



# 数据结构与算法（ Python ）-12/第13周

北京大学 陈斌

2021.06.01

# 线下课堂

- › 本周内容小结：图及算法（下）
- › 大作业相关：决策树介绍



# W12：图及算法（下）

- › 709 通用的深度优先搜索
- › 710 图的应用：拓扑排序
- › 711 图的应用：强连通分支
- › 712 图的应用：最短路径
- › 713 图的应用：最小生成树
- › 714 图结构小结

# 通用的深度优先搜索

- ❖ 一般的深度优先搜索目标是在图上进行尽量深的搜索，连接尽量多的顶点，必要时可以进行分支（创建了树）

有时候深度优先搜索会创建多棵树，称为“深度优先森林”

- ❖ 深度优先搜索同样要用到顶点的“前驱”属性，来构建树或森林

另外要设置“发现时间”和“结束时间”属性

- 前者是在第几步访问到这个顶点（设置灰色）
- 后者是在第几步完成了此顶点探索（设置黑色）

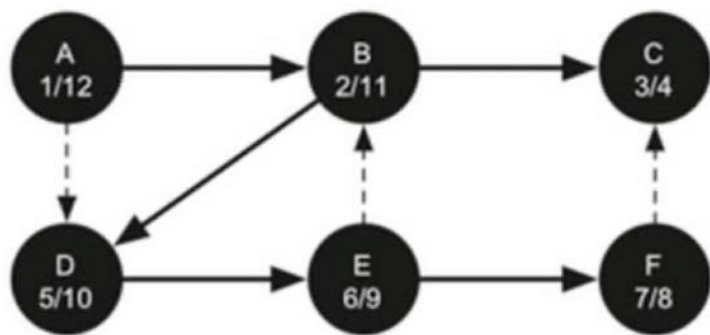
这两个新属性对后面的图算法很重要



# 通用的深度优先搜索算法：分析

❖ DFS构建的树，其顶点的“发现时间”和“结束时间”属性，具有类似**括号**的性质  
 即一个顶点的“发现时间”总小于所有子顶点的“发现时间”

而“结束时间”则大于所有子顶点“结束时间”  
 比子顶点更早被发现，更晚被结束探索



```
def dfsvisit(self, startVertex):
    startVertex.setColor('gray')
    self.time += 1
    startVertex.setDiscovery(self.time)
    for nextVertex in startVertex.getConnections():
        if nextVertex.getColor() == 'white':
            nextVertex.setPred(startVertex)
            self.dfsvisit(nextVertex)
    startVertex.setColor('black')
    self.time += 1
    startVertex.setFinish(self.time)
```

# 拓扑排序Topological Sort

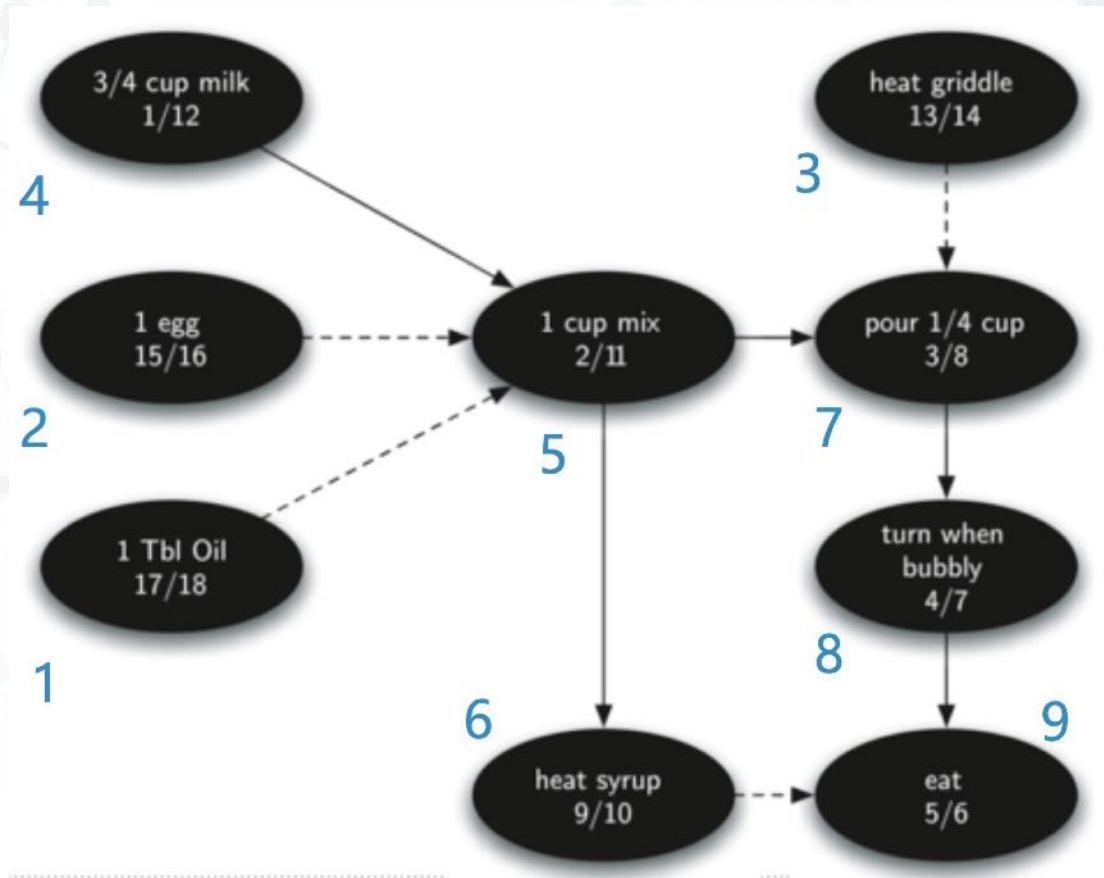
## ❖ 拓扑排序可以采用DFS很好地实现：

将工作流程建立为图，工作项是节点，依赖关系是有向边

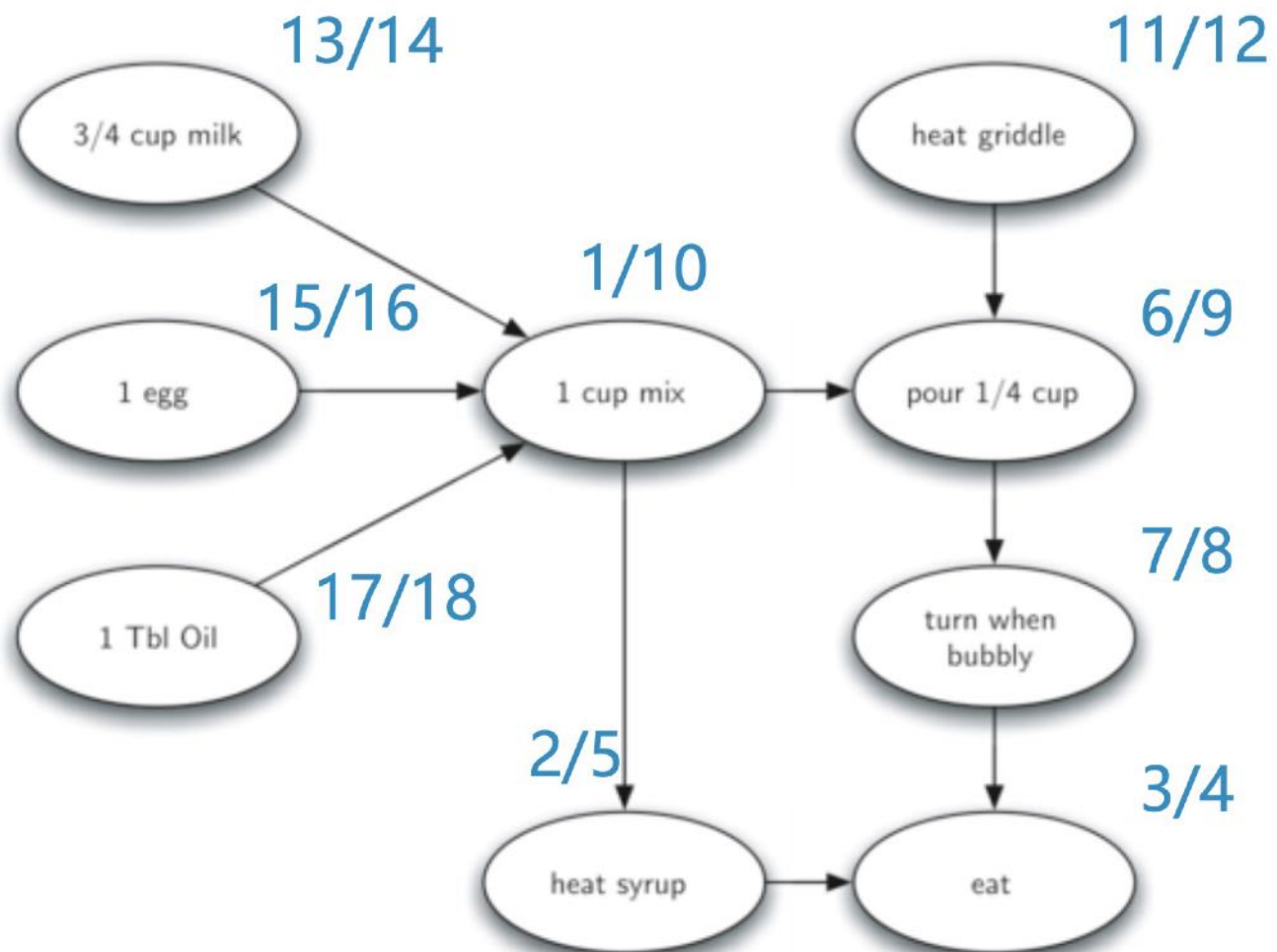
工作流程图一定是一个DAG图，否则有循环依赖

对DAG图调用DFS算法，以得到每个顶点的“结束时间”

按照每个顶点的“结束时间”从大到小排序  
输出这个次序下的顶点列表



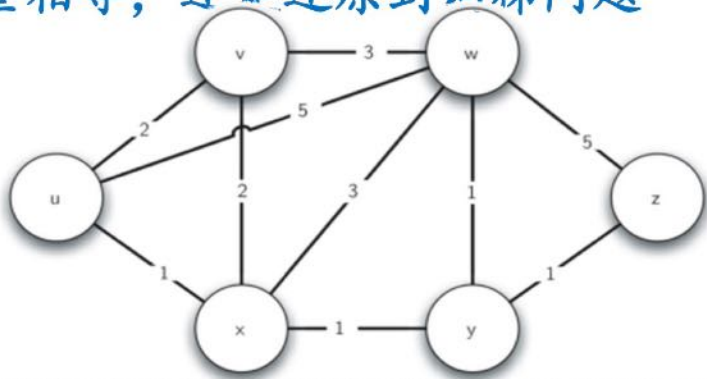
## 拓扑排序：示例2





## 最短路径问题：介绍

- ❖ 解决信息在路由器网络中选择传播速度最快路径的问题，就转变为在**带权图上最短路径**的问题。
- ❖ 这个问题与广度优先搜索BFS算法解决的词梯问题相似，只是在边上增加了权重  
如果所有权重相等，还是还原到词梯问题



