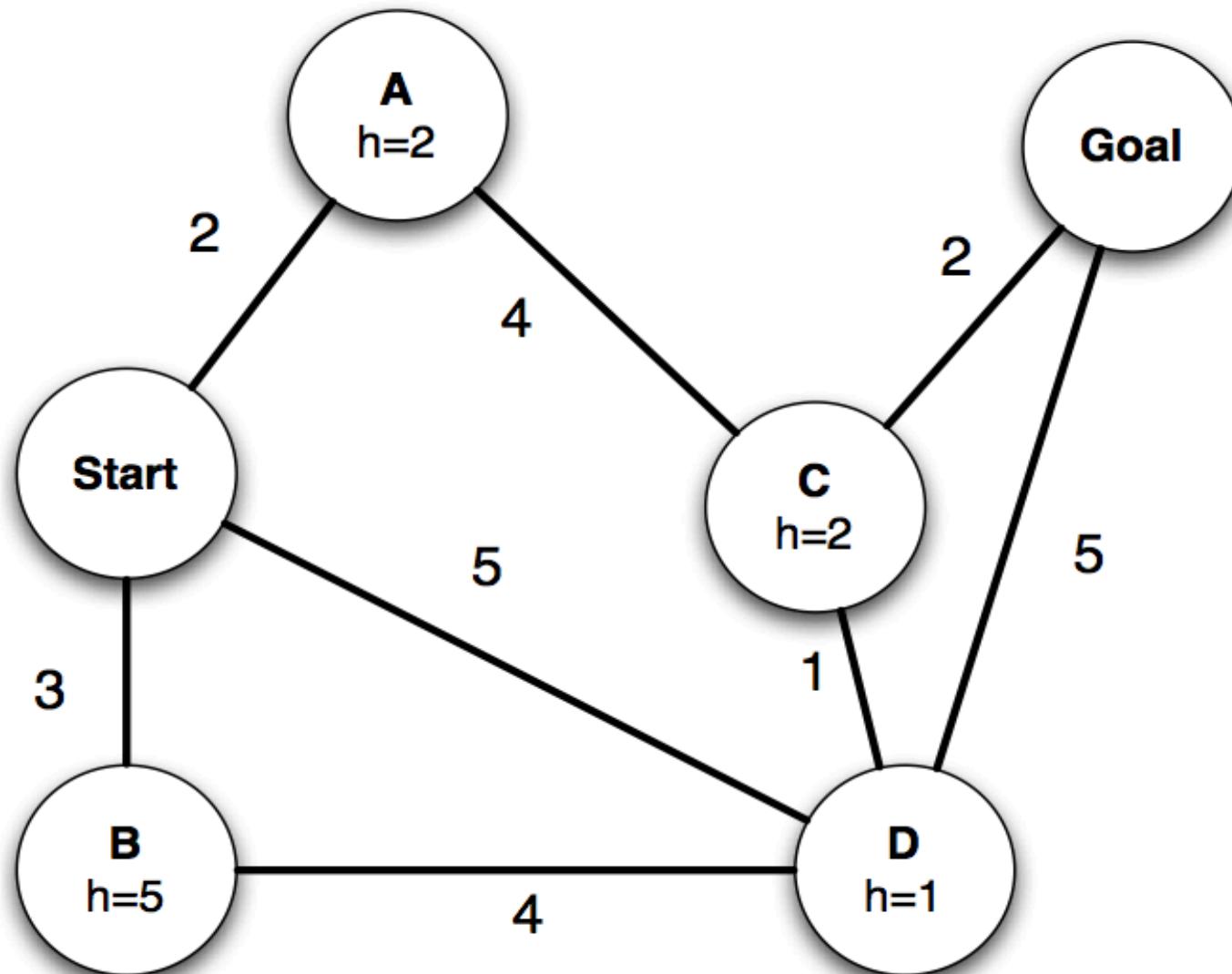


# 今天的内容

- ~~作业1讲解~~
- 小测验 (搜索算法应用)
- 对抗性搜索问题的复习
- 约束满足问题

请写出以下每种图搜索策略：

1. 节点被扩展的顺序（从Start开始，到Goal结束） 2. 返回的路径



- 1) 深度优先搜索
- 2) 广度优先搜索
- 3) 基于成本的统一搜索
- 4) 贪婪搜索（使用启发函数 $h$ 的值）
- 5) A\* 搜索（使用启发函数 $h$ 的值）

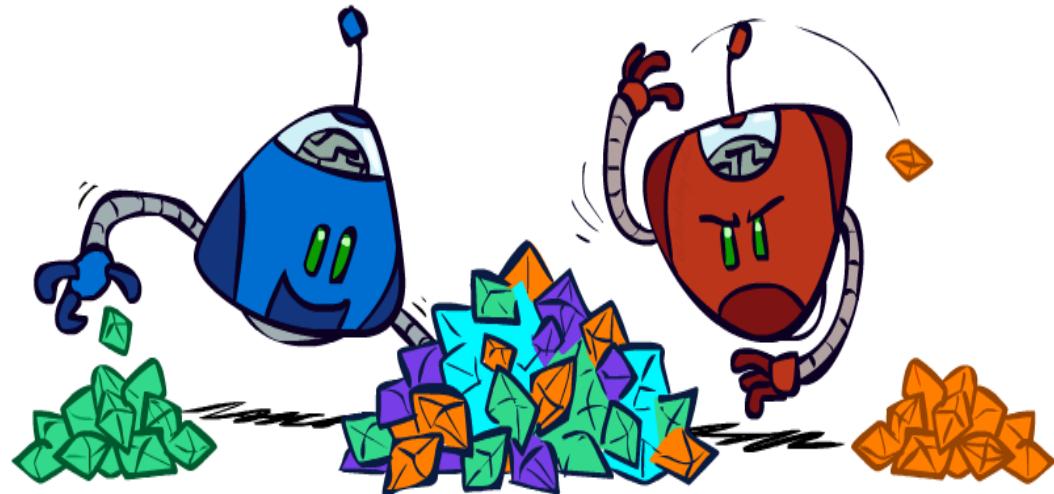
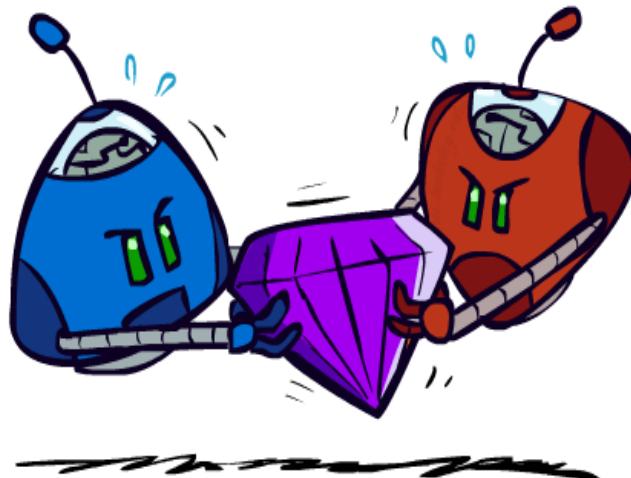
# 对抗性的搜索，复习

---

# 人工智能所研究的游戏比赛 (games)

- 标准的游戏是，确定的，全局可观察的，轮流行动的，两人的，零和的。
- 游戏问题的数学模型建立：
  - 初始状态:  $s_0$
  - 玩家:  $\text{Player}(s)$  显示在当前状态轮到哪一个玩家行动
  - 行动:  $\text{Actions}(s)$  当前轮次的玩家可能的移动
  - 状态转换模型:  $\text{Result}(s,a)$
  - 终局状态检测:  $\text{Terminal-Test}(s)$  是否是终局
  - 终局得分:  $\text{Utility}(s,p)$  玩家  $p$  的得分
    - 或只用  $\text{Utility}(s)$  代表游戏一开始时最先移动的玩家的得分

# 零和游戏



- 零和游戏

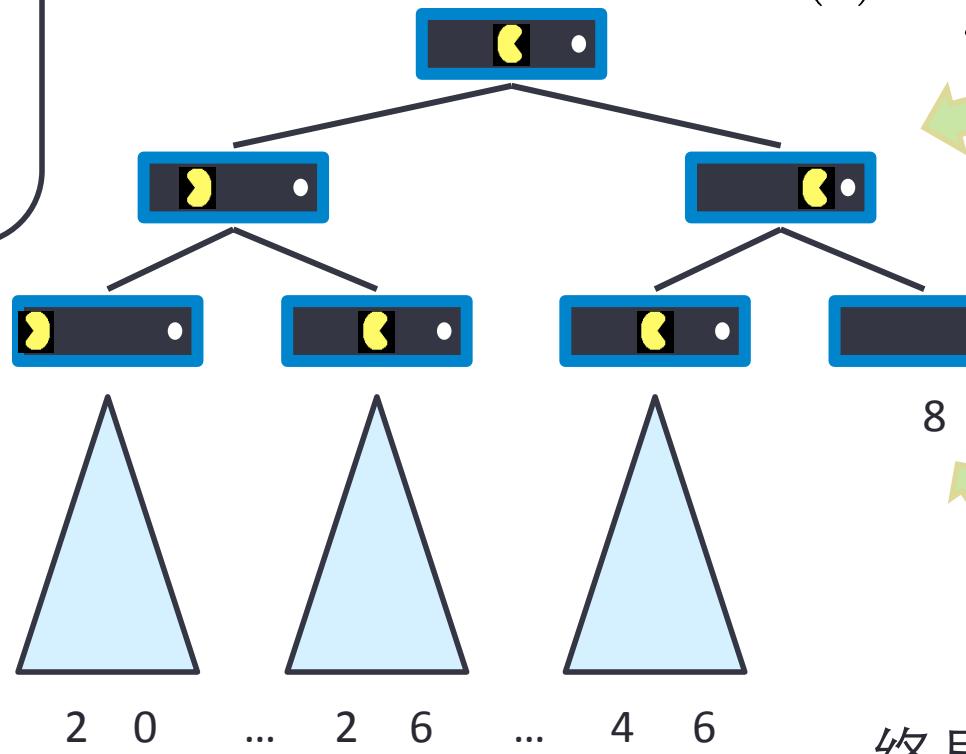
- 智能体竞争实现相反的利益
- 一方 **最大化**这个利益, 另一方 **最小化**它

- 通常游戏

- 智能体有 **独立的** 利益
- 合作, 竞争, 联盟等相互关系, 都有可能

# 单一智能体搜索树: 状态值

一个状态的  
值: 从这个状  
态往下可能  
达到的最大  
利益值



中间状态:  
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

终局状态的值已知

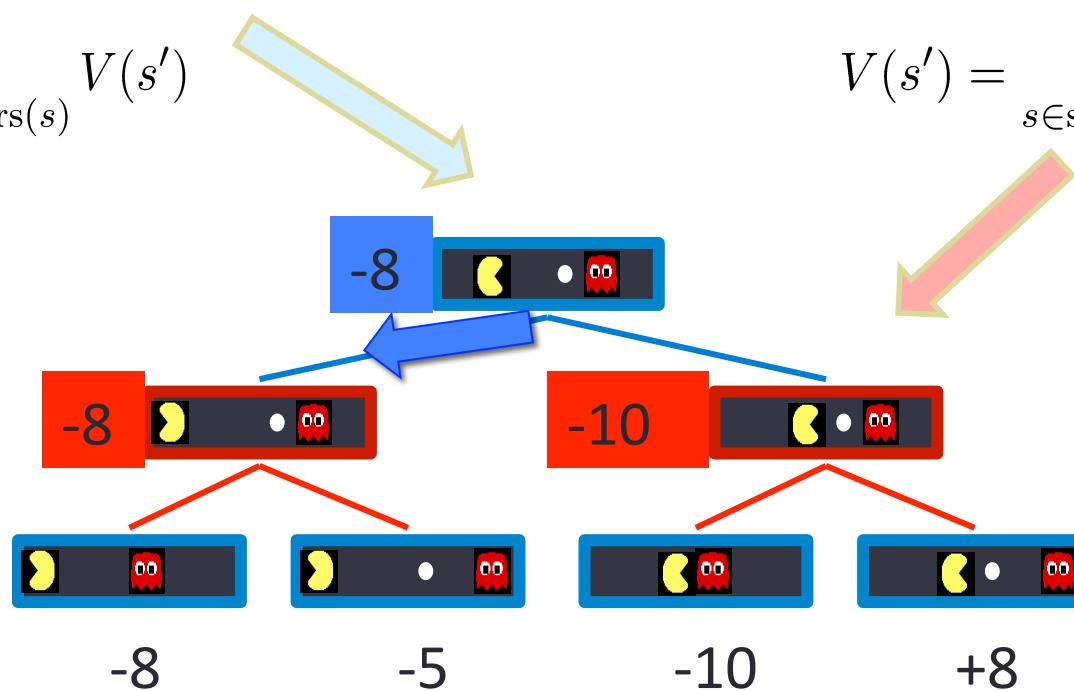
# 最小最大值 (Minimax values)

MAX 节点: 智能体控制下的状态

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

MIN 节点: 对手控制下的节点

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



终局状态的值是已知的

# 最小最大算法(Minimax) 实现

深度优先搜索

Function 最小最大-决策( $s$ ) returns 一个行动

return 行动  $a$  in  $\text{Actions}(s)$  , 它能导致最大的  
最小-值(Result( $s,a$ ))的返回值

Function 最大-值( $s$ ) returns 一个值  
If 终局-检测( $s$ ) then return Utility( $s$ )  
初始化  $v = -\infty$   
for each  $a$  in  $\text{Actions}(s)$ :  
     $v = \max(v, \text{最小-值}(\text{Result}(s,a)))$   
return  $v$

