽略了一切障碍物。这是对剩余距离的一个估算，而非实际值，这也是这一方法被称为启发式的原因。

F的值是G和H的和。第一步搜索的结果可以在下面的图表中看到。F,G和H的评分被写在每个方格里。正如在紧挨起始格右侧的方格所表示的，F被打印在左上角，G在左下角，H则在右下角

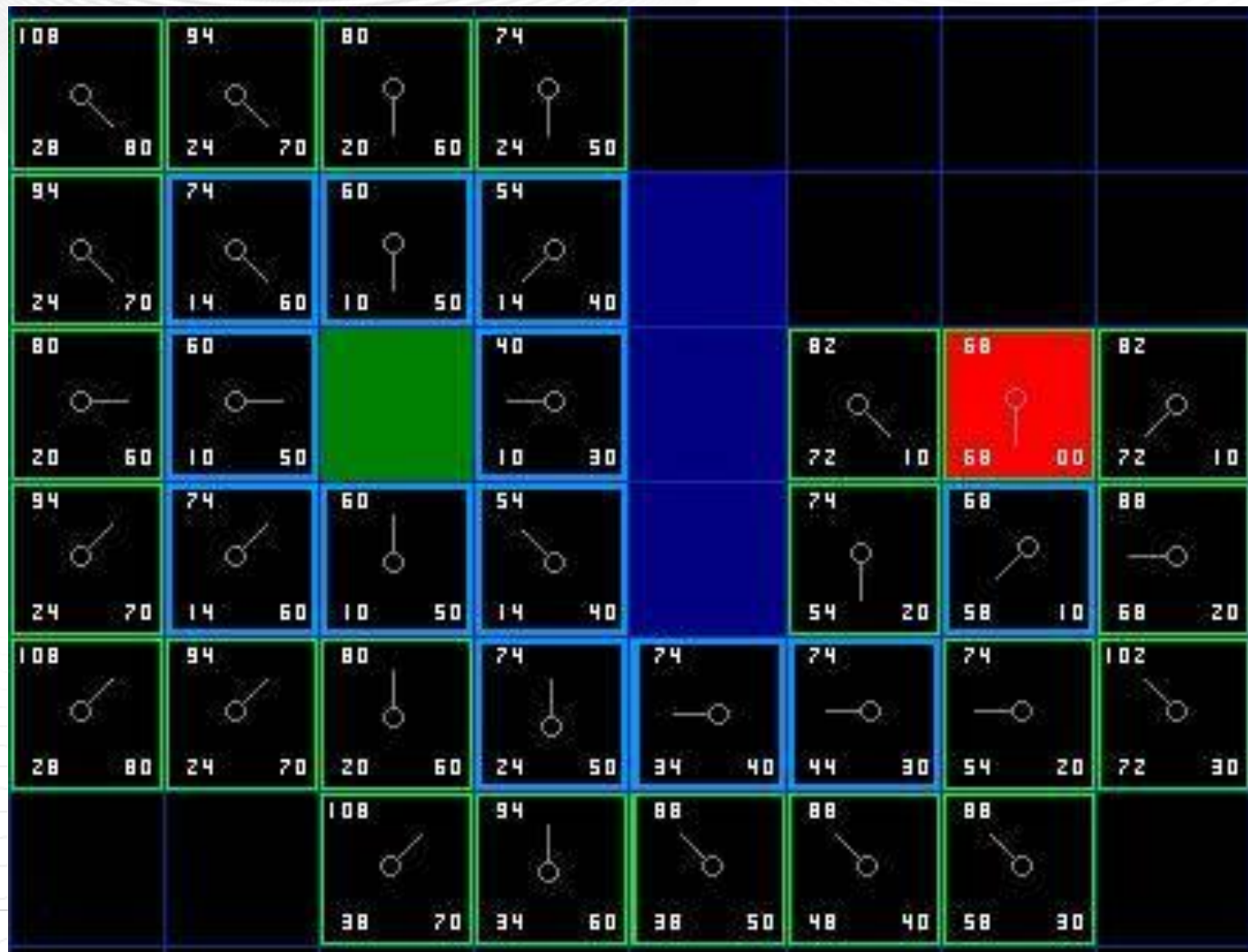


## 2.2 开始搜索

5. 把它从开启列表中删除，然后添加到关闭列表中。
6. 检查所有相邻格子。跳过那些已经在关闭列表中的或者不可通过的(有墙，水的地形，或者其他无法通过的地形)，把他们添加进开启列表，如果他们还不es不在里面的话。把选中的方格作为新的方格的父节点。
7. 如果某个相邻格已经在开启列表里了，检查现在的这条路径是否更好。换句话说，检查如果我们用新的路径到达它的话，G值是否会更低一些。如果不是，那就什么都不做。