## 最短路径问题：Dijkstra算法

- ❖ 顶点的访问次序由一个**优先队列**来控制，队列中作为优先级的是顶点的dist属性。
- ❖ 最初，只有**开始顶点**dist设为0，而其他所有顶点dist设为sys.maxsize（最大整数），全部加入优先队列。
- ❖ 随着队列中每个**最低dist**顶点率先出队
- ❖ 并计算它与邻接顶点的权重，会引起其它顶点dist的减小和修改，引起堆重排
- ❖ 并据更新后的dist优先级再依次出队



# 最短路径问题: Dijkstra算法代码

对所有顶点建堆,  
形成优先队列

优先队列出队

修改出队顶点所邻接  
顶点的dist, 并逐个  
重排队列

```
from pythonds.graphs import PriorityQueue, Graph, Vertex
def dijkstra(aGraph, start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                    + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance( newDist )
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert, newDist)
```

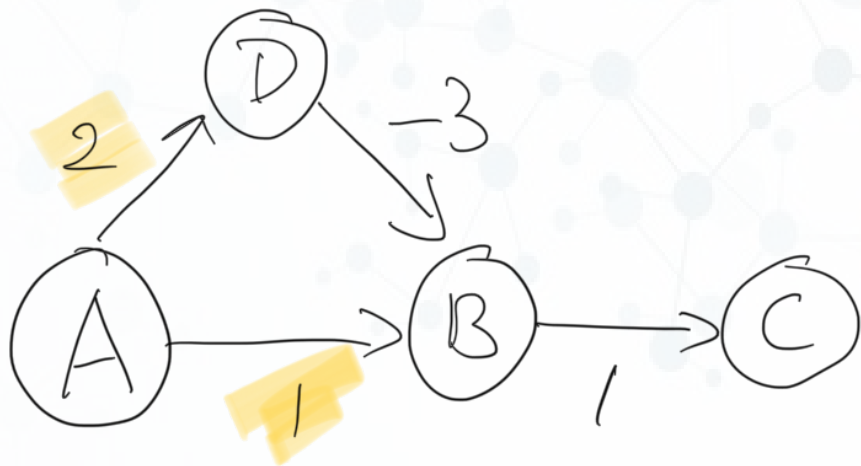
# 问题解答

› 能说说最小路径算法如果出现负权重的情况

› 为什么不一定会得到正确结果吗？

……Dijkstra算法是一个**贪心**算法，优先队列会按照距离值，每次移走一个距离值最小的节点，而且不再更新距离加回队列

默认A-B如果是A连接所有节点的最小权重的话，那么B就会以这个权重作为最短路径，出队如果权重全都非负，这个贪心就成立，否则就不成立。



从 A-B 已有 A-B  
A-D  
最短为 1, B 已  
标黑出队



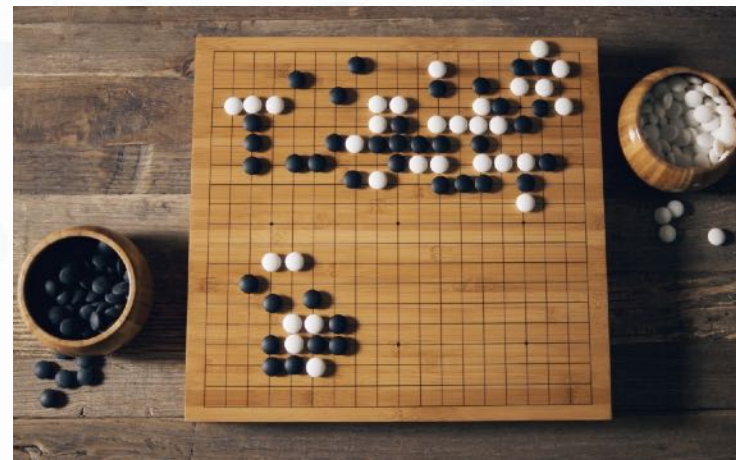
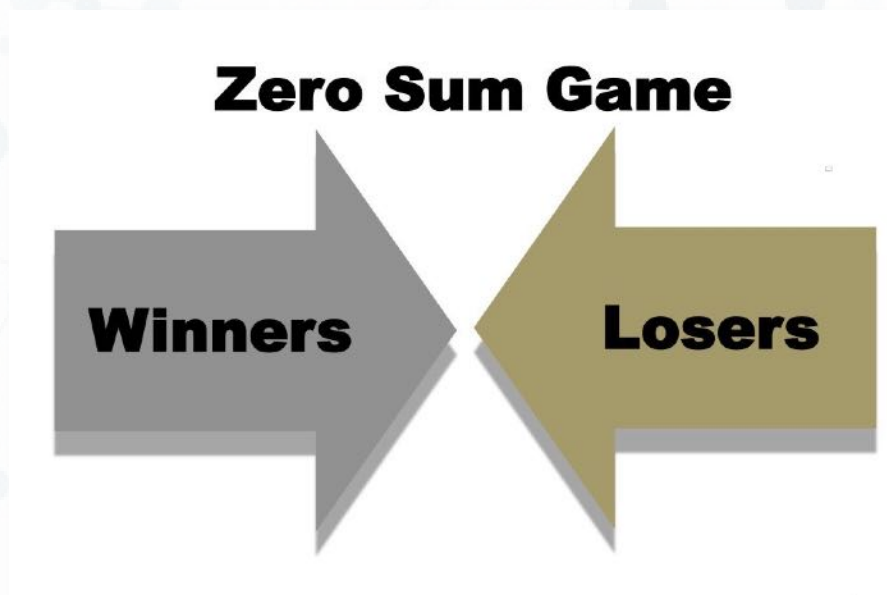
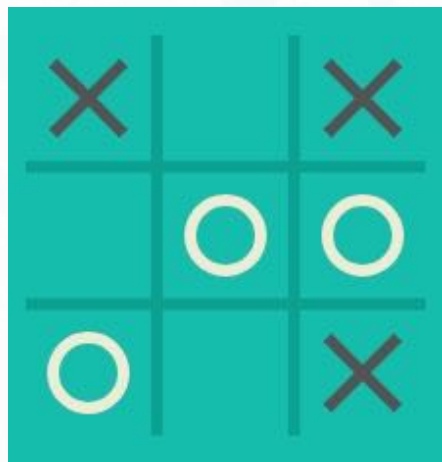