Function 最小-值( $s$ ) returns 一个值  
If 终局-检测( $s$ ) then return Utility( $s$ )  
初始化  $v = +\infty$   
for each  $a$  in  $\text{Actions}(s)$ :  
     $v = \min(v, \text{最大-值}(\text{Result}(s,a)))$   
return  $v$

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

# 另一种实现方法

Function 最小最大-决策( $s$ ) returns 一个行动

return 行动  $a$  in  $\text{Actions}(s)$  , 它能导致最大的  
 $\text{value}(\text{Result}(s,a))$  的返回值



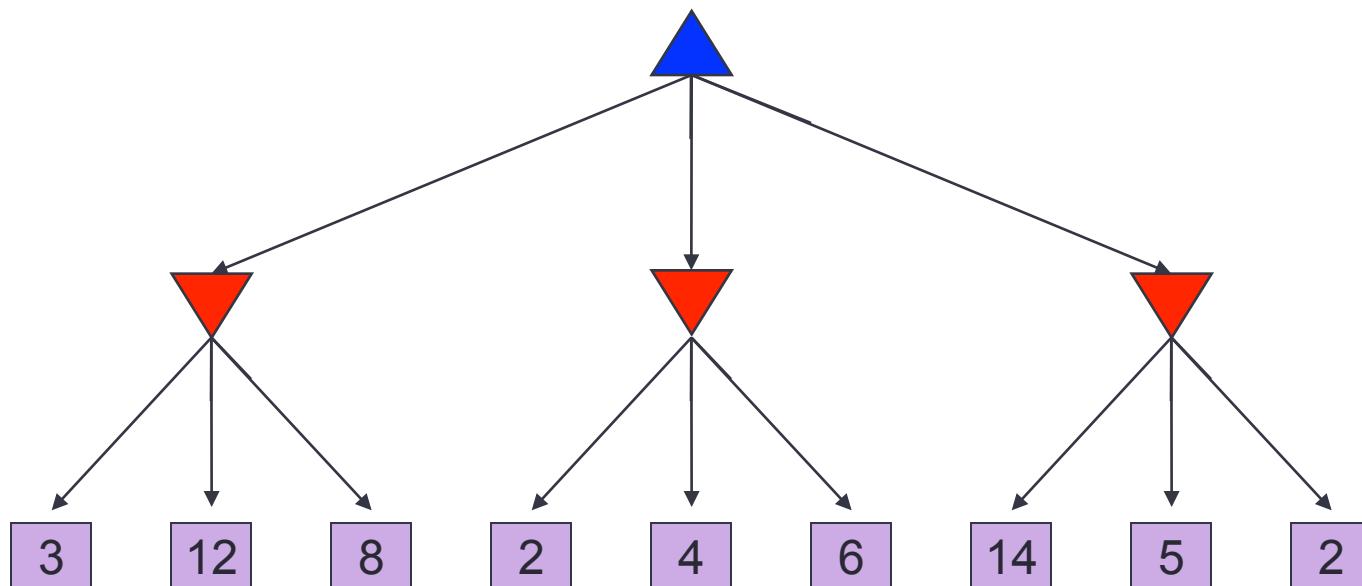
function  $\text{value}(s)$  returns 一个值

If 终局-检测( $s$ ) then return  $\text{Utility}(s)$

if  $\text{Player}(s) = \text{MAX}$  then return  $\max_{a \text{ in } \text{Actions}(s)} \text{value}(\text{Result}(s,a))$

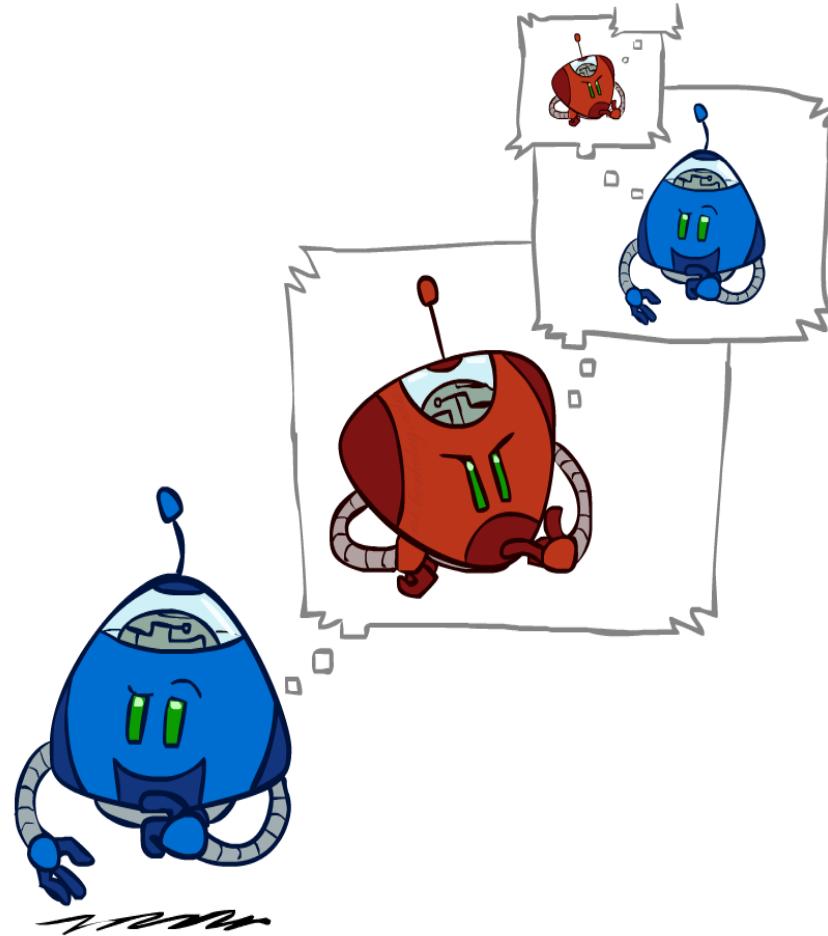
if  $\text{Player}(s) = \text{MIN}$  then return  $\min_{a \text{ in } \text{Actions}(s)} \text{value}(\text{Result}(s,a))$

# 最小最大值 (Minimax) 举例



# Minimax 的效率

- Minimax 的效率?
  - 深度优先穷尽搜索
  - 时间复杂度:  $O(b^m)$
  - 空间复杂度:  $O(bm)$
- 举例: 国际象棋,  $b = 35, m = 100$ 
  - 找到准确解, 不可行
  - 有必要探索整棵树吗?

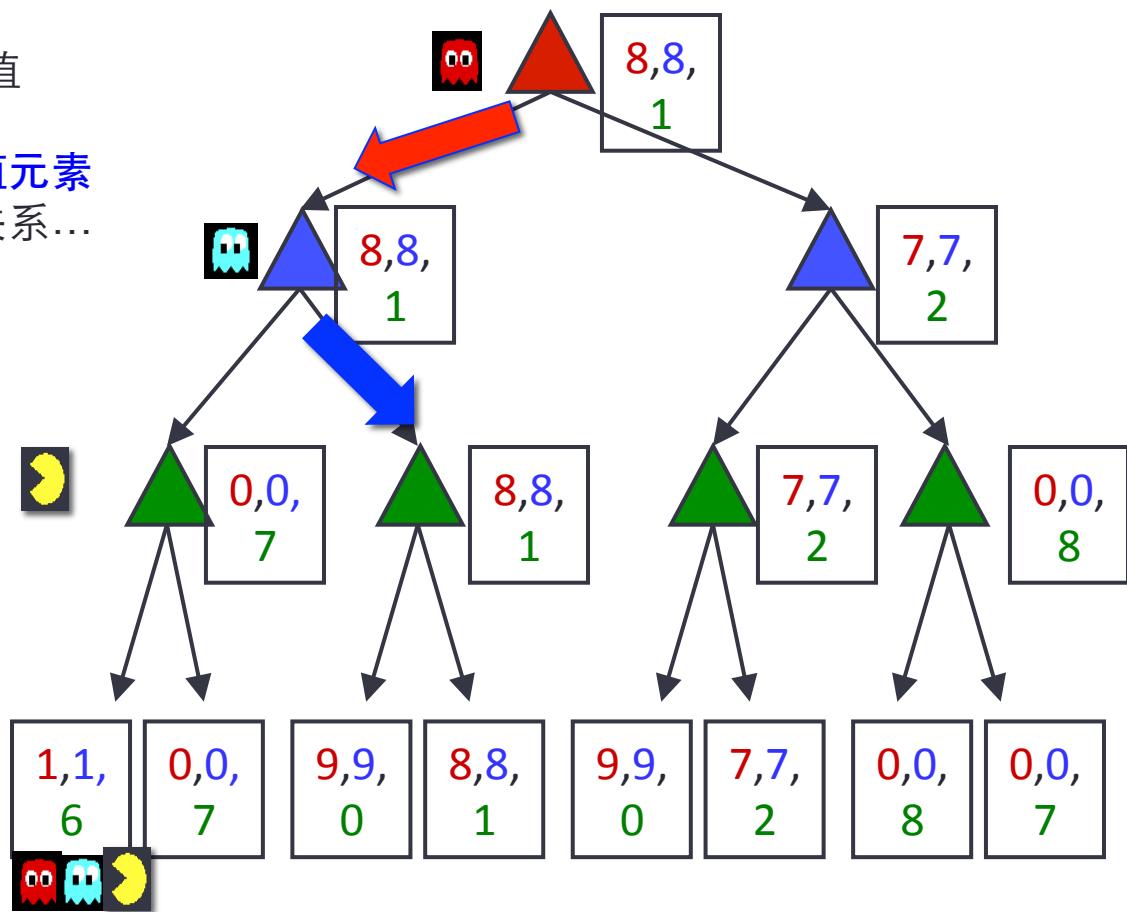
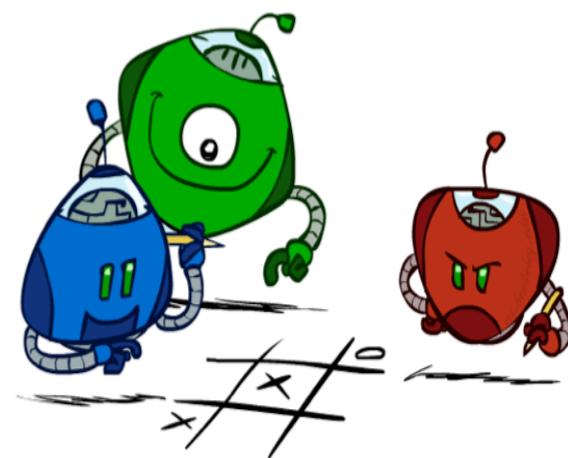


# 普遍化的最小最大值法 (minimax)

- 如果不是零和游戏，或有超过两个玩家？

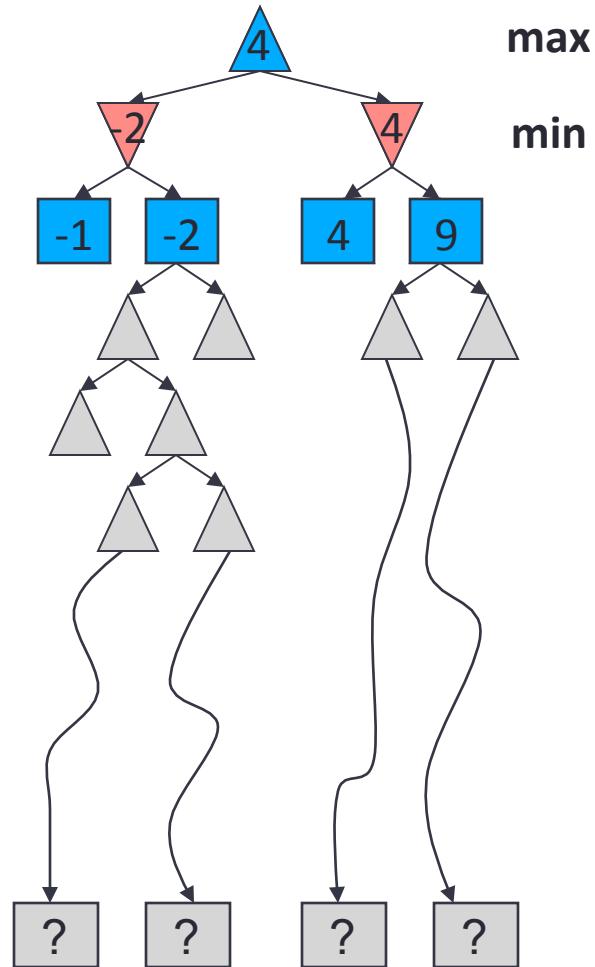
- 普遍情况：

- 终局（叶结点）的值是一组值
- 中间节点的值也是一组值
- 每个玩家最大化自己的利益值元素**
- 能够动态产生协作和竞争等关系...