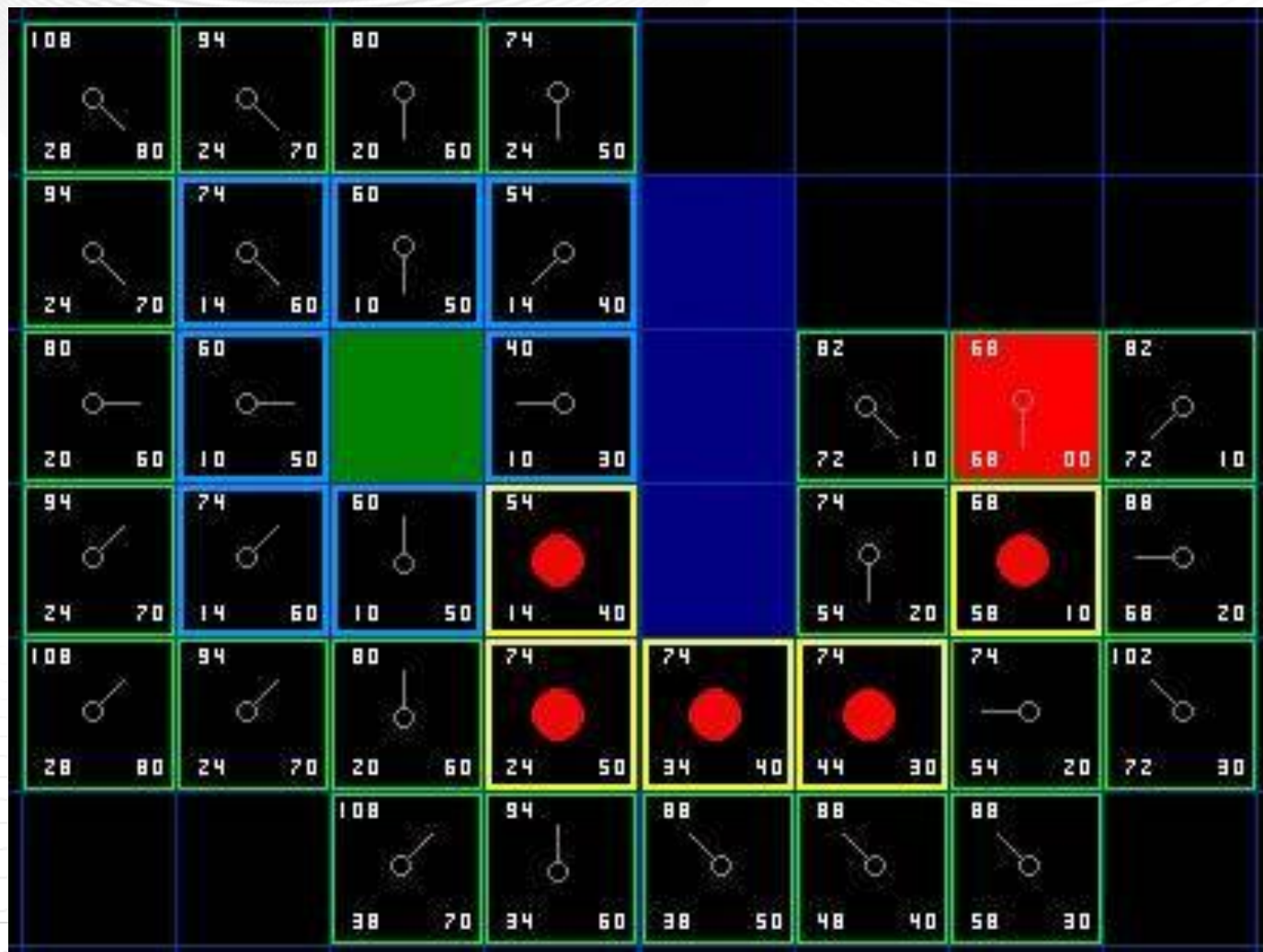


## 2.2 开始搜索





## 2.2 开始搜索



## 2.2 算法小结

1. 把起始格添加到开启列表。
2. 重复如下的步骤：
  - a) 寻找开启列表中F值最低的格子。我们称它为当前格。
  - b) 把它切换到关闭列表。
  - c) 对相邻的8格中的每一个？
    - \* 如果它不可通过或者已经在关闭列表中，略过它。反之如下。
    - \* 如果它不在开启列表中，把它添加进去。把当前格作为这一格的父节点。记录这一格的F,G,和H值。
    - \* 如果它已经在开启列表中，用G值为参考检查新的路径是否更好。更低的G值意味着更好的路径。如果是这样，就把这一格的父节点改成当前格，并且重新计算这一格的G和F值。如果你保持你的开启列表按F值排序，改变之后你可能需要重新对开启列表排序。