# 大作业相关算法策略

## › 完全信息的零和博弈

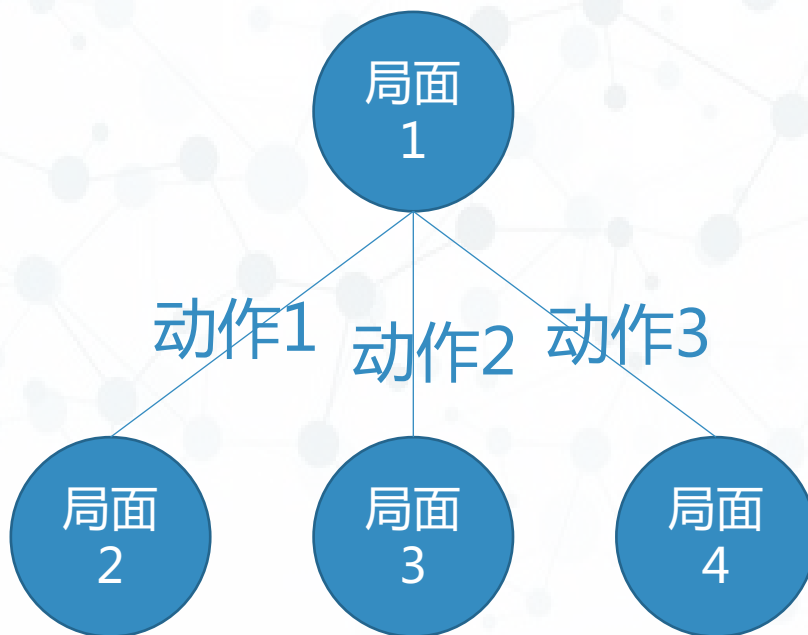
完全信息：双方都知道关于棋局的所有信息，以及查询下棋历史

零和博弈：只需要使对手收益最小化，自己的收益即可实现最大化



# 博弈树构建

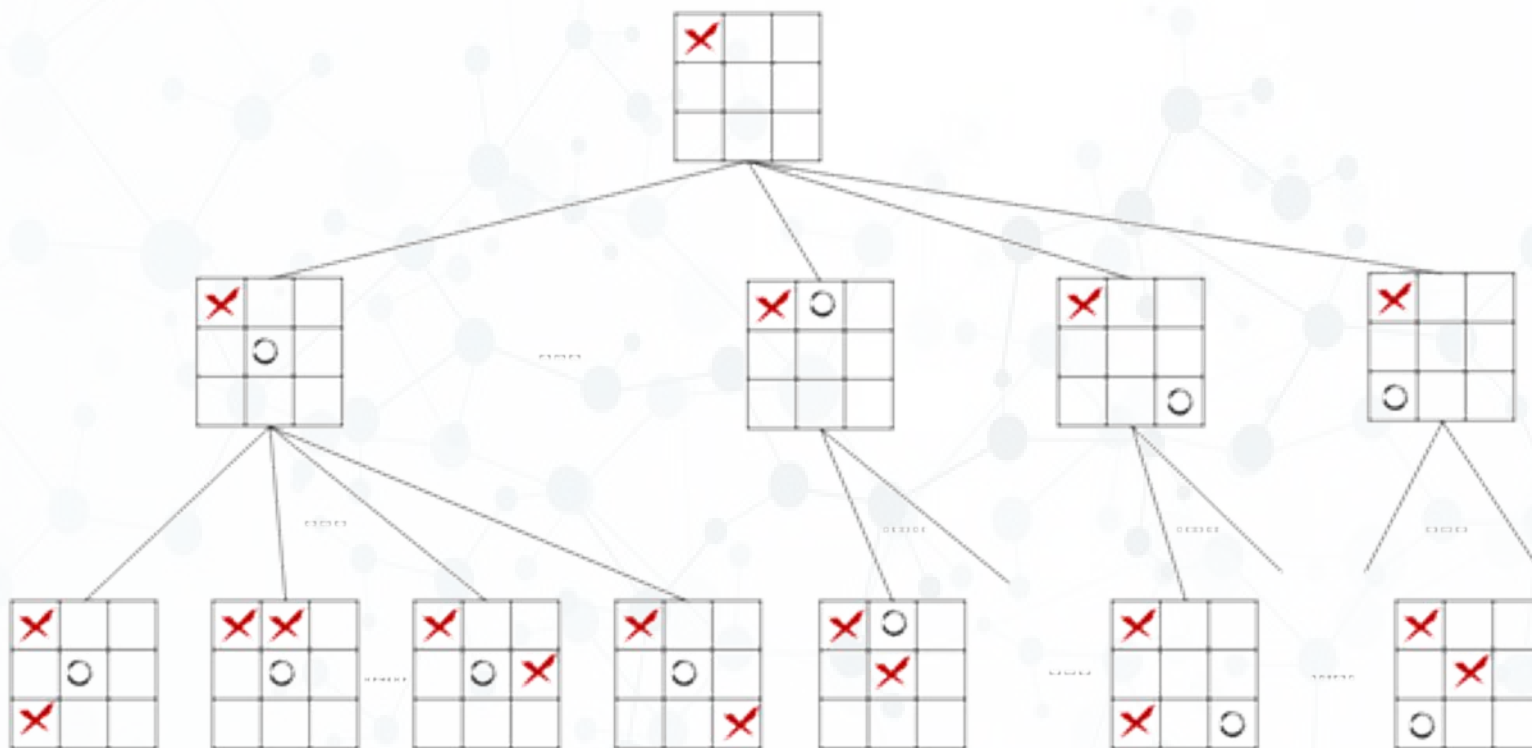
- › 博弈树的每个节点对应于每一个**局面**，每条边对应于一个**动作**
- › 在完全信息零和博弈的条件下，能够构建简单的博弈树
- › 如果在不完全信息、非零和博弈的情况下，博弈树较为复杂



# 博弈树构建

## 博弈树（例子：井字棋）

最高有9层

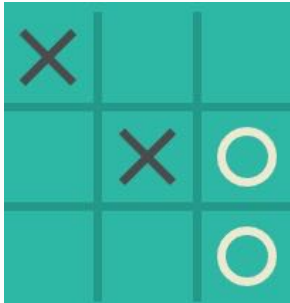




# 如何判断哪个动作最有利？

## 估值函数

对每一种局面给出一个估值

$$f(\text{局面}) = ?$$


## 一般的依据

静态子力、数量

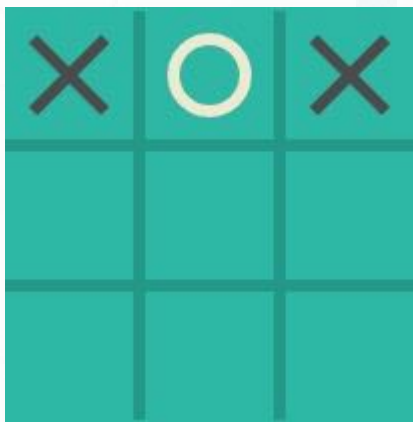
动态特征：不同位置、时间阶段价值不同

		中国象棋		国际象棋	
		价值	数量	价值	数量
	兵(卒)	1	5	2	8
守子	仕(士)	2	2	-	-
	相(象)	2	2	-	-
轻子	马	5	2	6	2
	炮	5	2	-	-
	象	-	-	6	2
重子	车	10	2	10	2
	后	-	-	18	1
总价值(双方)		106	32	156	32
棋盘大小		90		64	

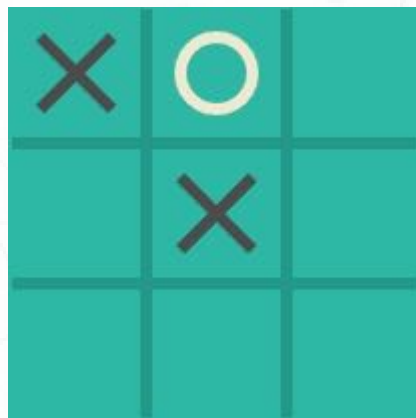
棋子名称		活动范围			子力价值		
红	黑	中央	边线	角落	开局	中局	残局
帅	将	4	3	2	-	-	-
仕	士	4	-	1	1	2	2
相	象	4	2	-	2	2	3
马	马	8/6/4	4/3	2	4	5	5
车	车	17	17	17	10	10	10
炮	炮	17/13	17/14	17/15	5	5	6
兵	卒	1/3	1/2	1	2	1/3/5	3/2/1

# 井字棋的估值函数

- › 玩家X还存在可能性的行、列、斜线数减去
- › 玩家O还存在可能性的行、列、斜线数



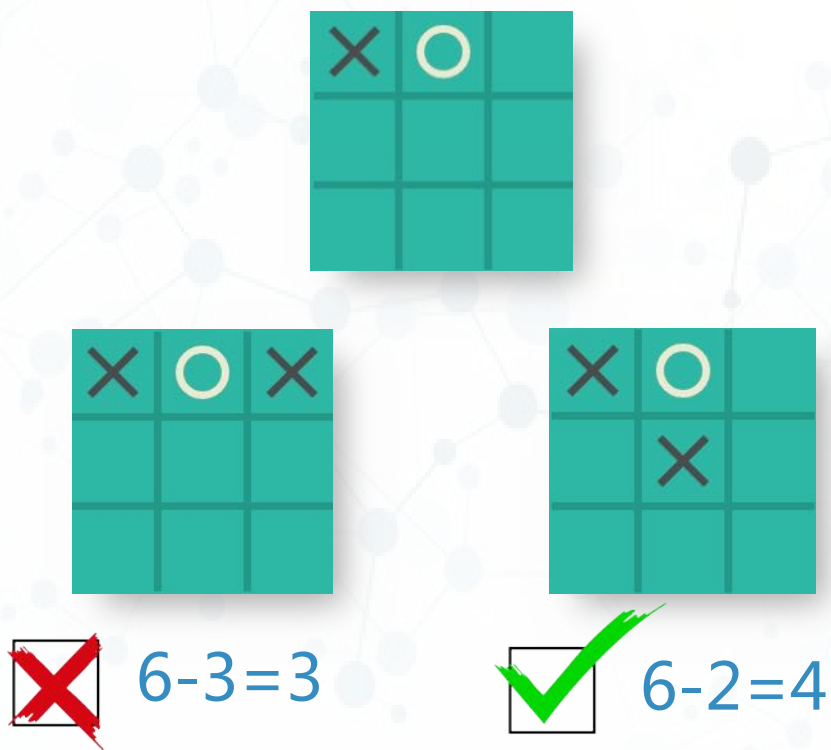
$$6-3=3$$



$$6-2=4$$

# 选取策略

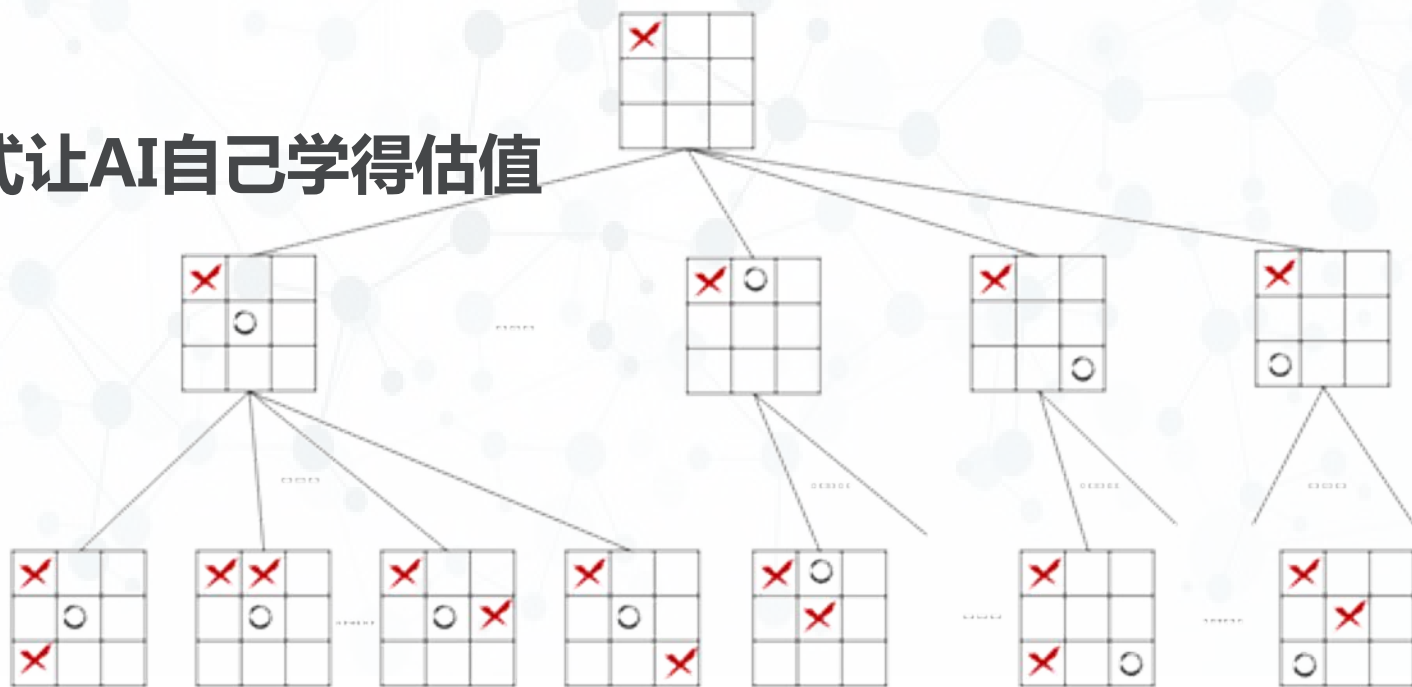
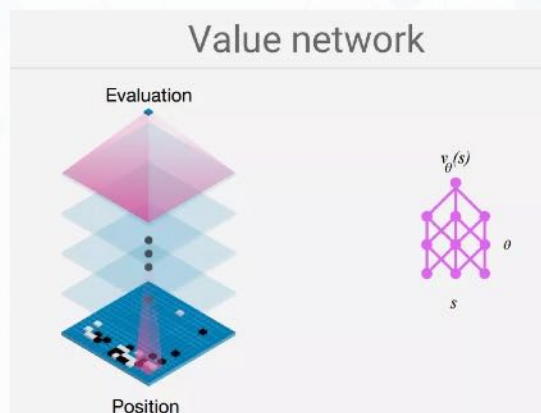
- › 根据估值函数选择最优策略
- › 最佳行动就是能够使得下一个状态的评估值**最大**的行动