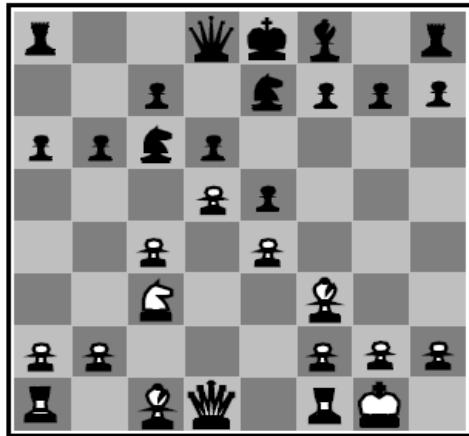
# 资源局限

- 问题: 现实中, 几乎不能搜索到叶节点!
- 办法之一: 有界搜索加预测
  - 搜索只到预定深度层次
  - 使用 **评估函数** 预测搜索边界节点的值
- 失去了最优解的保证
- 搜索的层次越多结果就越不相同
- 例如:
  - 假设计算时间为100 秒, 每秒能探索1万个节点
  - 所以每步能检查1百万个节点
  - 在国际象棋中,  $b \sim 35$ , 搜索大致能达到搜索树的第4层 – 还是不够好

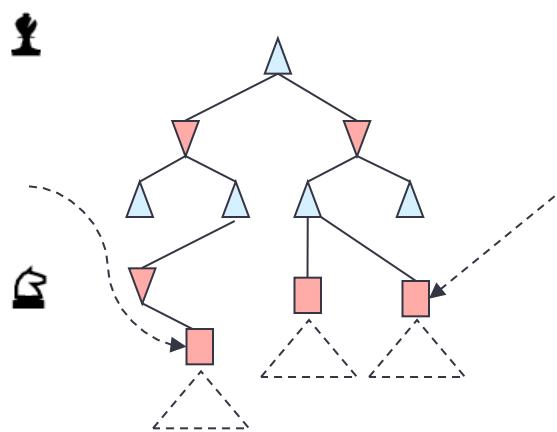


# 评估函数

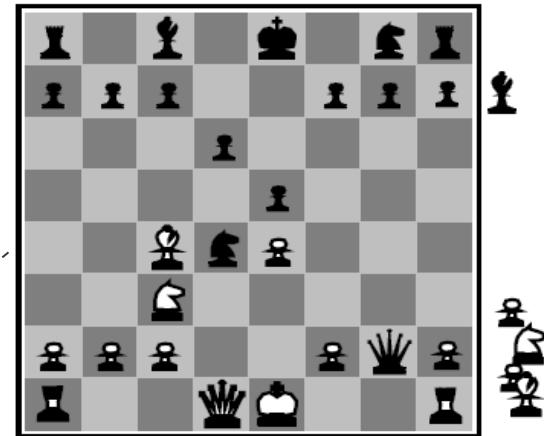
- 用来给非终局状态打分，在一个深度有限搜索中。



Black to move



White slightly better

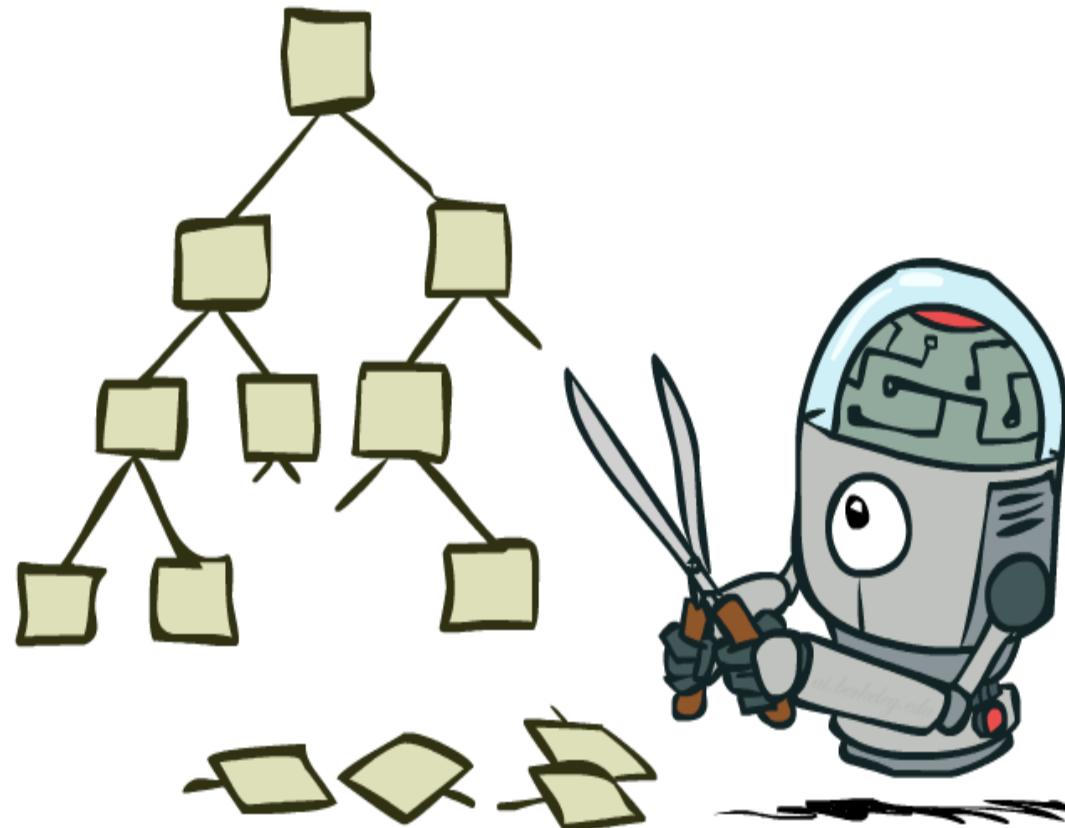


White to move

Black winning

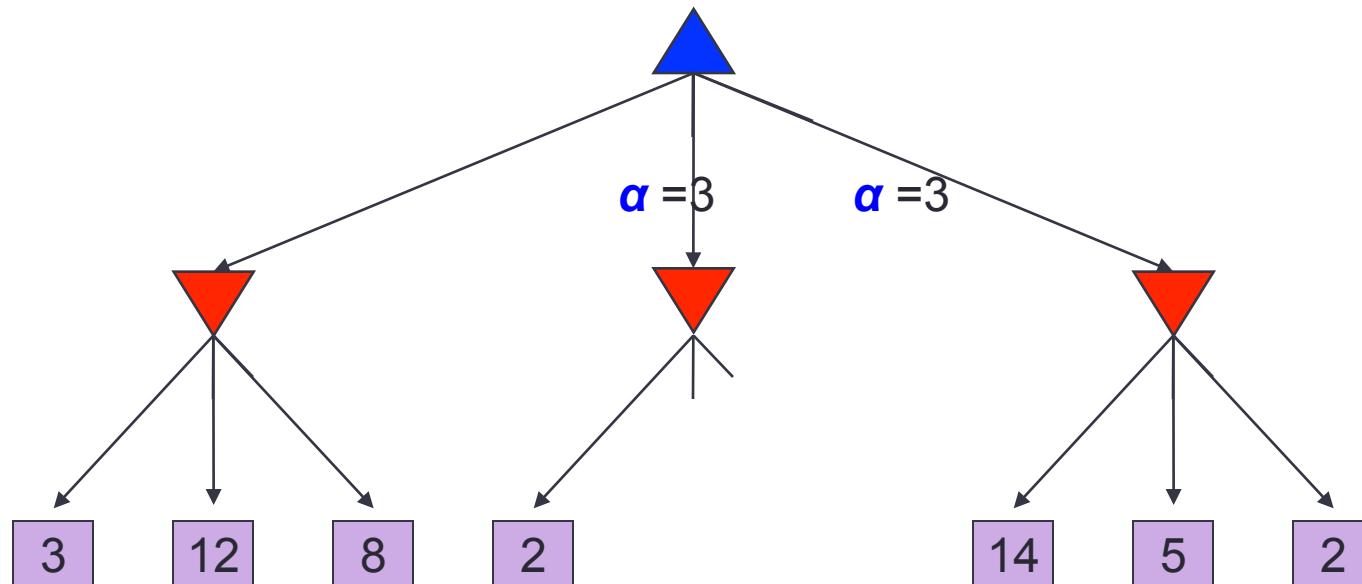
- 理想函数: 返回这个状态的实际最小最大值
- 实践中: 特征函数值的加权线性和:
  - $\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$
  - 例如  $w_1 = 9$ ,  $f_1(s) = (\text{白皇后数量} - \text{黑皇后数量})$ , 等。
- 评估函数应作用在 **沉寂** 状态, 即评估值在其后继状态相对变化不大。

# 博弈（游戏）树的剪枝



# Alpha-Beta 剪枝例子

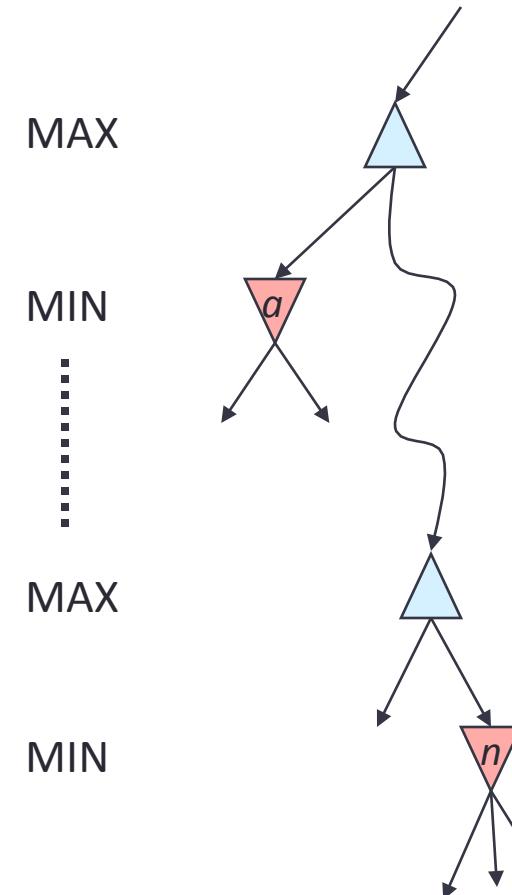
$\alpha$  = 在当前路径上所有MAX节点中最大的值



节点产生的顺序与结果有关: 可能导致不同的可被剪掉的节点数量

# Alpha-Beta 剪枝

- 假定修剪 MIN 节点的子节点
  - 假设正在计算节点  $n$  的 最小-值
  - $n$  节点的值在检查其子节点的过程中逐渐减小
  - 令  $\alpha$  是从根到当前节点路径上的 MAX 分支节点所能达到的最大值
  - 如果  $n$  的当前值比  $\alpha$  小，那么路径上的 MAX 分支节点将会避开这条路径，所以我们可以剪掉(不去检查)  $n$  的其他子节点
- 对 MAX 节点剪枝是对称的
  - 令  $\beta$  是从根到当前节点路径中的 MIN 分支节点所能达到的最小值
  - 如果  $n$  当前值比  $\beta$  大，则可以剪掉  $n$  的其他子节点 (不去检查)



# Alpha-Beta 剪枝算法实现

```
def alpha-beta-search(state) return 一个行动  
v <- max-value(state, -∞, +∞)  
return 导致值为 v 的行动
```

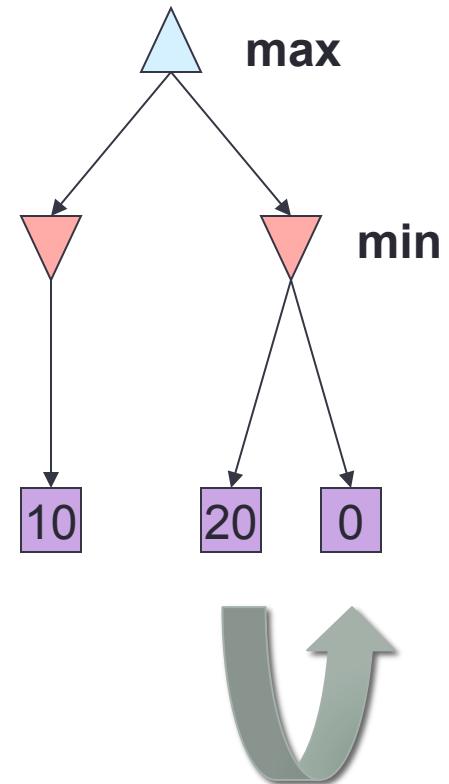
α: MAX节点的最大值, 当前路径上  
β: MIN节点的最小值, 当前路径上

```
def max-value(state, α, β):  
if 终局-检测(state) return Utility(state)  
初始化 v = -∞  
for each a in ACTIONS(state) do  
    v = max(v, min-value(Result(s,a),  
                           α, β))  
    if v ≥ β return v  
    α = max(α, v)  
return v
```