## 2.2 开始搜索

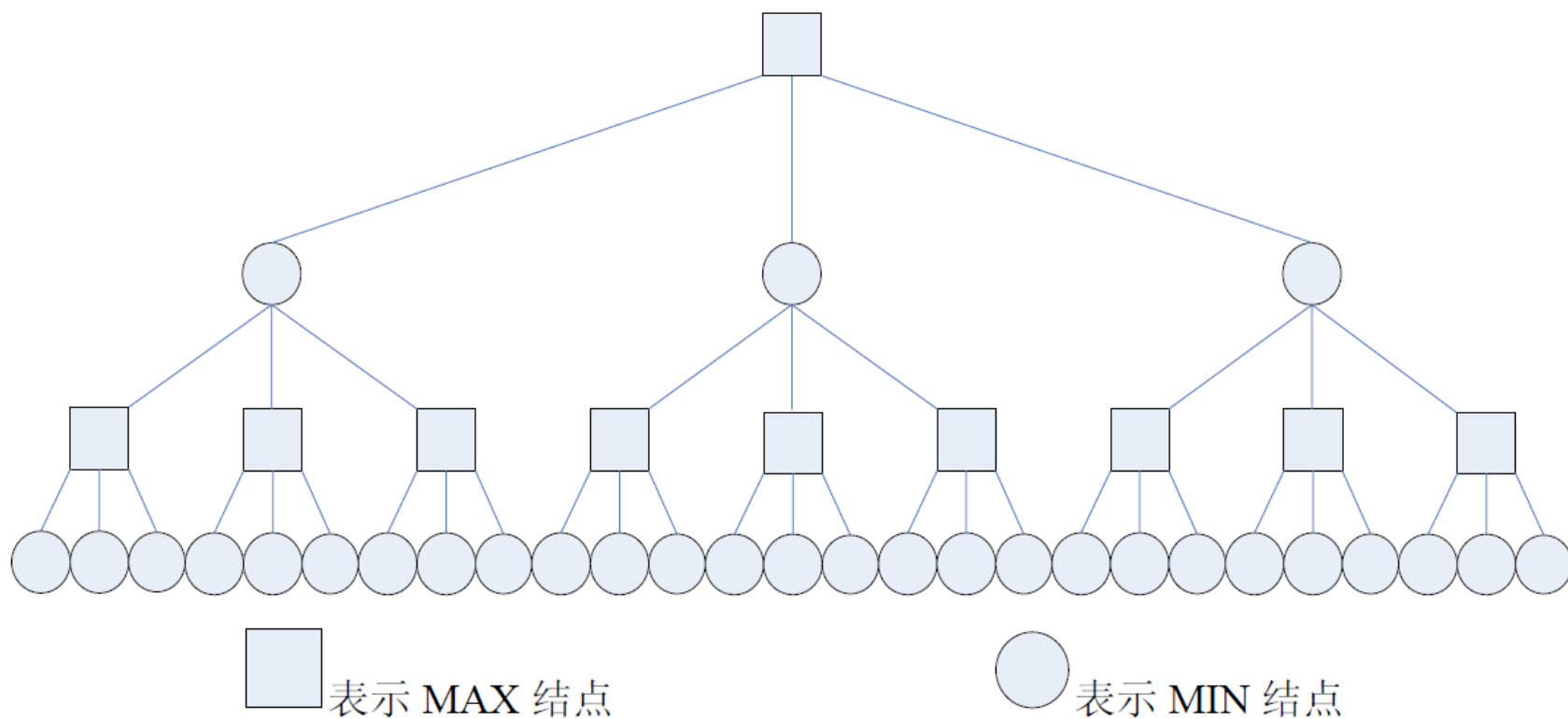
d) 停止，当你

- \* 把目标格添加进了开启列表，这时候路径被找到，或者
- \* 没有找到目标格，开启列表已经空了。这时候，路径不存在。

3. 保存路径。从目标格开始，沿着每一格的父节点移动直到回到起始格。这就是你的路径。

### 3. 中国象棋的搜索策略

中国象棋是一种完全知识的博弈游戏，也就是无论棋局进展到什么程度，双方各自的剩余棋子，阵型等等信息都是对方可见的。