# 进一步改进

- 采用**多步**搜索策略
- 提高搜索的**深度**，尽量接近搜索过程的终止状态
- 搜索配合**剪枝**提高效率
- 改进估值函数
- 通过神经网络等方式让AI自己学得估值



# 最大最小值法

## › 零和游戏中

玩家在可选的选项中选择将其优势**最大化**的选择  
也就是说要选择令**对手**优势**最小化**的方法

## › 回合制的游戏

双方都很**聪明**，都会采用最优策略

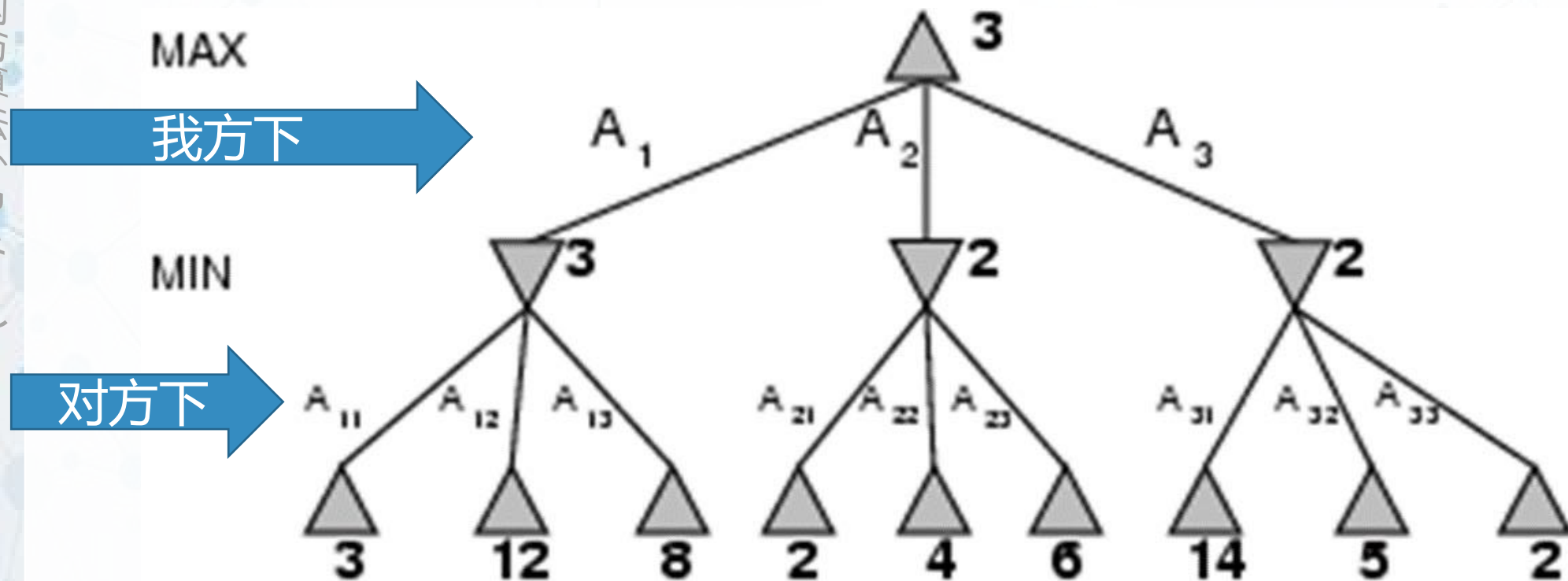
用**最大最小值法**统一表示

轮到我方下，选择我方的**最大**估值

轮到对方下，选择我方的**最小**估值

# minimax值

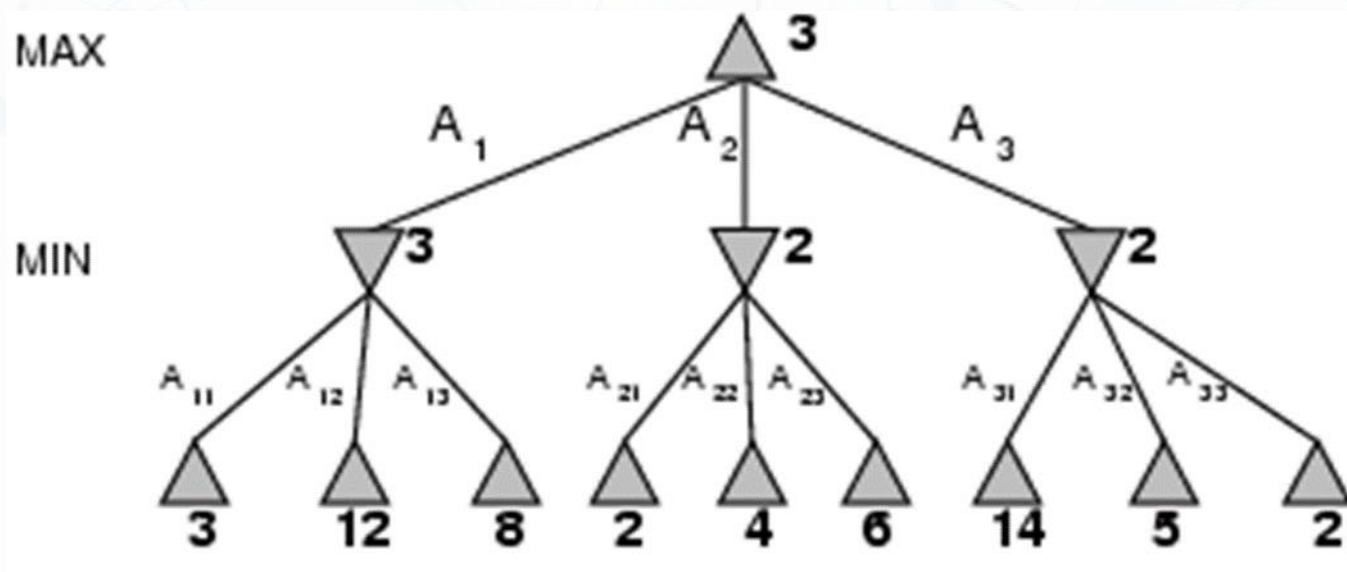
- 一个minimax决策树，包括max结点、min结点和终止结点





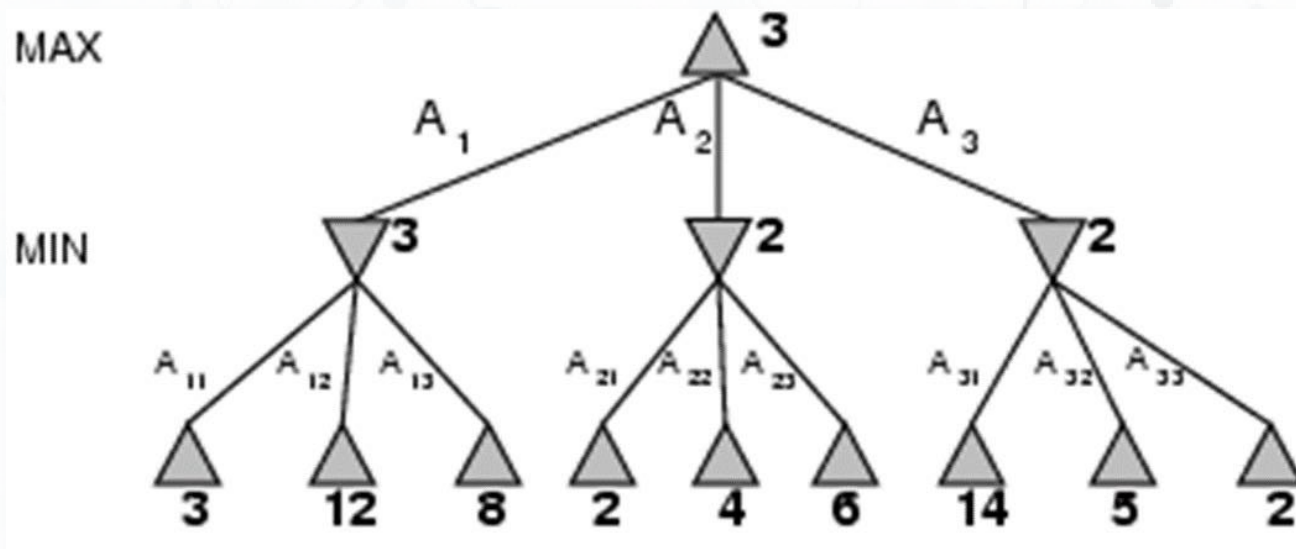
# minimax值：表示决策树上结点的估值

- 对于终止结点，minimax值等于直接对局面的估值
- 对于MAX结点，选择minimax值最大的子结点的值作为MAX结点的值
- 对于MIN结点，选择minimax值最小的子结点的值作为MIN结点的值