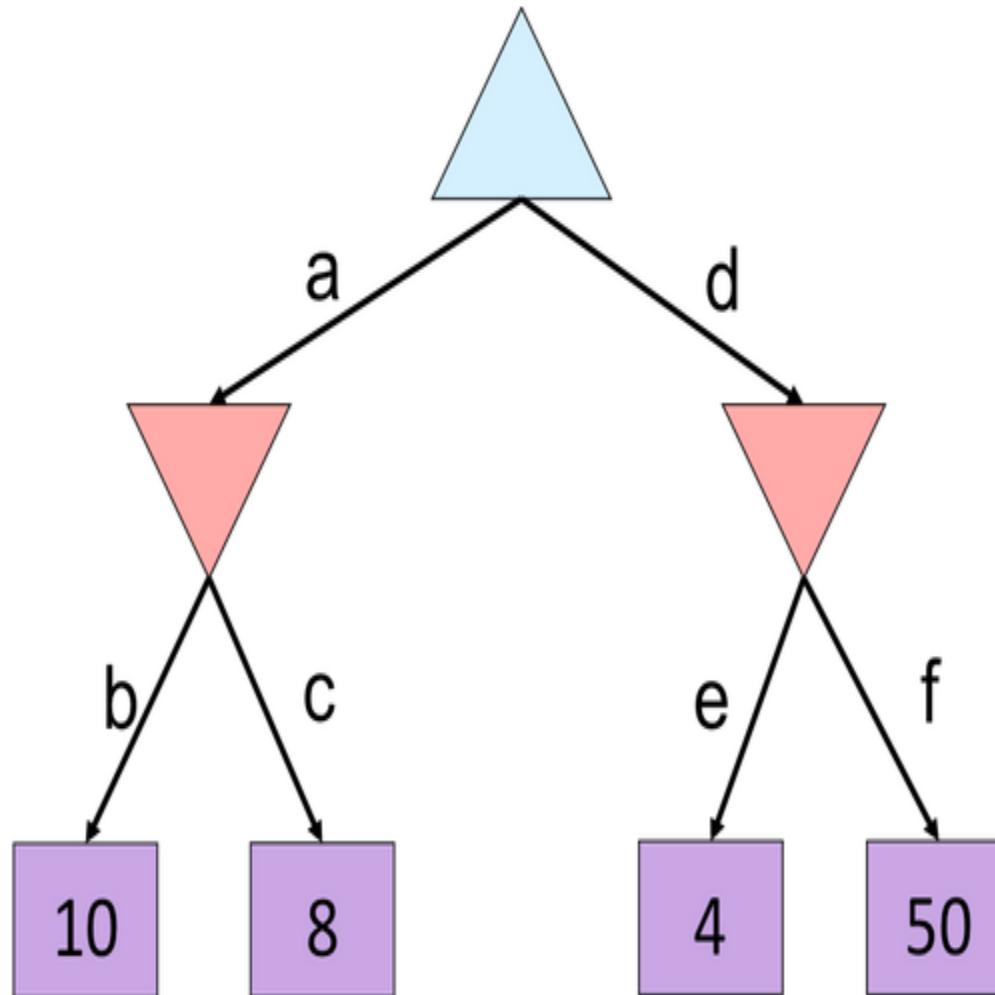
```
def min-value(state , α, β):  
if 终局-检测(state) return Utility(state)  
初始化 v = +∞  
for each a in ACTIONS(state) do  
    v = min(v, max-value(Result(s,a),  
                           α, β))  
    if v ≤ α return v  
    β = min(β, v)  
return v
```

# Alpha-Beta 剪枝属性

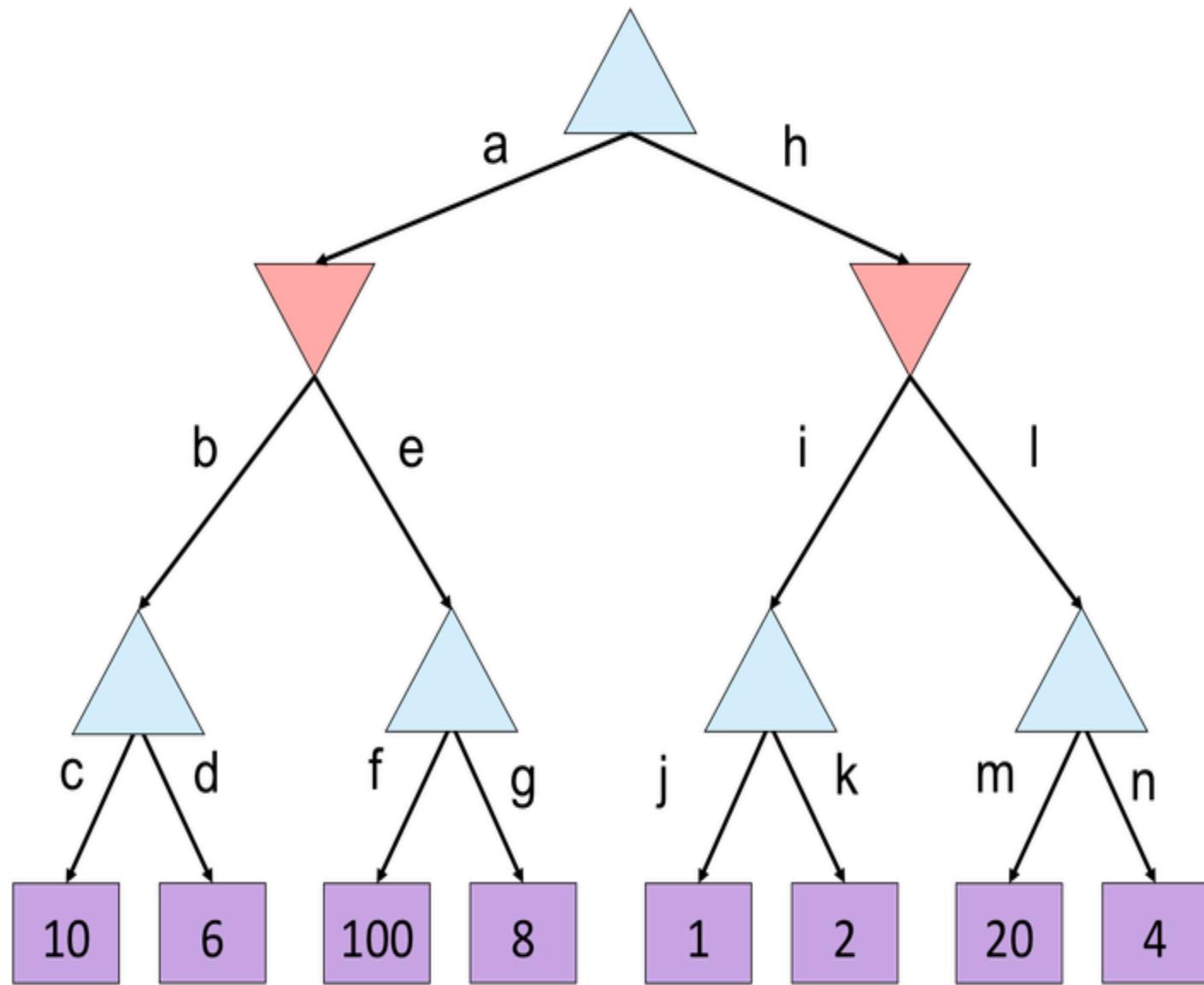
- 剪枝**不影响** 根节点的最小最大值的计算！
- 好的子节点的排序能提高剪枝的有效性
- 假定是“完美的排序”：
  - 时间复杂度能降到  $O(b^{m/2})$
  - 搜索深度能提高一倍！



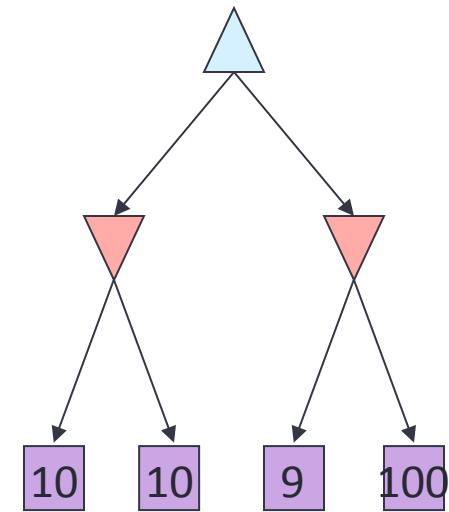
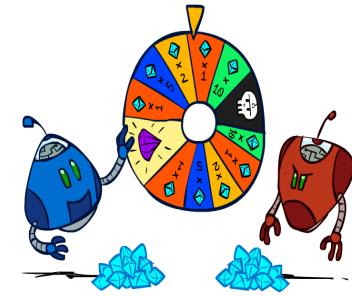
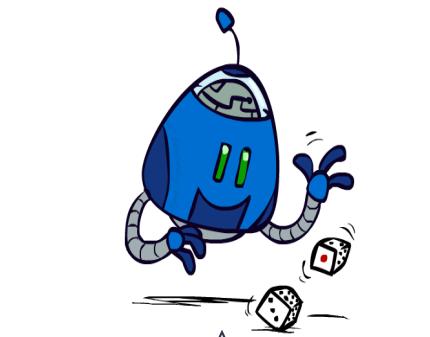
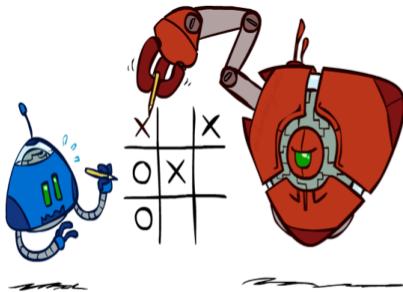
# Alpha-Beta 剪枝测试



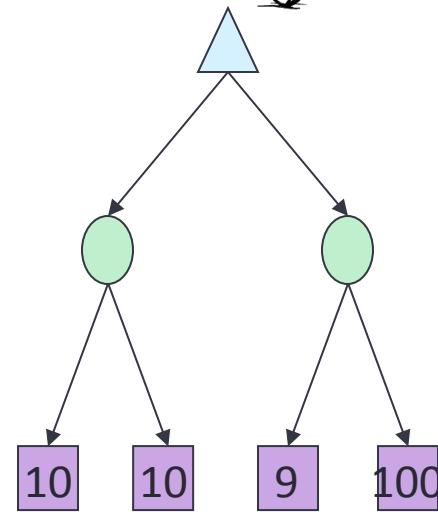
# Alpha-Beta 剪枝测试2



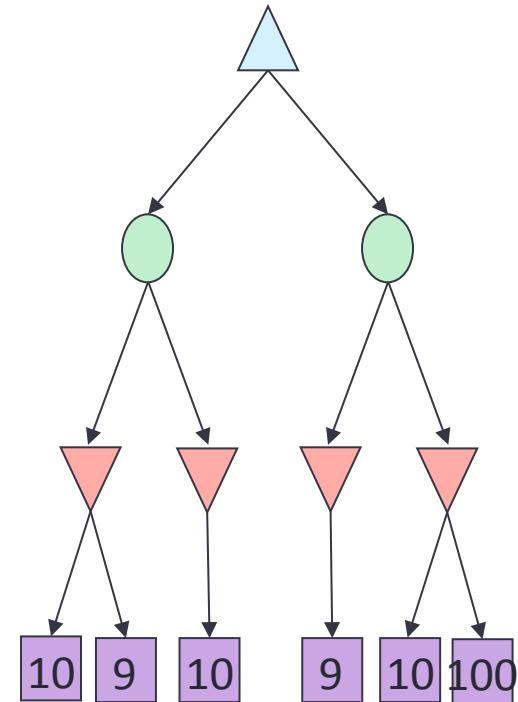
# 搜索树中可能有机遇节点



井字游戏  
最小最大值(*Minimax*)



投资游戏  
期望最大值  
(*Expectimax*)



大富翁游戏  
期望最小最大值(*Expectiminimax*)

# 最小最大值

Function 决策(s) returns 一个行动  
return Actions(s) 中的行动  $a$ , 它的  
value(Result(s,a)) 最大



function value(s) returns a value  
If 终局-检测(s) then return Utility(s)  
if Player(s) = MAX then return  $\max_{a \text{ in Actions}(s)}$  value(Result(s,a))  
if Player(s) = MIN then return  $\min_{a \text{ in Actions}(s)}$  value(Result(s,a))

# 期望最小最大值(Expectiminimax)

Function 决策(s) returns 一个行动

return Actions(s) 里的一个行动  $a$ ，  
它的 value(Result(s,a)) 是最大的



function value(s) returns a value

if 终局-检测(s) then return Utility(s)

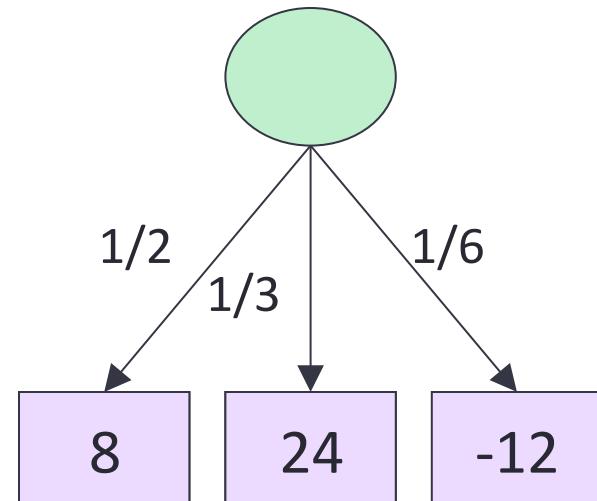
if Player(s) = MAX then return  $\max_{a \text{ in } \text{Actions}(s)}$   
value(Result(s,a))

if Player(s) = MIN then return  $\min_{a \text{ in } \text{Actions}(s)}$   
value(Result(s,a))

if Player(s) = CHANCE then return  $\sum_{a \text{ in } \text{Actions}(s)} \text{Pr}(a) * \text{value}(\text{Result}(s,a))$

# 计算期望最小最大值的代码

$\text{sum}_{a \in \text{Action}(s)}$   
 $\Pr(a) * \text{value}(\text{Result}(s,a))$