从这个意思上说，对弈过程可以用一棵树来模拟。双方交替出着，最后组成了一棵博弈树。

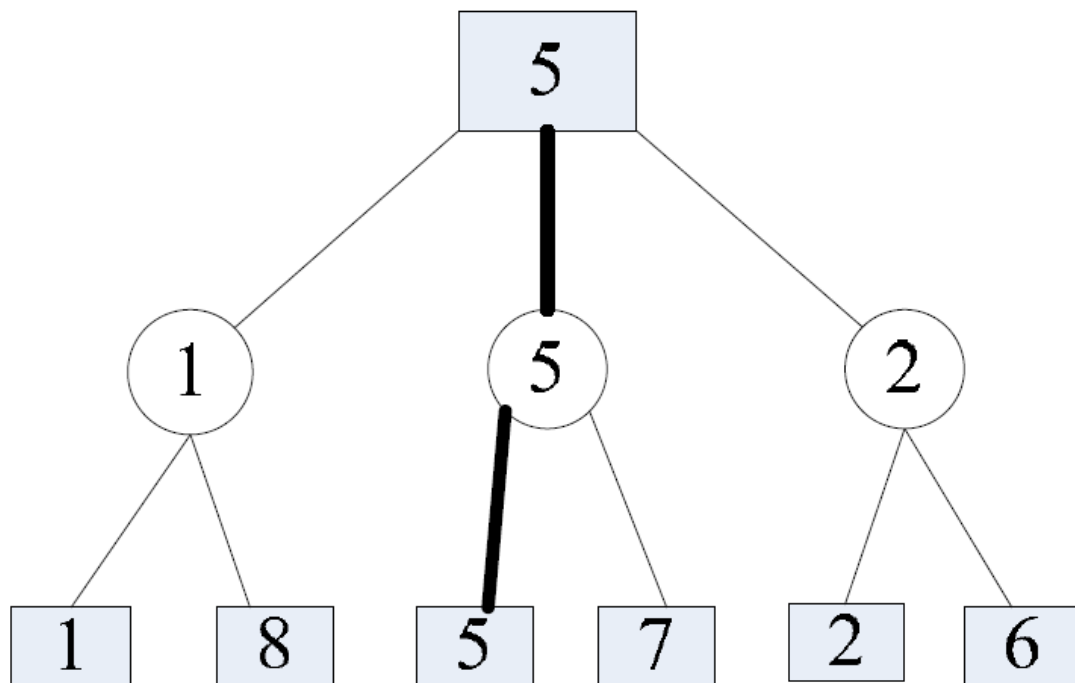


## 3.1 极大极小算法

在这棵博弈树中，根节点是初始局面，一个节点下的每个分支代表一个着法，分支相连的子节点都代表如果走了这个着法后会出现的局面。由于双方交替出着，因此相邻的两层是分别站在双方各自的角度上进行了考虑，使自己的利益最大化。