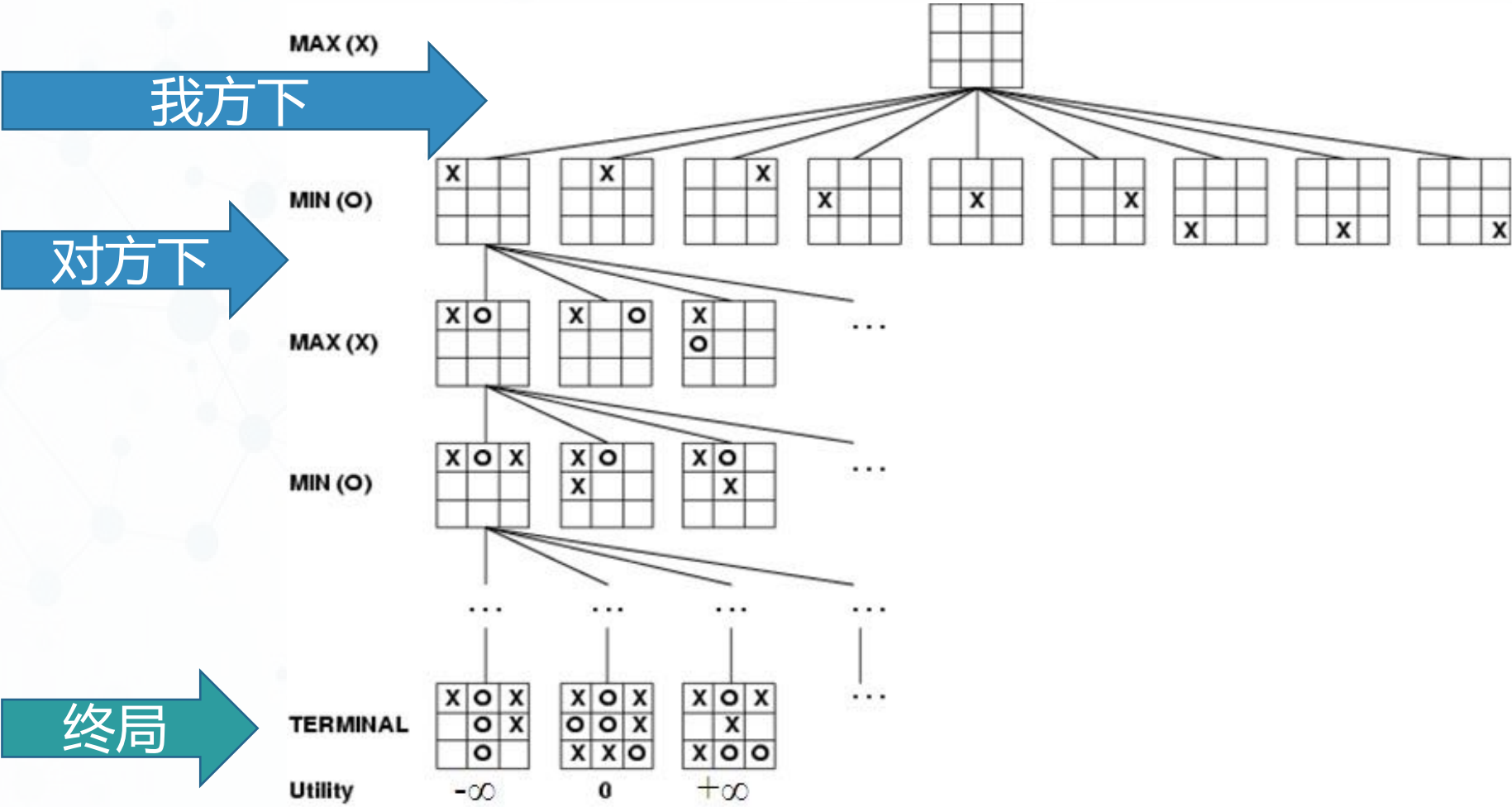


# 算法过程

1. 构建决策树
2. 将评估函数应用于终局的**叶子**结点
3. 自底向上计算每个结点的minimax值
4. 从根结点选择minimax值**最大**的分支，作为**行动策略**

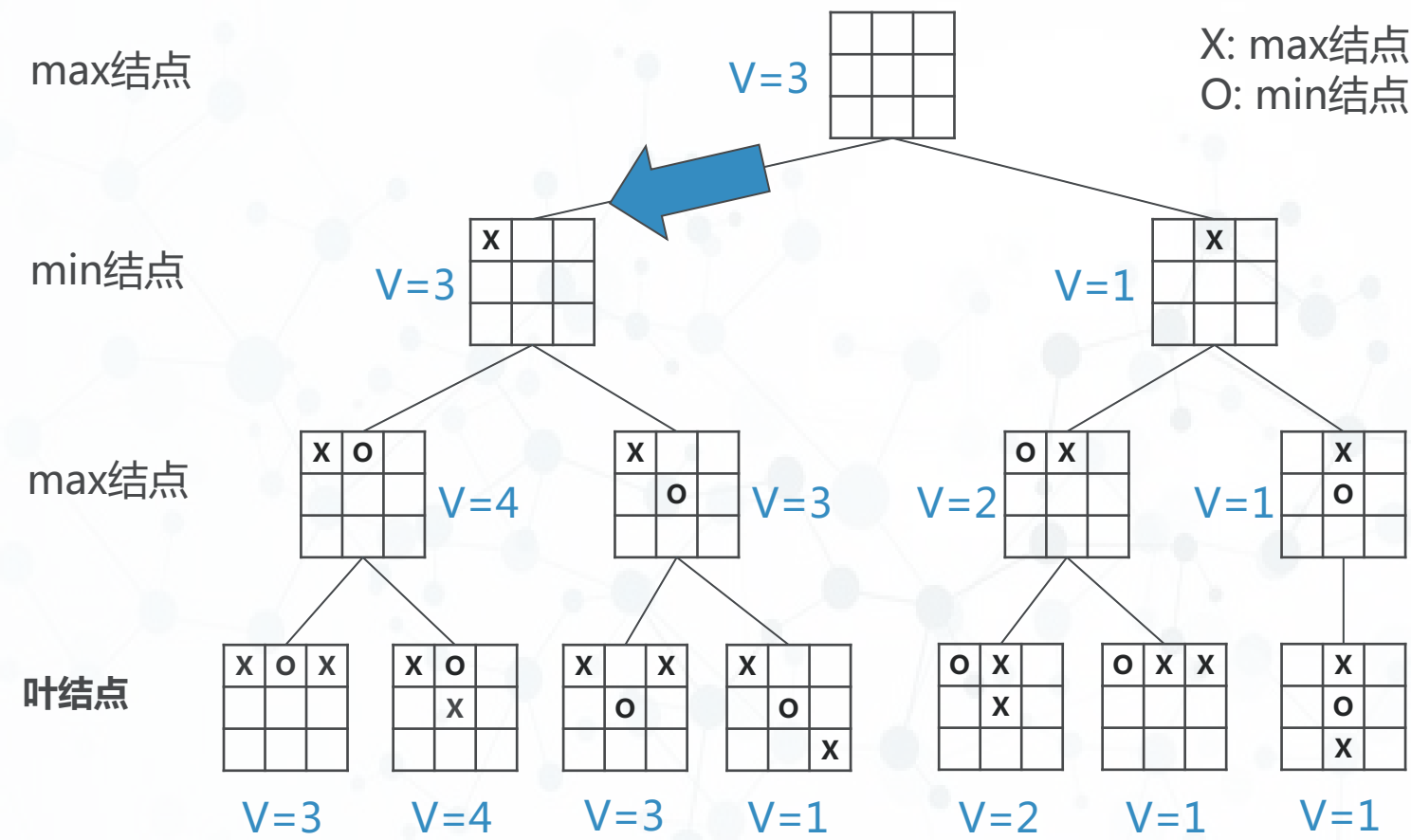


# 构建决策树，叶子结点估值



# 算法过程演示

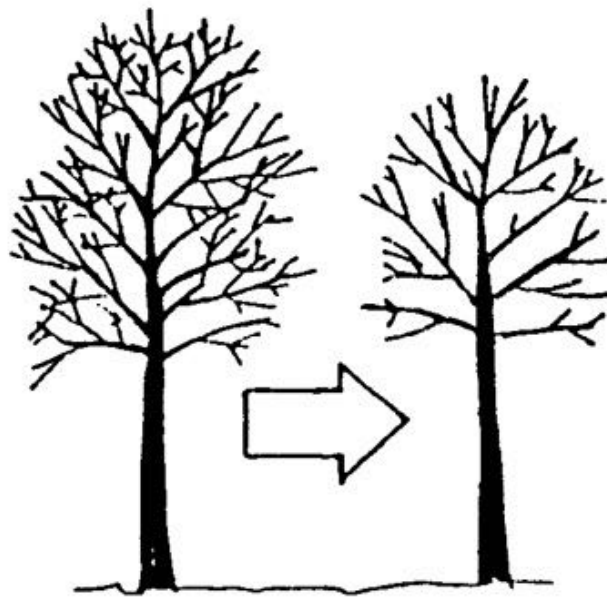
数据结构与算法 (Python)





# Minimax的优化：剪枝

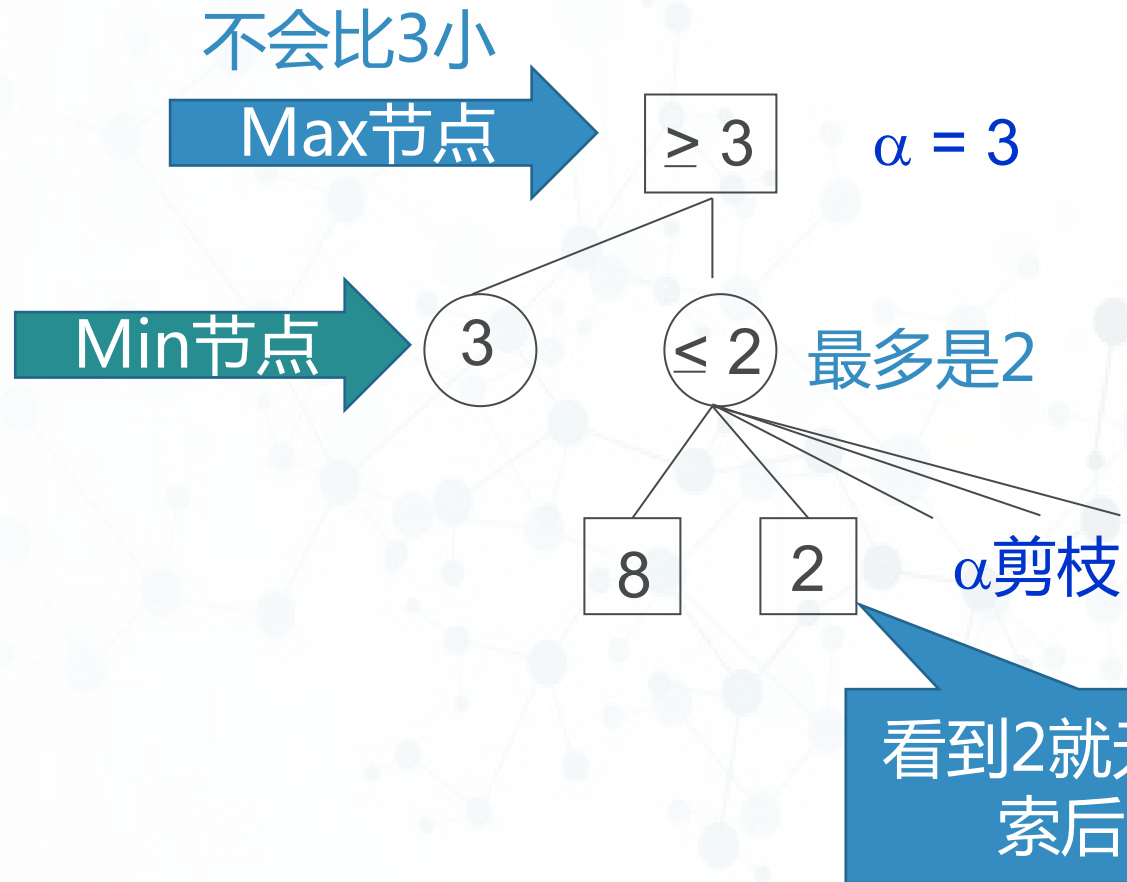
- › Minimax需要展开**整个**决策树
- › 对于局面复杂的问题，需要**很大**的空间
- › 有**部分结点**跟最后的结果**无关**，无需展开局面和计算估值
- › 不计算这些结点可节省算法搜索的时间
- › 去掉这些节点的过程叫做“剪枝”