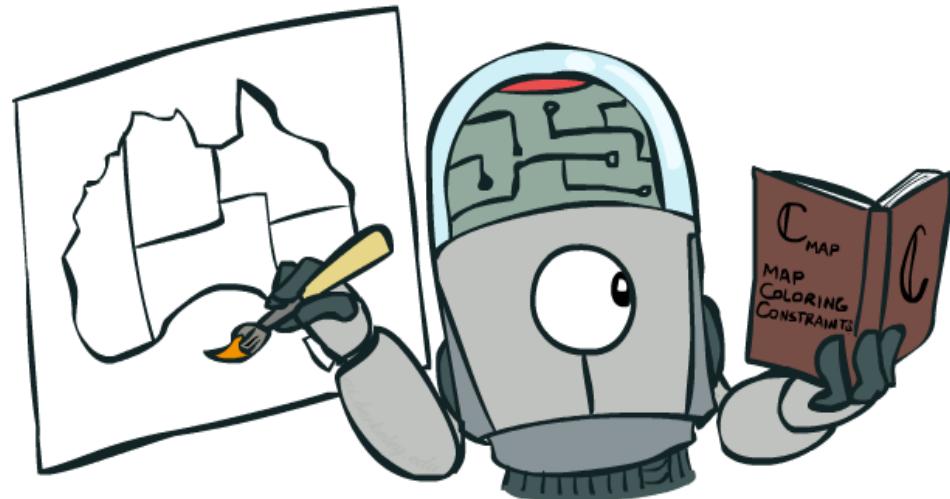


$$v = (1/2)(8) + (1/3)(24) + (1/6)(-12) = 10$$

# 人工智能导论： 约束满足问题

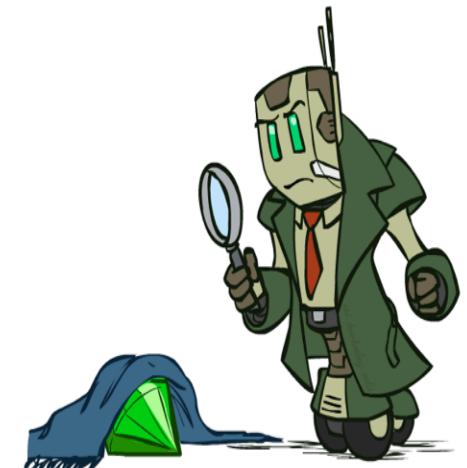
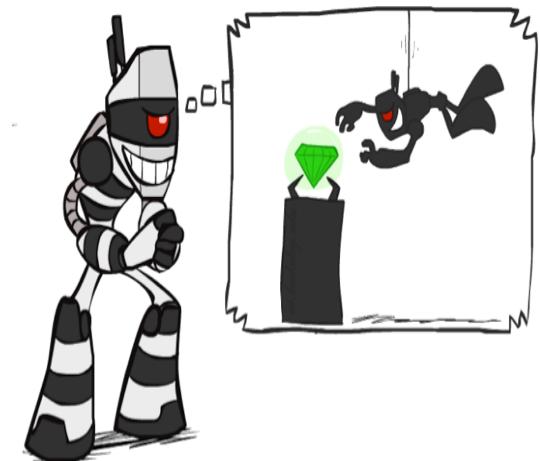
---

齐琦  
海南大学



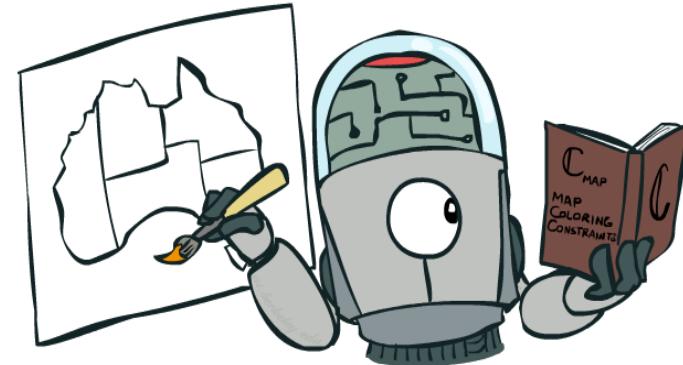
# 两种主要的问题求解

- 规划: 解是一个行动序列 (或行动策略)
  - 重要的是到目标状态的路径
  - 路径有不同的成本和探索深度
  - 和行动的顺序相关
- 鉴定: 对变量的配置
  - **路径不重要，目标状态本身重要**
  - 约束满足问题是其中一类基本的问题



# 约束满足问题 (Constraint Satisfaction Problems; CSPs)

- 状态由若干个 变量  $X_i$  组成，每个变量有一个 取值域  $D_i$
- 目标测试是一套 约束规则，规定了允许的变量间取值组合
- 通用算法求解，比普通搜索问题的算法更 高效



# 举例: 地图着色

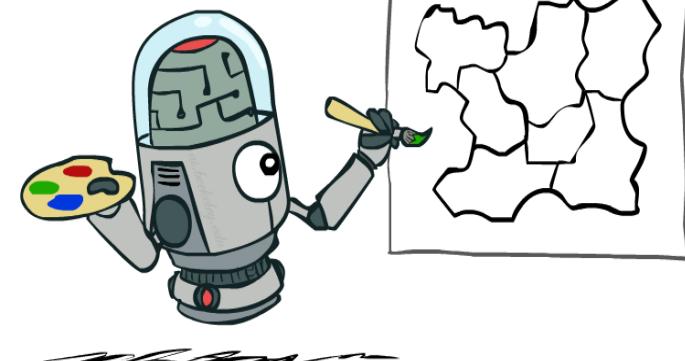
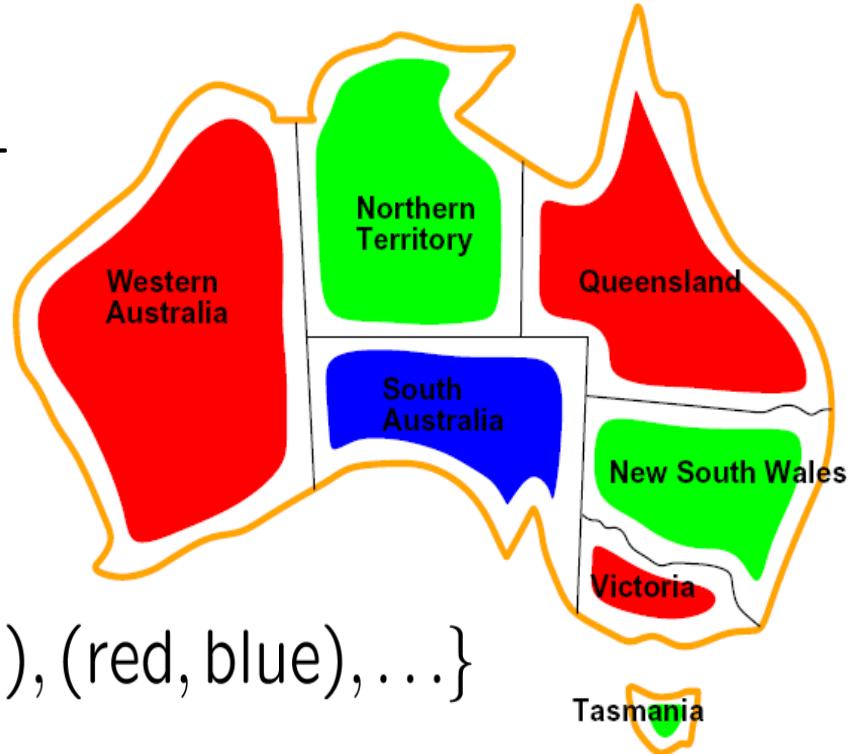
- 变量: WA, NT, Q, NSW, V, SA, T
- 值域:  $D = \{\text{red}, \text{green}, \text{blue}\}$
- 约束: 临近区域必须有不同的颜色

隐式:  $\text{WA} \neq \text{NT}$

显示:  $(\text{WA}, \text{NT}) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), \dots\}$

- 解是一组对所有变量的配值，并满足所有的约束，例如：

$\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{green}\}$



# 举例: N-皇后

- 问题描述方法 1:

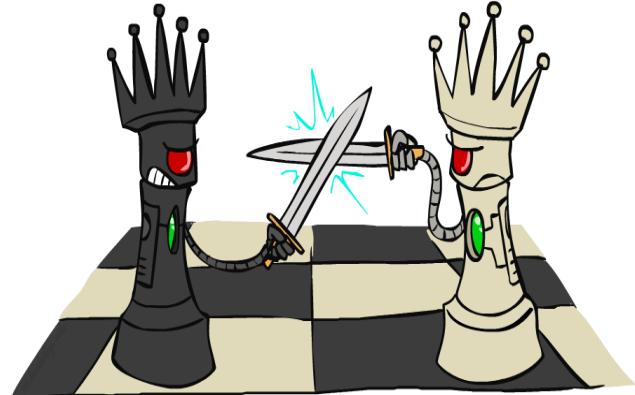
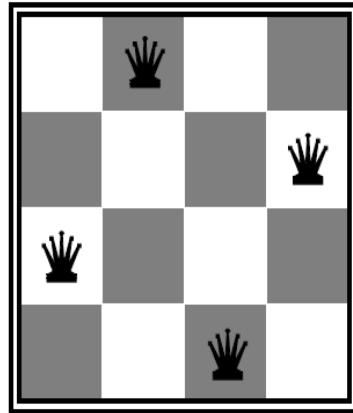
- 变量:

$$X_{ij}$$

- 值域:

$$\{0, 1\}$$

- 约束条件:



$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

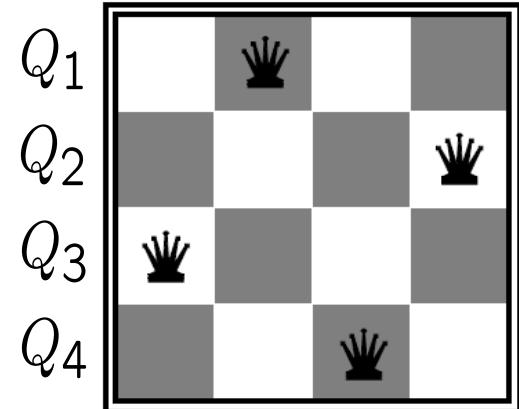
$$\forall i, j, k \quad (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\sum_{i,j} X_{ij} = N$$

# 举例: N-皇后

- 描述方法 2:

- 变量:  $Q_k$
- 值域:  $\{1, 2, 3, \dots, N\}$
- 约束条件:

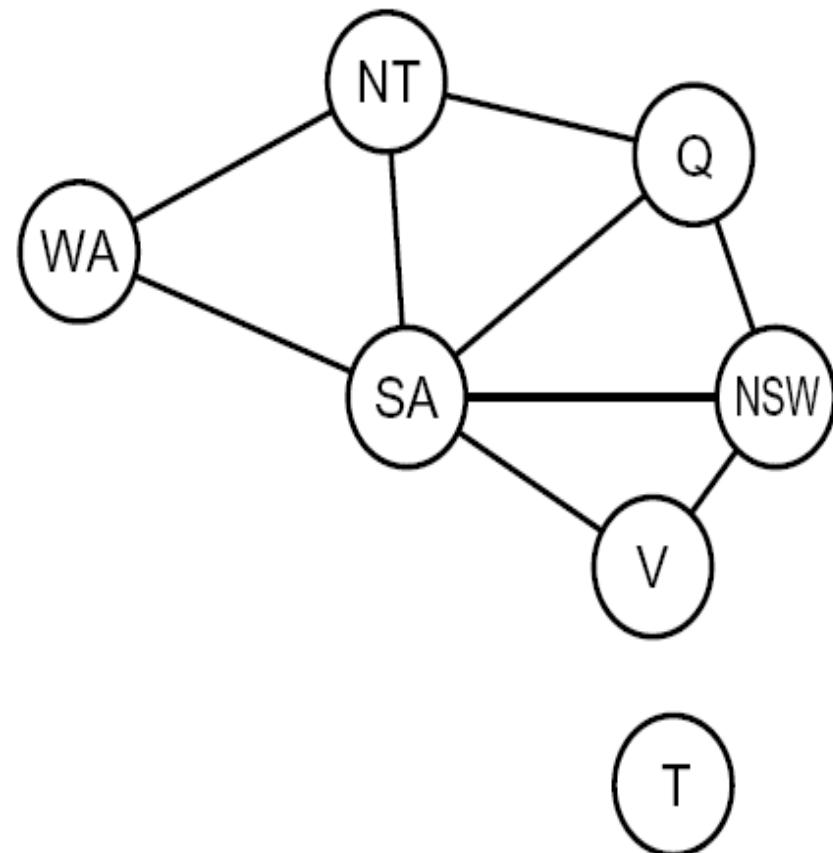
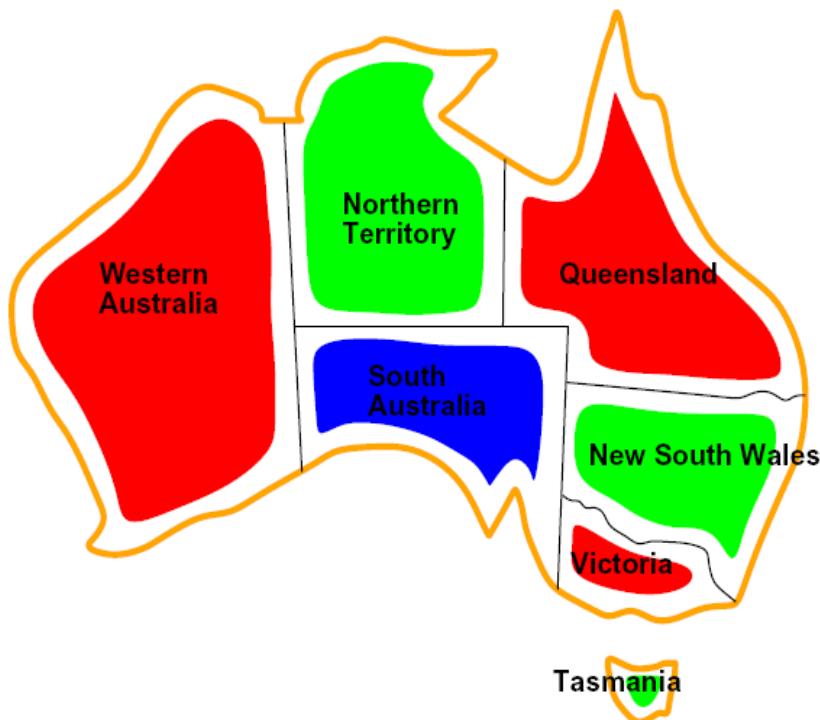


隐式表述:  $\forall i, j \mid$  彼此不威胁到对方  $(Q_i, Q_j)$

显式表述:  $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\}$

...