红方走棋时，红方总是选择对自己最有利的着法，从而在生成的一系列子节点中寻找分值最高的局面（即MAX层），据此返回最佳着法；然后轮到黑方走棋，黑方也选对自己最有利的着法，但在红方看来，这个着法对自己最不利，所以从红方的角度来看，黑方就选择了分值最低的局面（即MIN层），从而返回黑方的最佳着法。

## 3.1 极大极小算法



在这棵树中，矩形代表红方，圆代表黑方，分别取子节点的极大/极小值。根据上面的描述，我们选择了第二个着法。而加粗的这条路径就是极大极小算法思考的过程。

```

int MAXMIN ( int depth )
{
    if ( depth == 0 ) return Evaluate ( ); //叶子结点返回估值
    if ( MAX 结点)
    {
        for ( 每一步可能的着法 m )
        {
            执行着法 m;
            value = MAXMIN (depth - 1 );
            撤销着法 m;
            if ( value > best ) best = value;
        }
    }
}

```

从代码当中不难看出，极大极小算法需要比较每一层所有的节点，才能够找到该层的最佳着法，因此这是一种穷尽搜索算法，也即香农提出的A策略。

```

if ( MIN 结点)
{
    for ( 每一步可能的着法 m )
    {
        执行着法 m;
        value = MAXMIN (depth - 1 );
        撤销着法 m;
        if ( value < best ) best = value;
    }
}
Return best;
}

```



## 3.2 alpha-beta 剪枝算法

在开局阶段，中国象棋每个局面下平均有四十多种后继局面，层层展开的话，节点数量就是一个以指数量级迅速增加的天文数字。

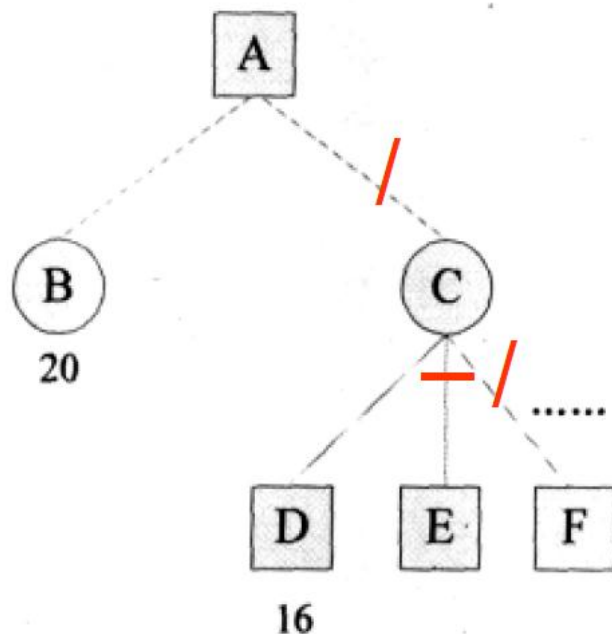
只能指定一个深度，让算法只搜索限定深度下的节点，然后返回这个深度下的最佳着法。由于更深层的着法更接近对弈的结局，因此被认为是更好的着法，所以我们要搜索的尽量深一点。