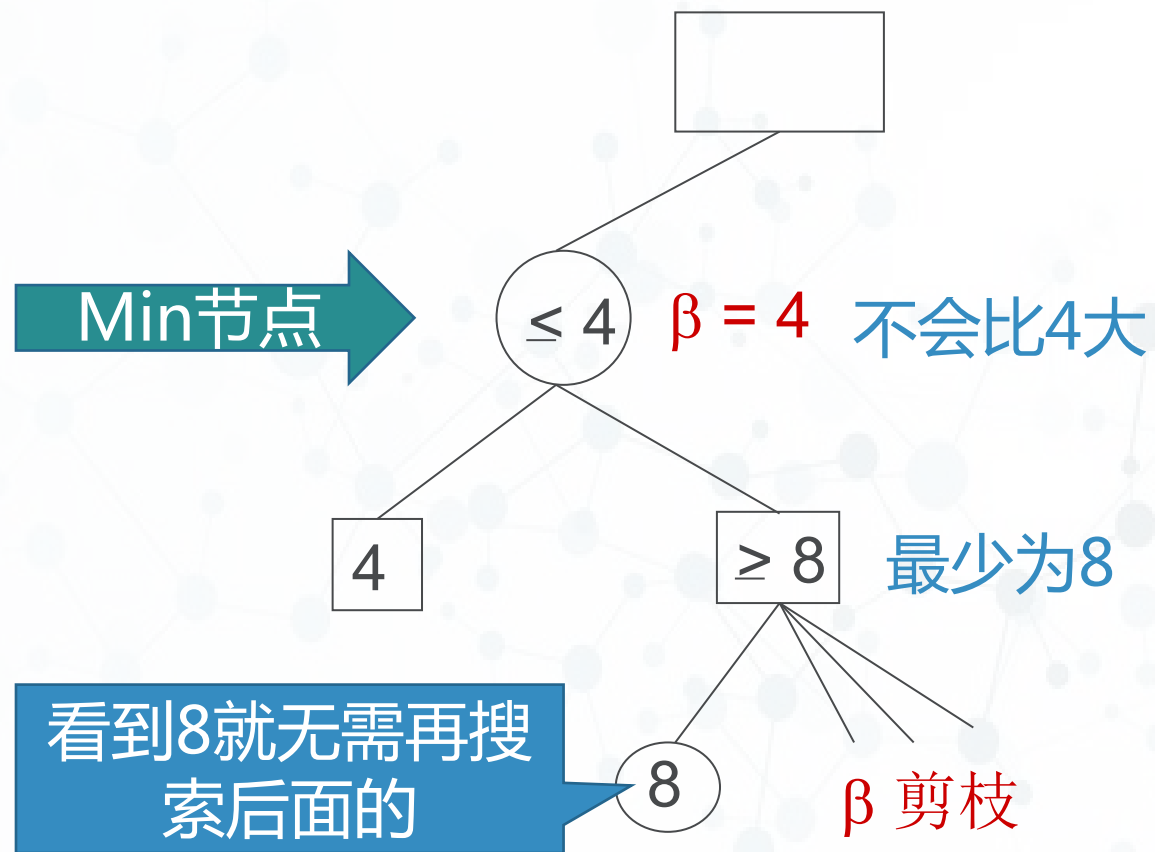
# Alpha-Beta剪枝

- › **加速** minimax 搜索过程
- › 每个结点存储局面估值之外，还存储**可能取值**的上下界
- › 估值：minimax 值
- › 下界（最小可能值）：Alpha 值
- › 上界（最大可能值）：Beta 值

# Alpha剪枝：搜索MAX节点子节点时



# Beta剪枝：搜索Min节点子节点时

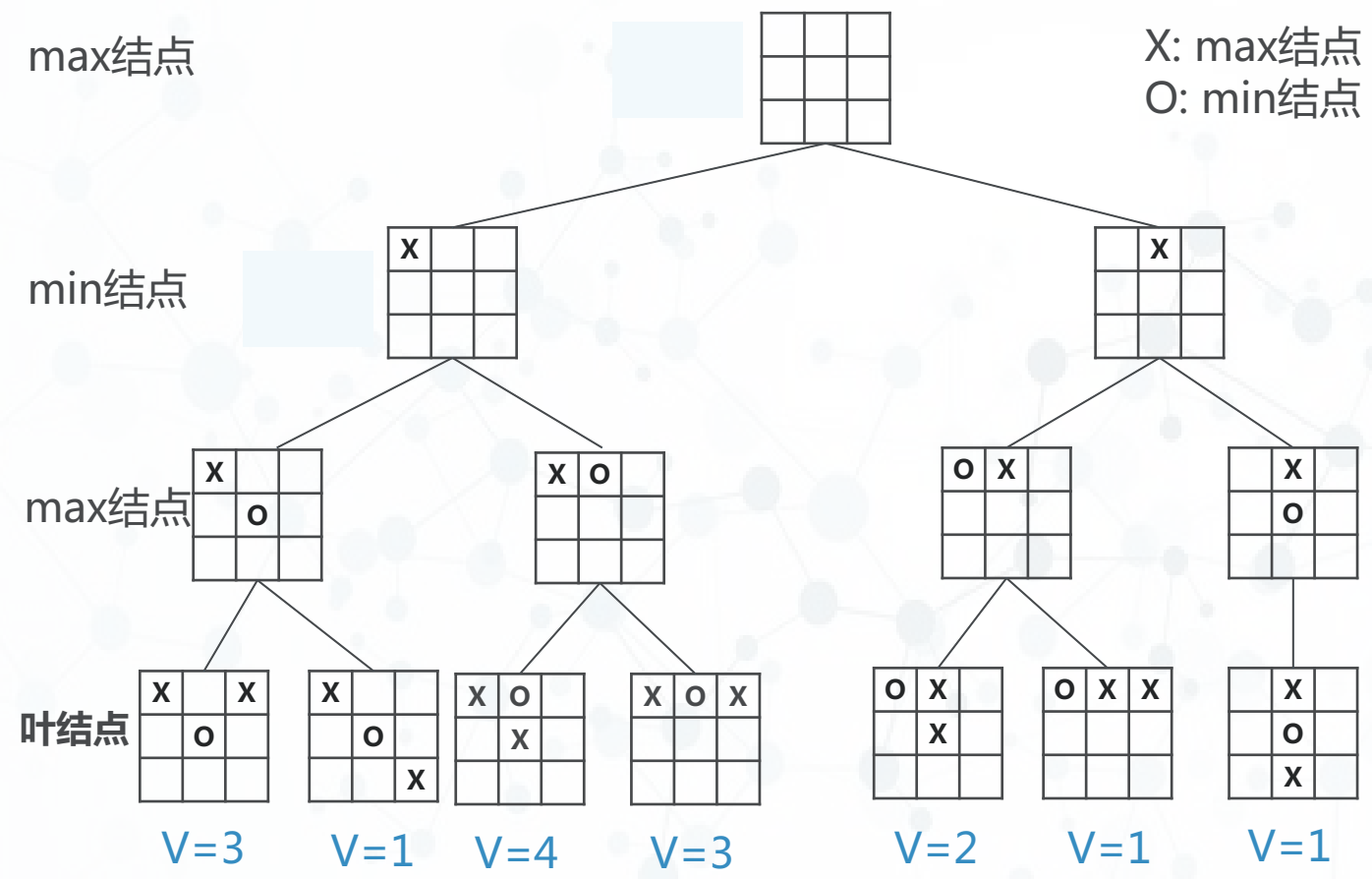




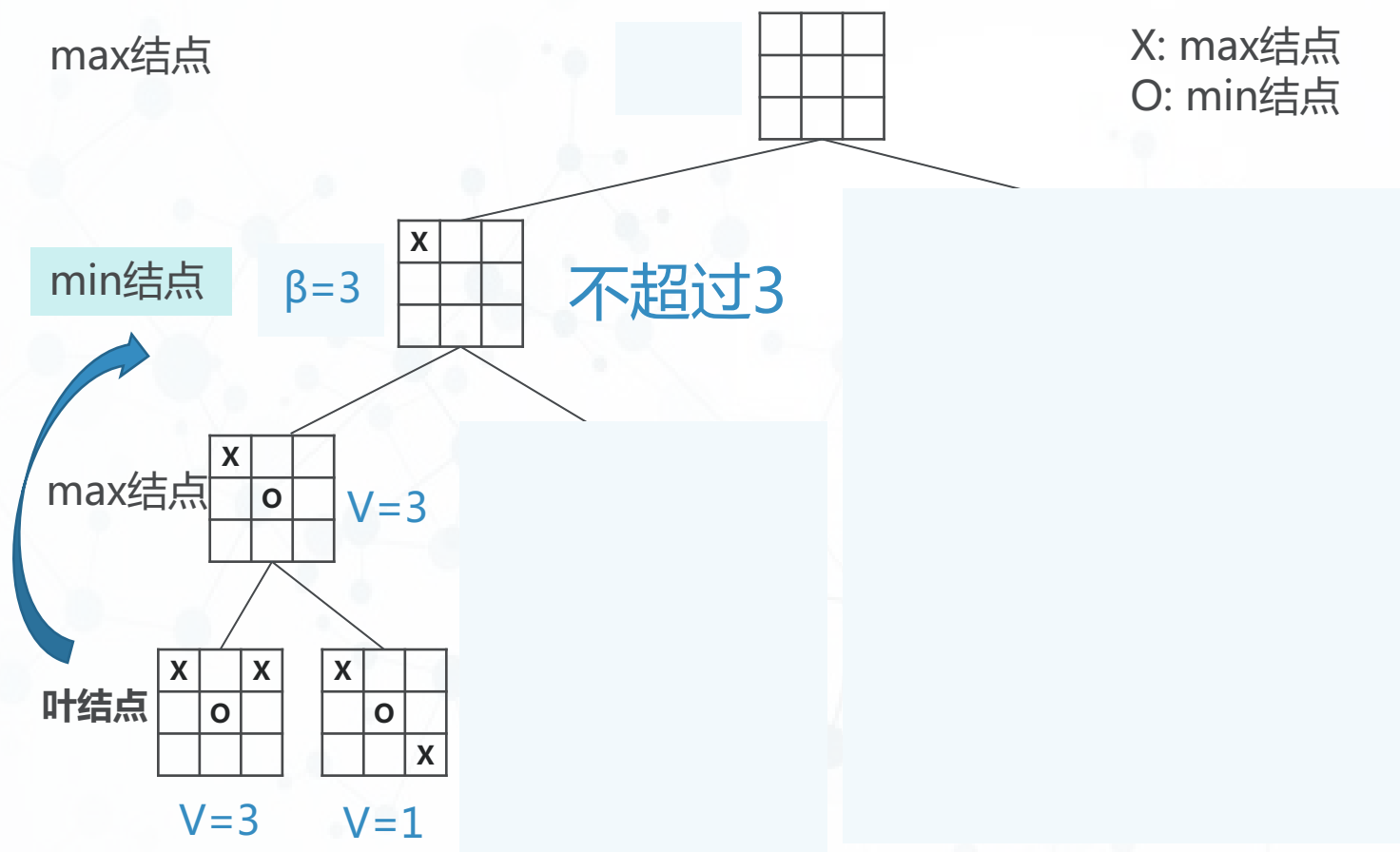
# 算法过程

1. 开始构建决策树
2. 将估值函数应用于叶子结点
3. 深度优先搜索，传递并更新 $\alpha$ 、 $\beta$ 、结点值  
Max结点更新  $\alpha$  值(下限)  
Min结点更新  $\beta$  值(上限)
4. 从根结点选择**评估值最大**的分支，作为行动策略