# 约束图

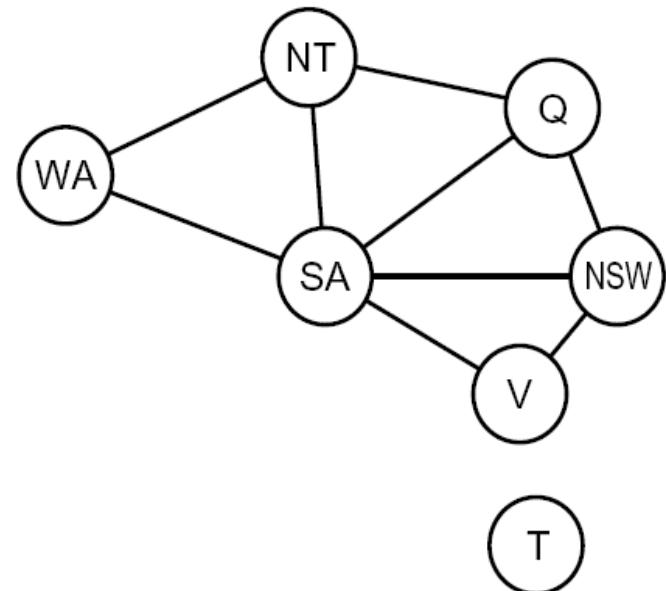


节点：变量

连接：存在约束关系

# 约束图

- 二元约束满足问题: 每个约束关联至多两个变量
- 二元约束图: 节点代表变量, 边代表约束
- 约束满足问题求解算法利用图的结构加速搜索 (利用约束关系削减搜索空间, 后面会看到)。



# 举例：密码算术

- 变量：

$F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

- 值域：

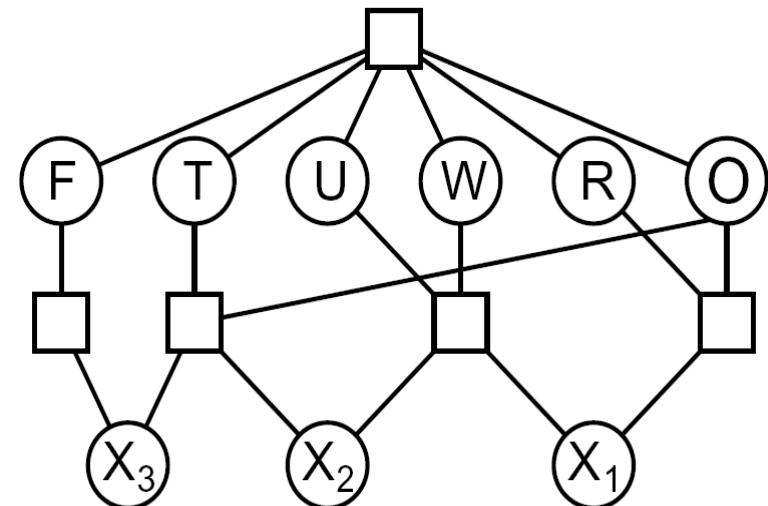
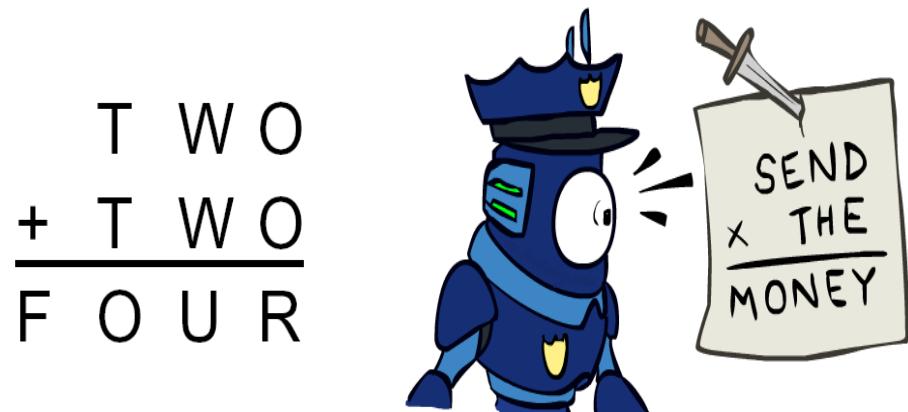
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- 约束条件：

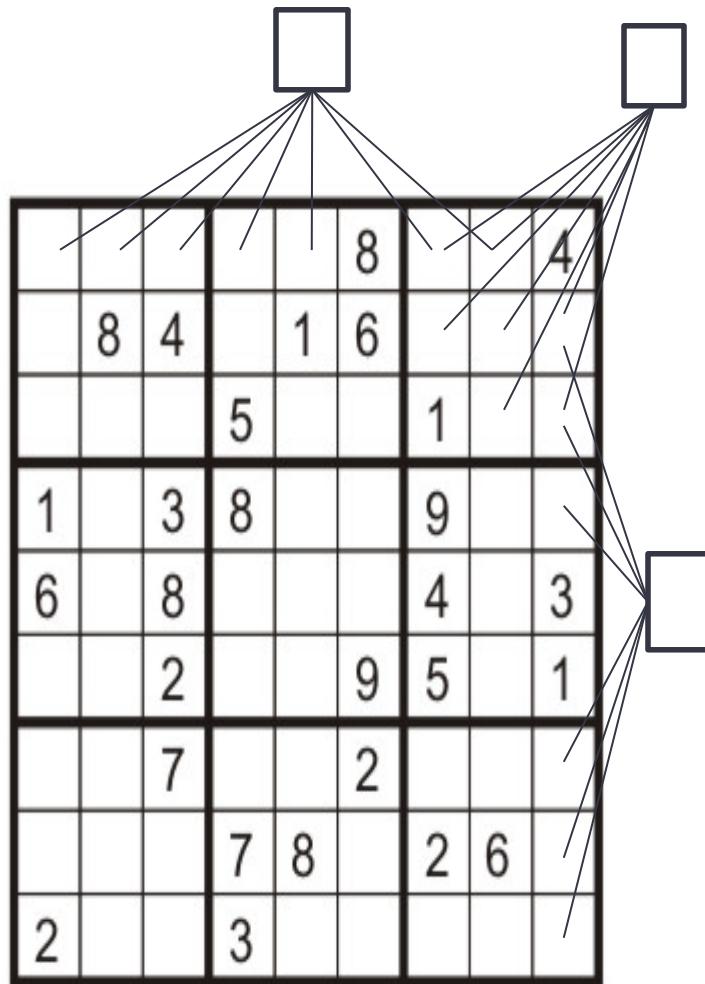
`alldiff( $F, T, U, W, R, O$ )`

$$O + O = R + 10 \cdot X_1$$

...

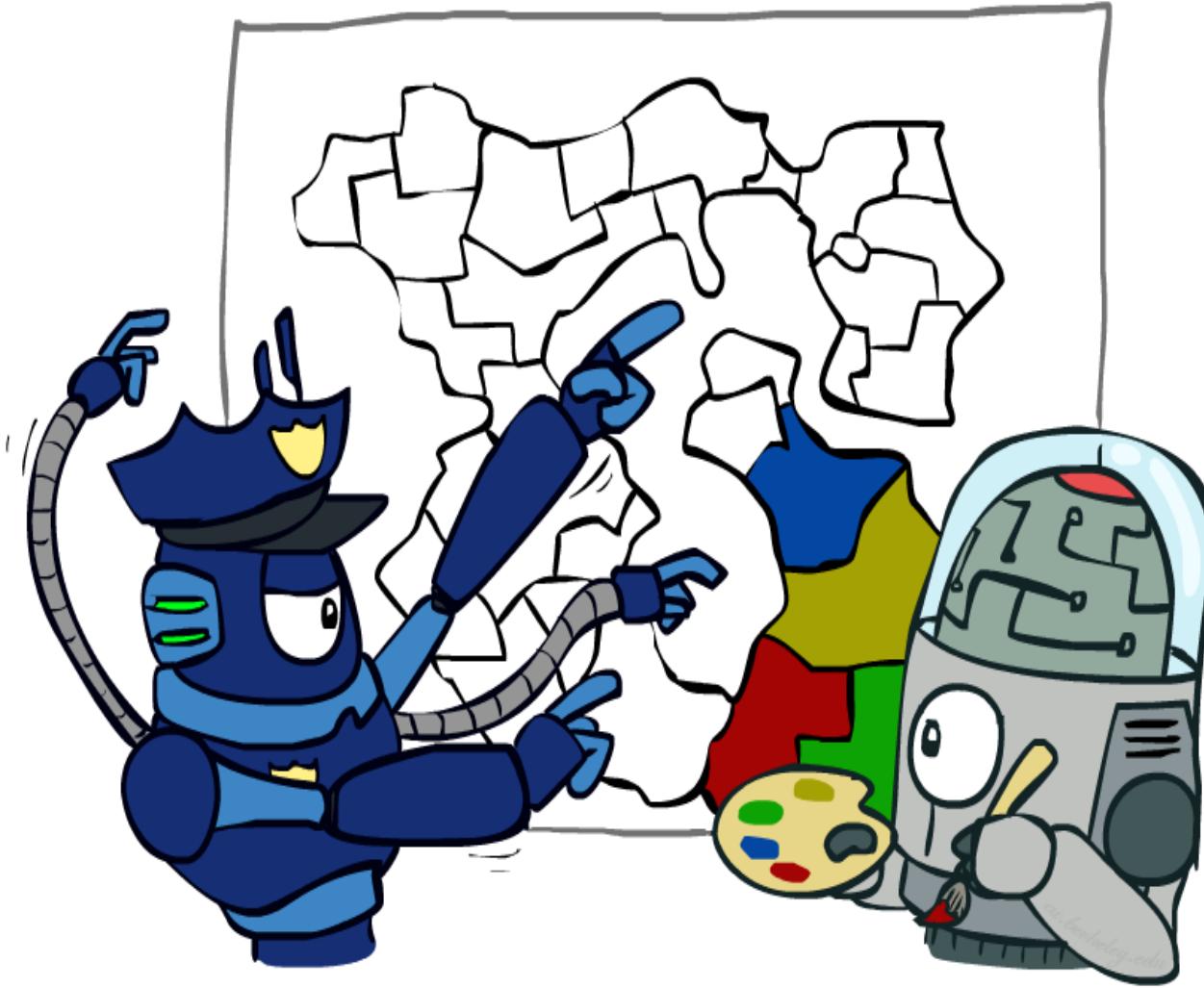


# 举例: 数独(Sudoku)



- 变量:
  - 每一个 空白方格
- 值域:
  - $\{1, 2, \dots, 9\}$
- 约束条件:
  - 每列9个数字都不同
  - 每行9个数字都不同
  - 每个9x9大方格里的9个数字都不同

# 多样的约束满足问题和约束条件



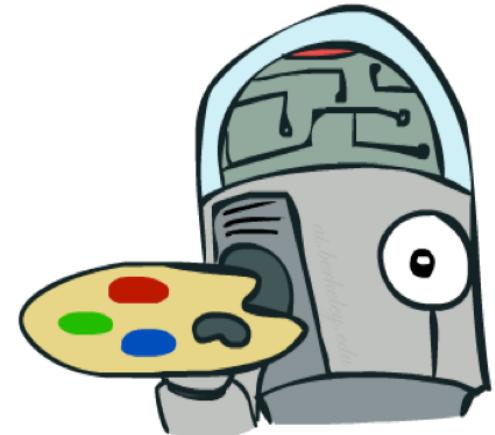
# 约束满足问题变量的多样性

- 离散变量

- 有限值域,  $n$  变量, 值域大小是  $d$ 
  - $O(d^n)$  完整的配值复杂度
  - 比如, 布尔可满足性问题 (SAT) ,  $d=2$ , 约束是逻辑子句 (SAT 是 NP-complete 问题)
- 无限值域 (整数, 字符串, 等.)
  - 比如, 作业调度, 变量是每个作业任务的开始时间
  - 线性约束可解的, 非线性约束不可判定