然而搜索深度越深，节点数就越多，算法反应时间就越长，很快变得让人难以接受。所以，我们需要有选择的搜索，通过剪掉某些不必要的分支，减少节点数量，增加搜索深度。

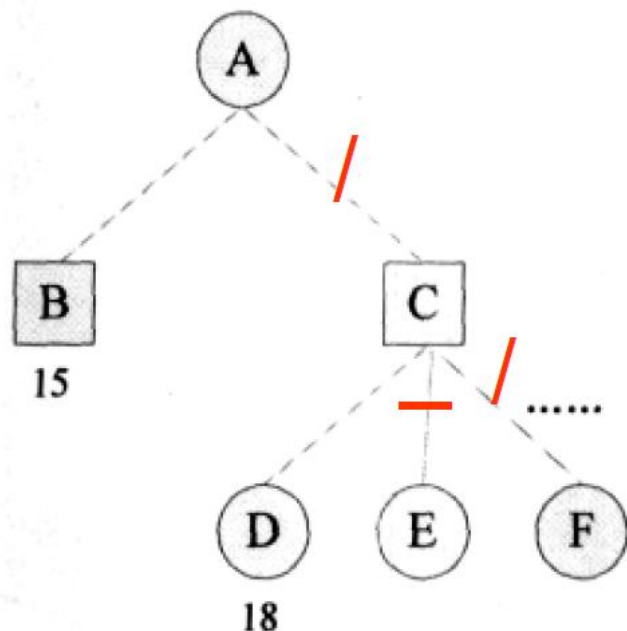


## 3.2 alpha-beta 剪枝算法

问题在于，需要剪掉哪些分支？在下面这棵博弈树中，发现有些节点是没有必要搜索的：



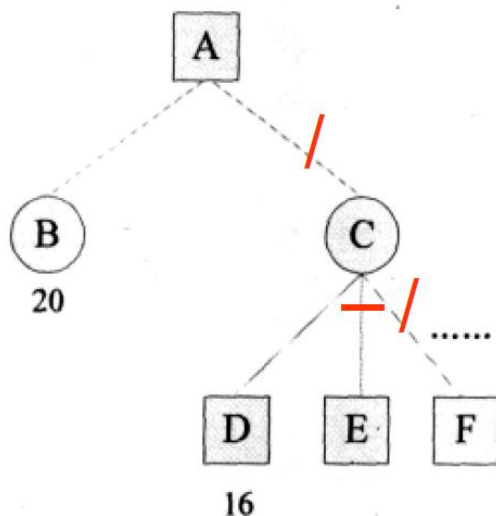
□ 取极大值的节点




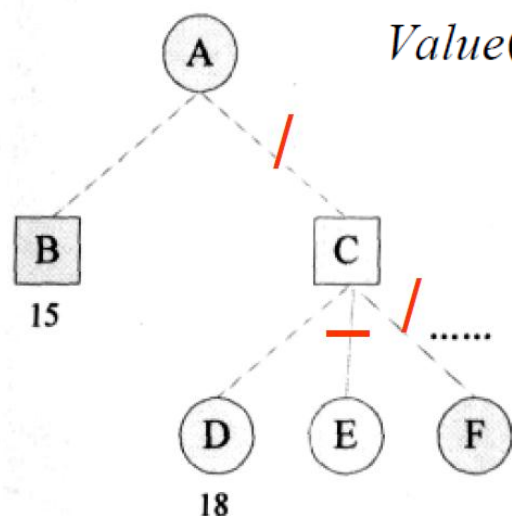
○ 取极小值的节点


## 3.2 alpha-beta 剪枝算法

在上图左侧，由于C节点取子节点的最小值，搜索到D节点时发现，D的值为16，因此C的值最大为16，也就是说C的值肯定没有B的值大，而A节点取根节点的最大值，那么就会选择B为下一步扩展节点，可以将以C这一分支剪枝。这个过程可以用一个公式来表示：