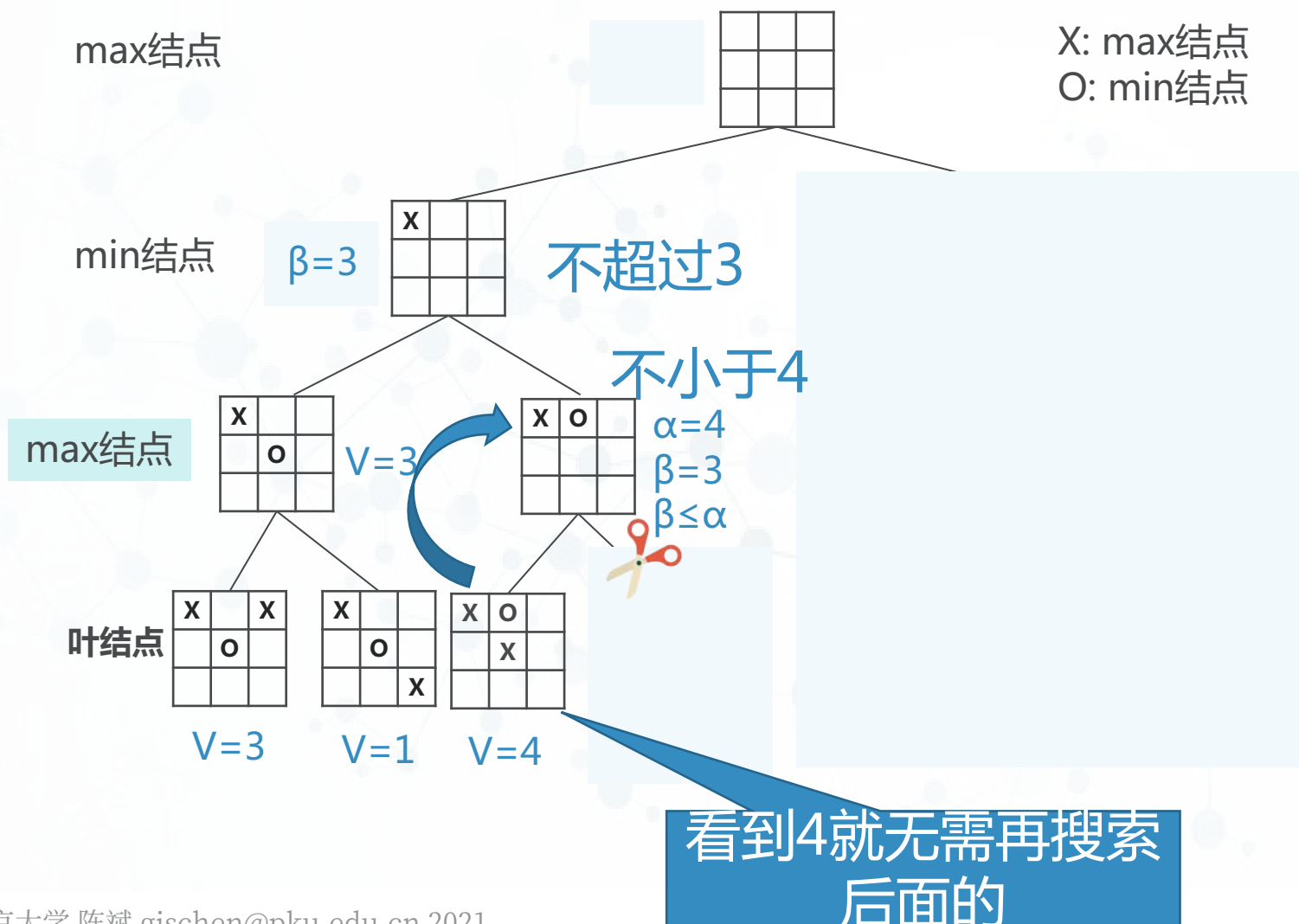
# 算法过程：未剪枝



# 算法过程

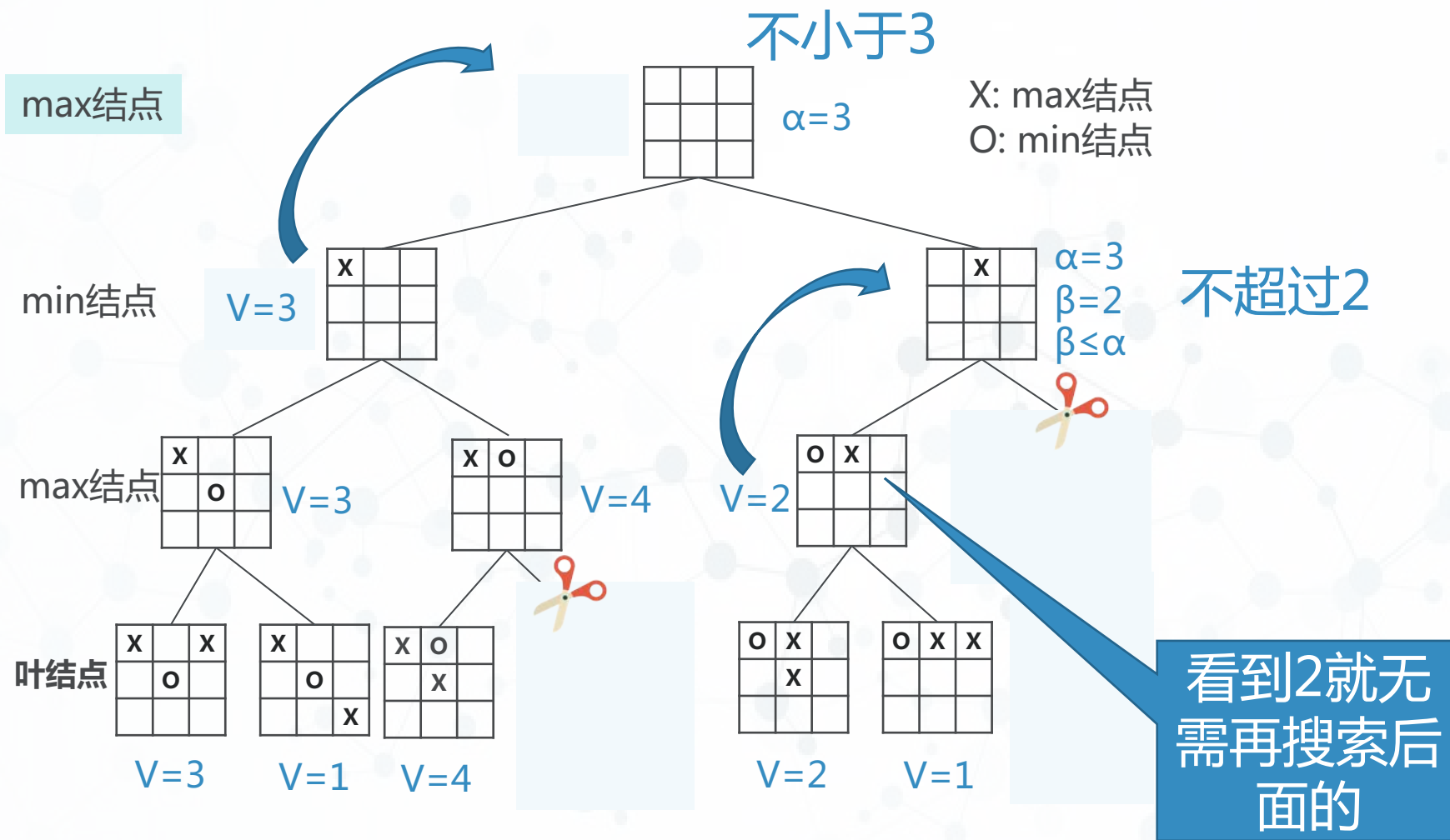


# 算法过程：Beta剪枝

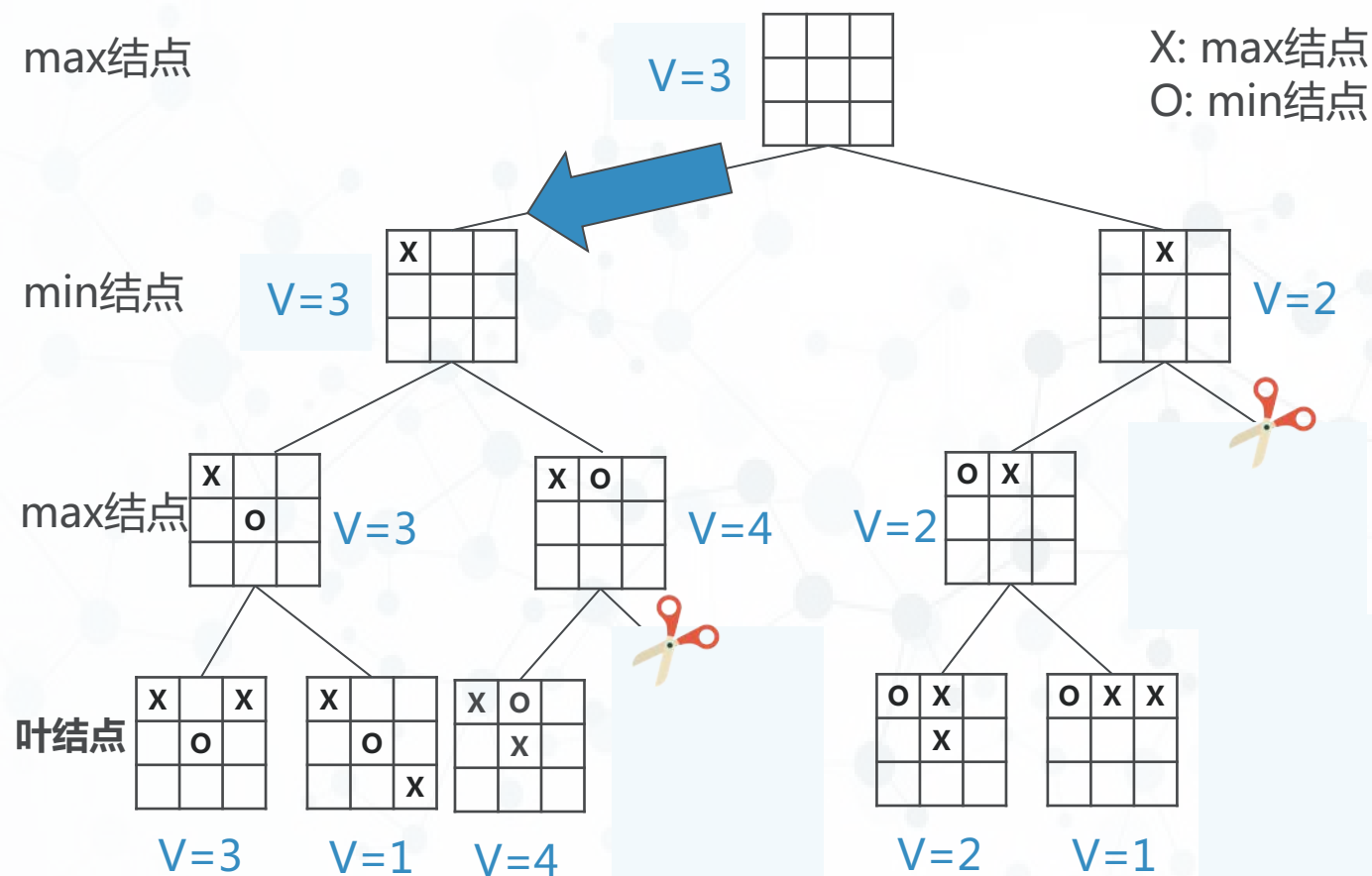




# 算法过程：Alpha剪枝



# 算法过程



# Alpha-Beta剪枝的局限性

- › 受到搜索次序的影响很大
- › 并不都能实现有效剪枝

# 启发式算法

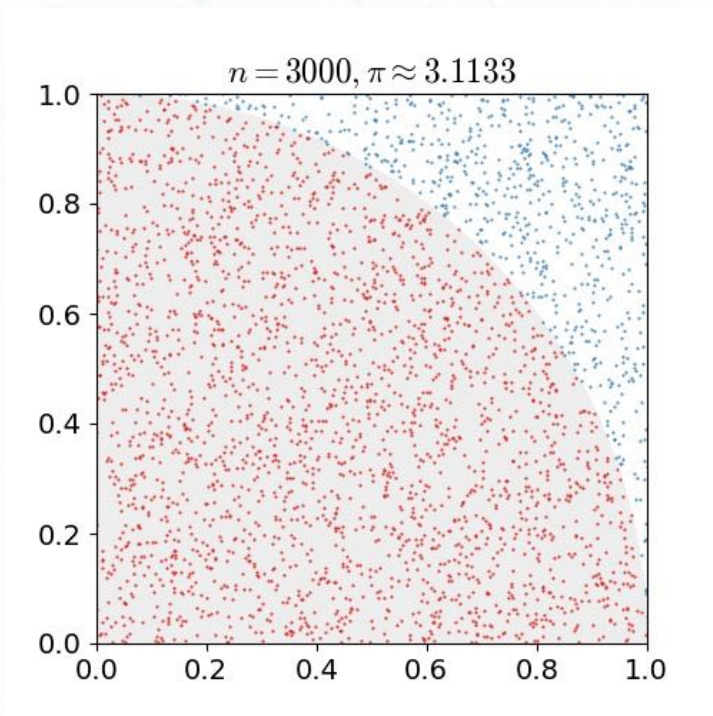
- 在搜索过程中，启发式算法被定义成一系列额外的规则
- 经验法则，利用一些特定的知识  
“高手怎么下，我也怎么下”
- 它常能发现很不错的解，但也没办法证明它不会得到较坏的解