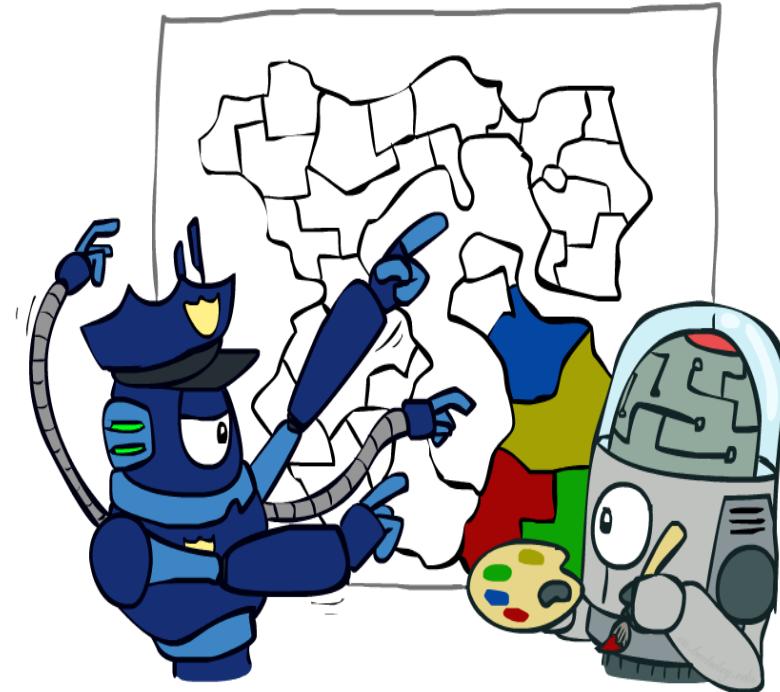
- 连续变量

- 比如, 哈勃望远镜观测的开始和结束时间的分配
- 线性约束问题, 可用线性规划方法有效解决



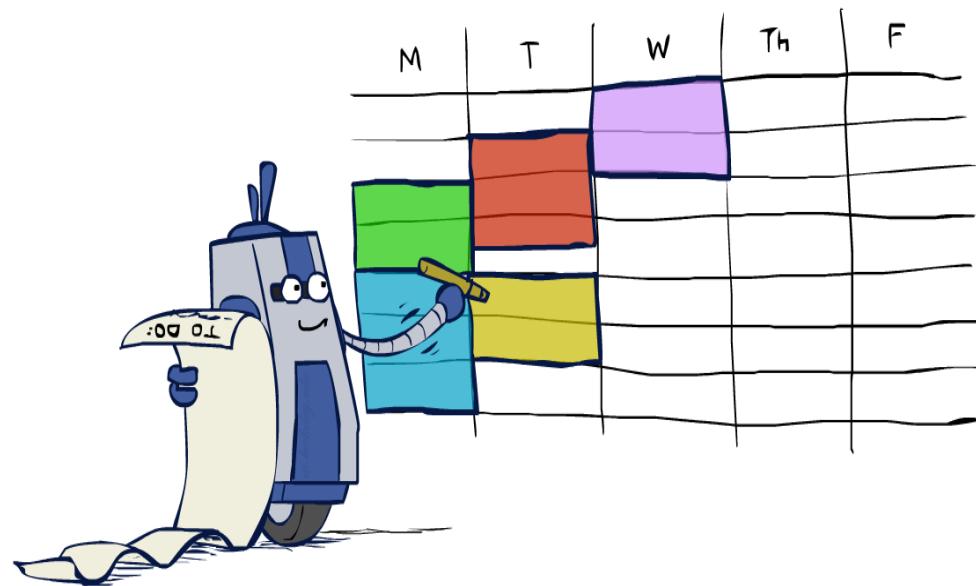
# 约束的多样性

- 多样的约束
  - 一元约束, 涉及一个变量(相当于缩减值域), 比如:  
 $SA \neq \text{green}$
  - 二元约束, 涉及成对的变量, 例如:  
 $SA \neq WA$
  - 高阶约束, 涉及三个以上的变量:  
比如, 密码算术问题中的列约束
- 偏好 (软约束):
  - 比如, 红色比绿色好
  - 可用成本函数对配值组合评估
  - 这种情况也叫, **有约束的优化** 问题



# 真实世界中的约束满足问题

- 配置问题: 比如, 哪个老师教哪一门课
- 时间表计划问题: 比如, 哪一门课被安排在哪个时间和地点?
- 硬件配置
- 公交调度
- 工厂调度
- 电路规划
- 故障诊断
- ... 还有许多!



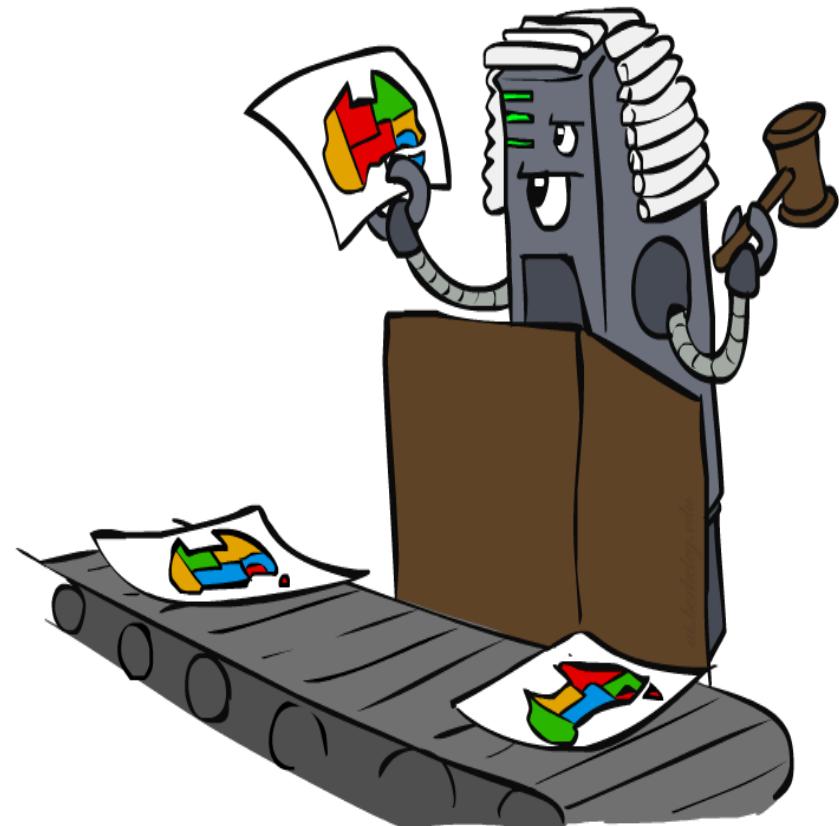
- 许多真实世界里的问题都涉及实数值变量...

# 求解约束满足问题



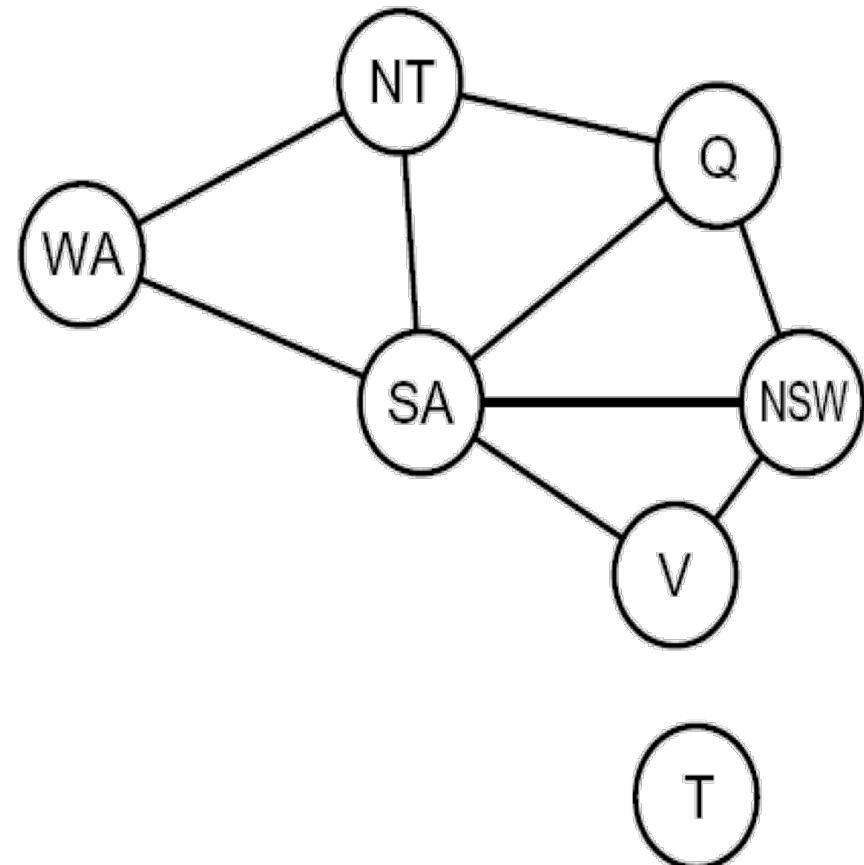
# 标准搜索的描述

- 状态反映了变量配值的当前情况  
(部分配值)
  - 初始状态: 没有配值, {}
  - 行动集合( $s$ ): 分配一个值给一个未赋值的变量
  - 结果状态( $s,a$ ): 该变量被赋了这个值
  - 目标-检测( $s$ ): 是否所有变量已被赋值 并且满足所有约束条件
- 我们开始将用最直接的方法, 然后逐步改进



# 一般搜索方法

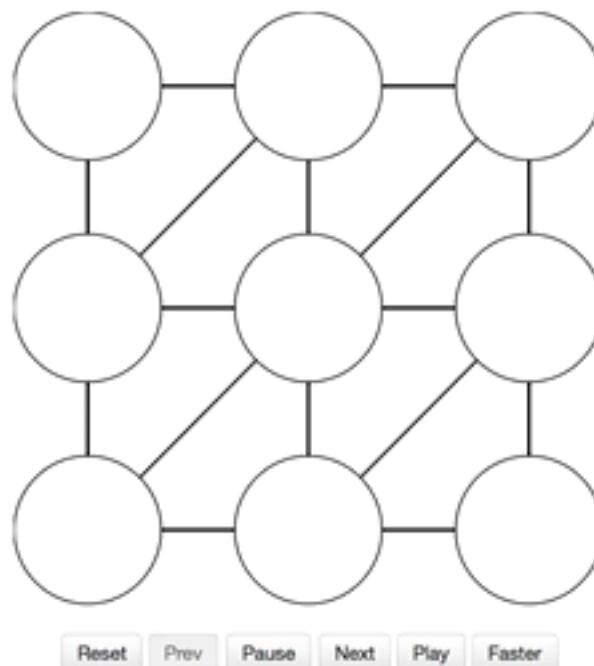
- 广度优先(BFS)会怎么样?



- 深度优先(DFS)会怎么样?

- 这些最直接的搜索方法在解这个问题时有什么问题?

# Video of Demo Coloring -- DFS



Graph

Simple

Algorithm

Naive Search

Ordering

- None
- MRV
- MRV with LCV

Filtering

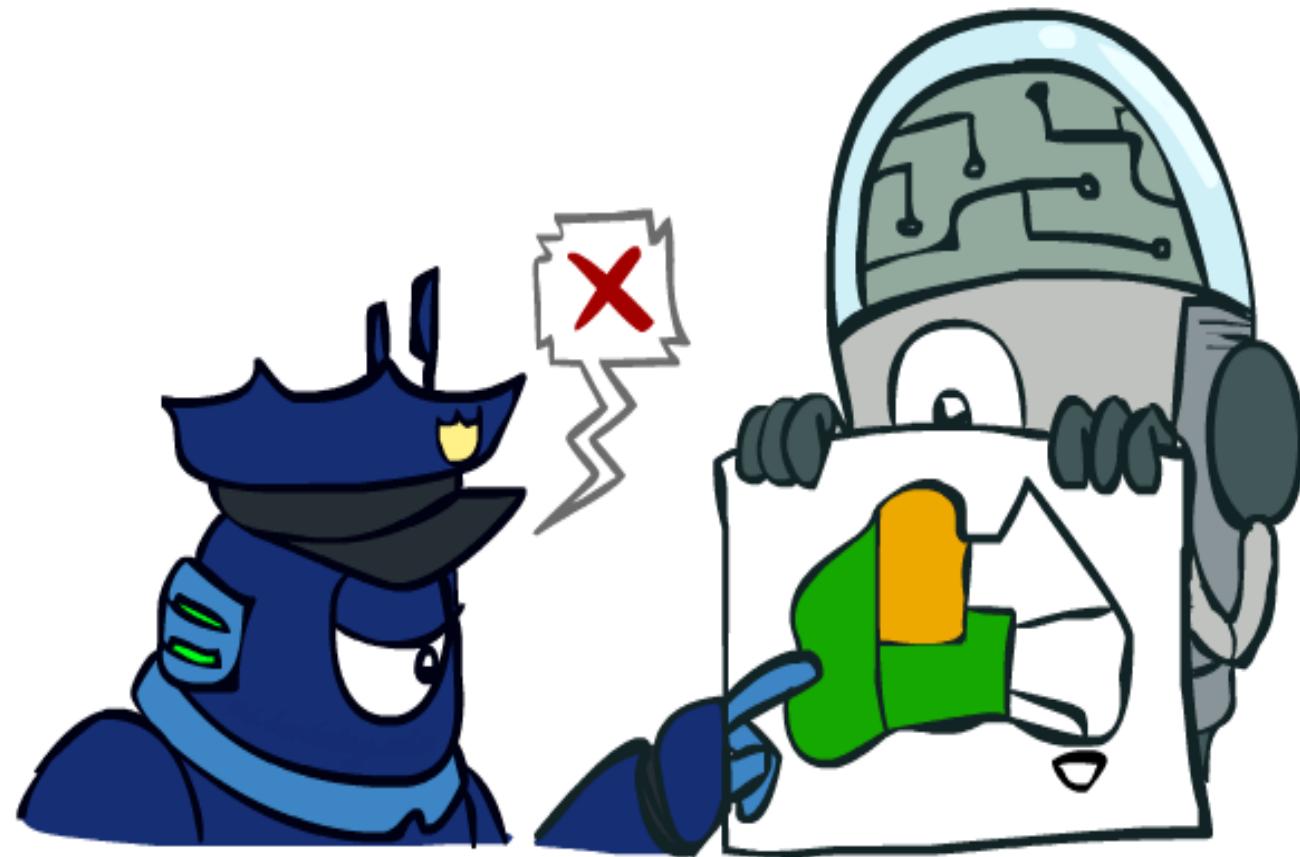
- None
- Forward Checking
- Arc Consistency