 取极大值的节点



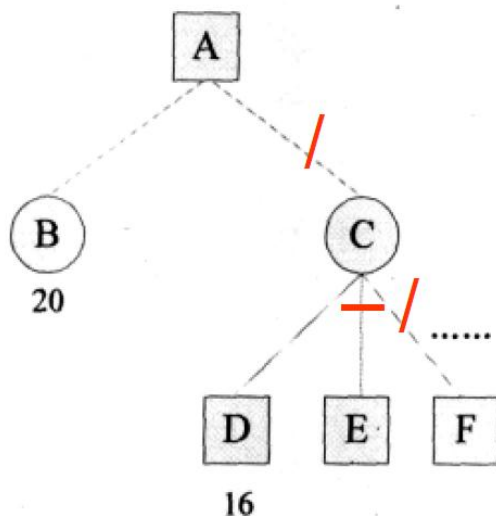
 取极小值的节点


$$Value(A) = \text{Max}(20, \text{Min}(16, E, F))$$

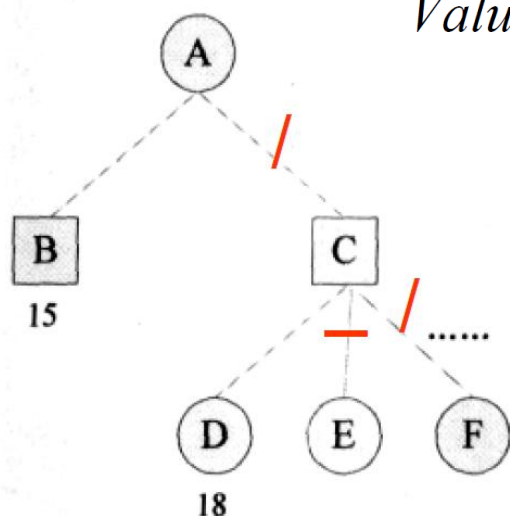
## 3.2 alpha-beta 剪枝算法


这个过程是 alpha 剪枝。右边的情况类似，C 节点取子节点的最大值，搜索到 D 时发现 D 的值是 18，因此 C 的值最小为 18，也即 C 的值肯定比 B 的值大，而 A 节点取根节点的最小值，就会选择 B 作为下一个扩展节点，C 这一分支就可以剪掉。这个过程是 beta 剪枝，其公式如下：

$$Value(A) = Min(15, Max(18, E, F))$$



 取极大值的节点



 取极小值的节点

## 3.2 alpha-beta 剪枝算法

上面这两个过程合起来就是alpha-beta搜索。

alpha-beta 搜索产生剪枝之后，剪掉了一些节点，同时也将以这些节点为根的所有子树都剪掉了。alpha-beta 搜索遍历一棵固定深度的博弈树所用的时间要远远少于极大极小算法。

## 3.2 alpha-beta 剪枝算法

```
int AlphaBeta ( int depth , int alpha, int beta )
{
    if ( depth == 0 ) return Evaluate ( ); //叶节点
    for ( 每一步可能的着法 m )
    {
        执行着法 m;
        value = -AlphaBeta ( depth - 1, -beta , -alpha, );
        //应用负极大值思想
        撤销着法 m;
        if ( value > alpha )
            alpha = value ; //取最大值
            if ( alpha >= beta ) break; //发生 beta 剪枝
    }
    return alpha;
}
```



## 3.2 alpha-beta 剪枝算法

- 历史表法与剪枝算法

博弈树中除了许多相同局面之外，还存在着很多相似局面。相似局面之间只有无关大局的一些子力的落位不同。而好的着法一般是指能对局势产生深远影响的着法，因此有理由相信相似局面之间可能存在相同的“好的着法”。

所谓好的着法是指，该着法引发了大量剪枝，或者是经搜索找到的最佳着法。所以，好的着法不一定是最佳着法。仍然要通过搜索才能得到最佳着法。



## 3.3 alpha-beta 剪枝算法

- 历史表法与剪枝算法

历史表的引入使得系统的效率大为提高，这说明数据库与算法的结合是有效的，因为我们在实际对战时更加关注电脑棋手的反应时间，因此这种以空间换取时间的做法是可取的。

## 3.3 A\*算法

A\*搜索算法是从评估函数最优的分支上进行搜索。

这样做比盲目的深度优先或广度优先搜索要高效得多，但是也存在较大的风险。如果最佳着法所在的分支节点评估值最低，实际上A\*搜索等同于穷尽搜索。