- 它通常可在合理时间解出答案，但也没办法知道它是否每次都可以这样的速度求解





# 蒙特卡洛方法

- › 通过**随机采样**计算得到近似结果
- › 一种通用的计算方法

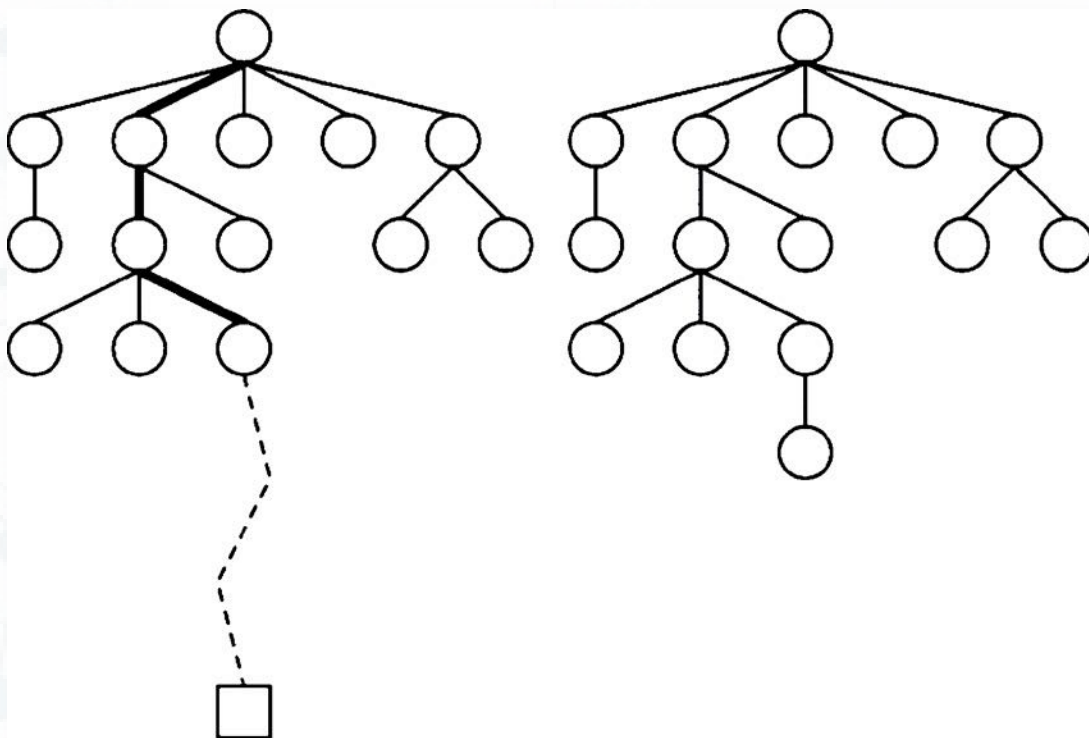


# 蒙特卡洛树搜索 (MCTS)

› 一种通过在决策空间中**随机采样**并根据结果构建决策树来寻找最优策略的方法

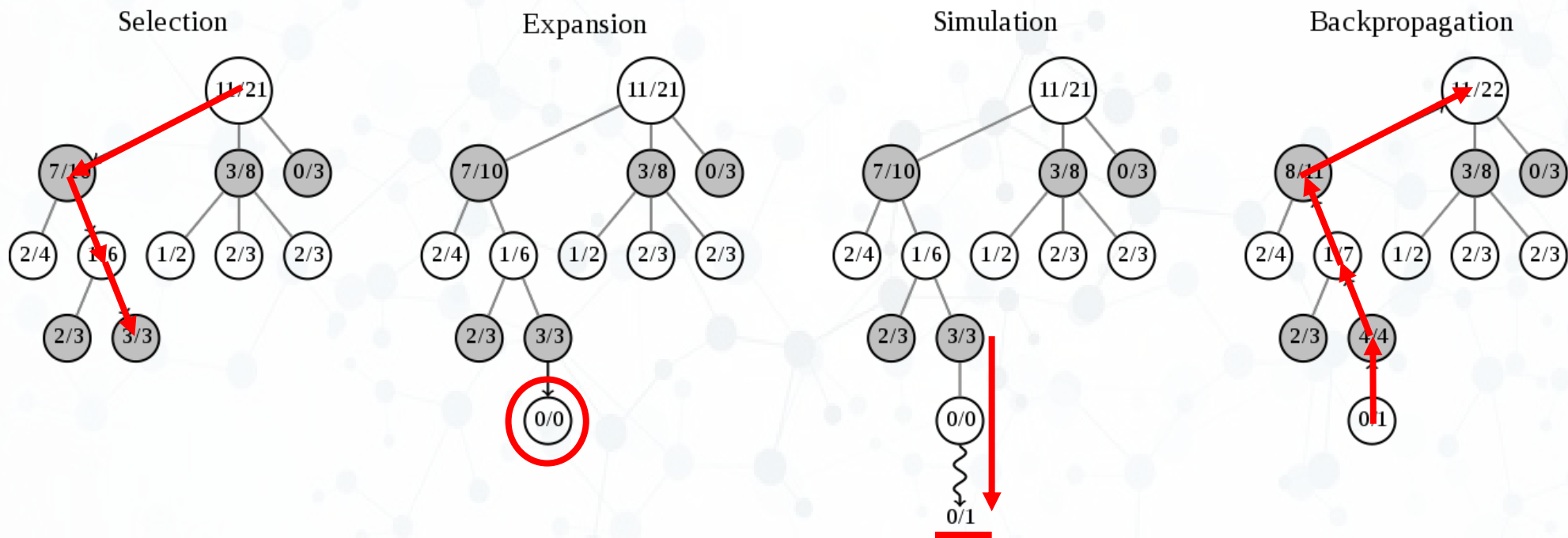
› 决策树的构建

选择、扩张、模拟、反馈



# MCTS例子

可以用来优化估值函数，以及得到更好的选择策略



# 小结

- › 棋局的估值函数
- › 决策树：最大最小法
- › Alpha-Beta剪枝
- › 启发式规则
- › 蒙特卡洛树搜索

# 下课！