Speed

Speedup  
1 x

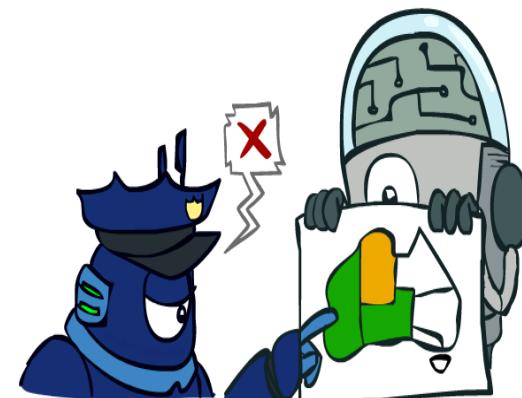
Frame Delay  
700

# 回溯搜索

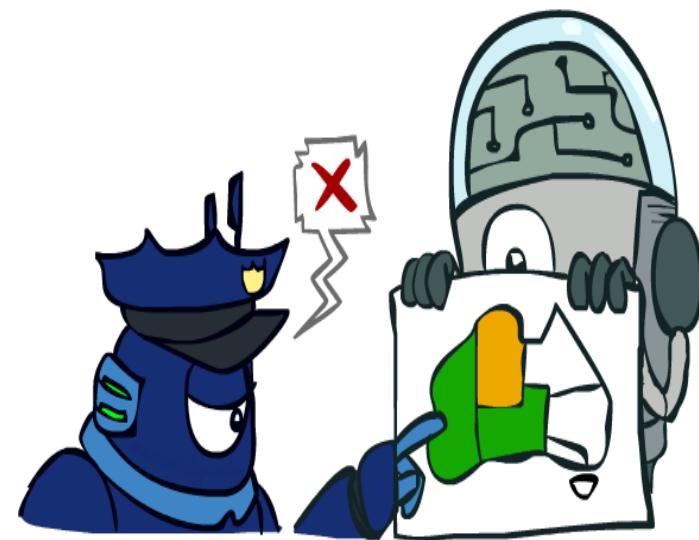
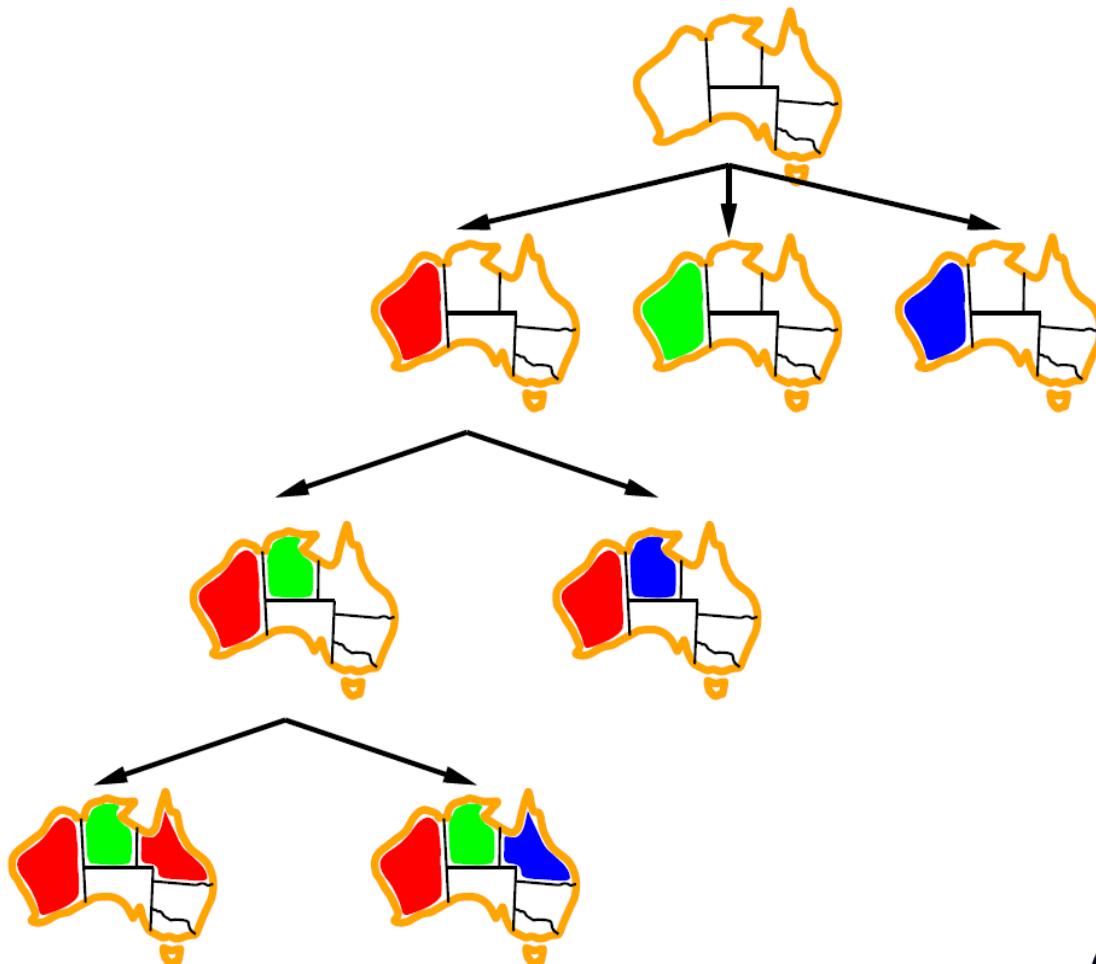


# 回溯搜索

- 回溯搜索是基本的无启发式信息的算法，用来求解CSP问题
- 想法 1：一次探索一个变量
  - 在每一步只需考虑给一个变量配值
- 想法 2：一边探索一边检查约束条件
  - 探索过程中检查当前的变量配值是否满足约束条件
  - 也许需要花费一些计算来检查约束条件是否满足
  - “逐步增加的目标测试”
- 深度优先搜索结合这两点改进，就叫作 **回溯搜索**
- 能够解决  $n$ -皇后问题，直至  $n = 25$



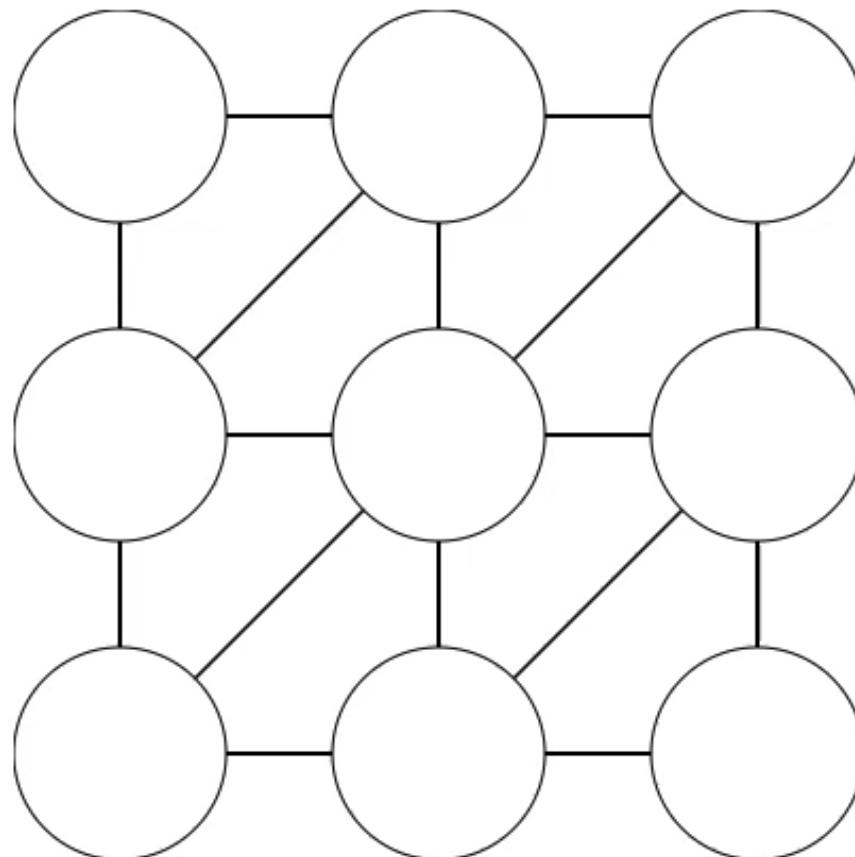
# 回溯搜索举例



# 回溯搜索

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
function BACKTRACK(assignment,csp) returns a solution, or failure
    if assignment 是完全的 then return assignment
    var ← 选择-未赋值的-变量(csp,assignment)
    for each value in 排序-值域中的值(var,assignment,csp) do
        if value 是一致的 with assignment then
            添加 {var = value} to assignment
            推断结果 ← 推断(csp,var,assignment)
            if 推断结果 ≠ failure then
                添加 推断结果 to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then return result
            移除 {var = value} and 推断结果 from assignment
    return failure
```

# 回溯搜索演示—着色问题



Reset Prev Pause Next Play Faster

## Graph

Simple

## Algorithm

Naive Search

## Ordering

- None
- MRV
- MRV with LCV

## Filtering

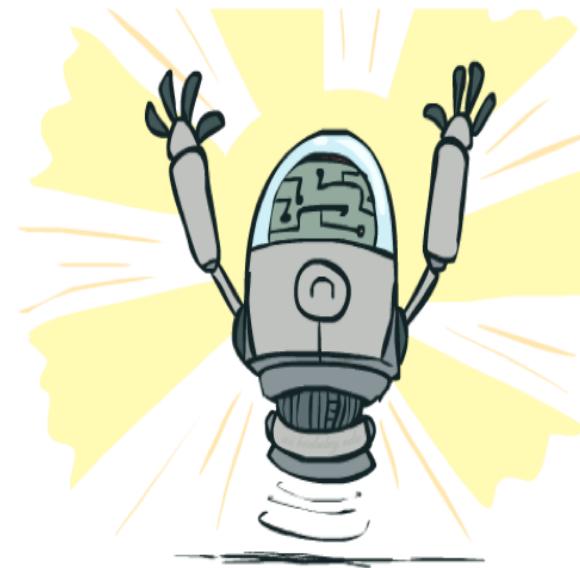
- None
- Forward Checking
- Arc Consistency

## Speed

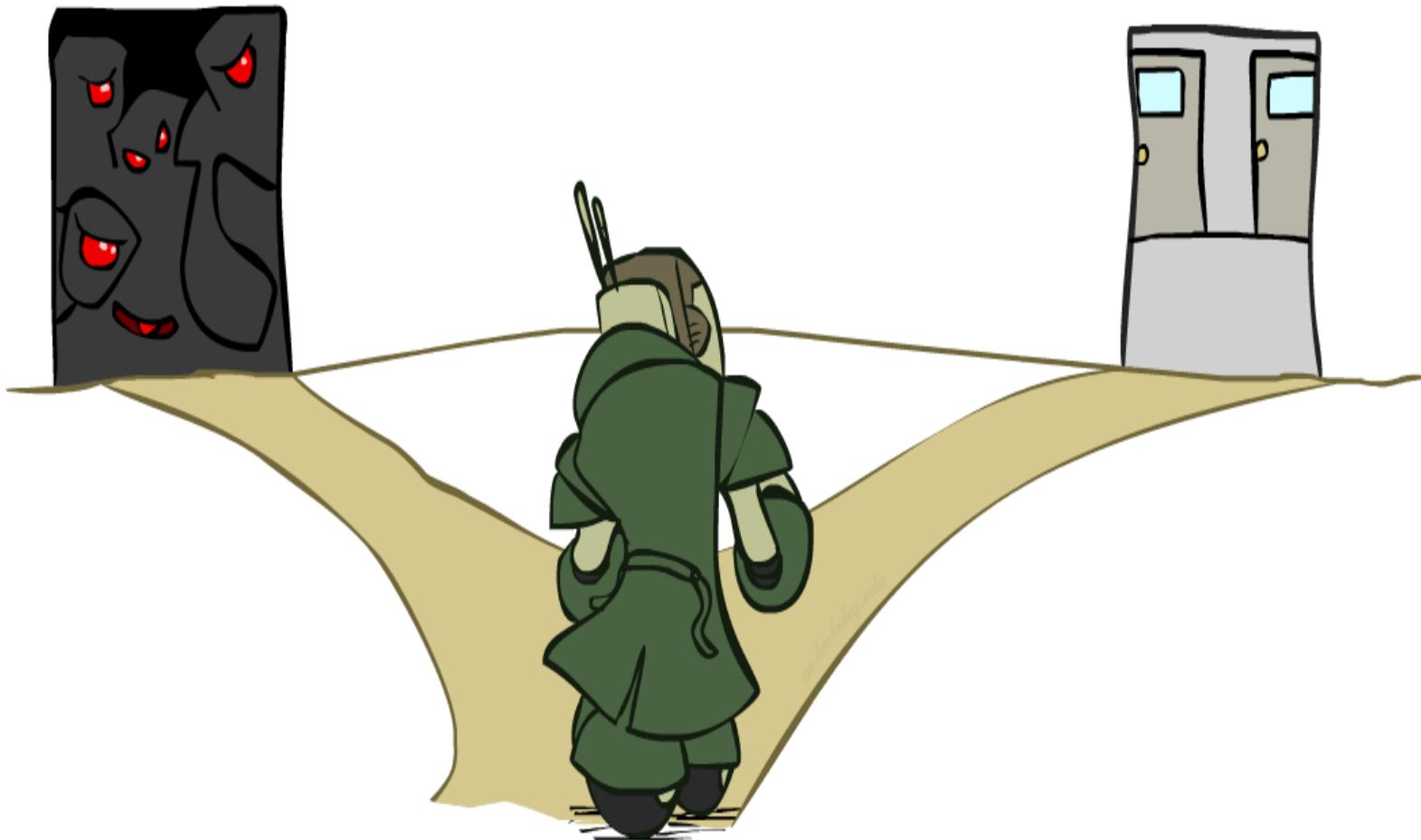
Speedup Frame Delay  
1 x 700

# 回溯搜索的改进

- 改进思想能极大提升搜索速度，并且适用于多方面的问题
- **排序**:
  - 下一轮挑选哪个变量进行配值？
  - 挑选值时，有什么顺序上的考虑？
- **过滤**: 我们能否提前预测不可避免的失败？
- **结构**: 我们能否利用问题的结构？

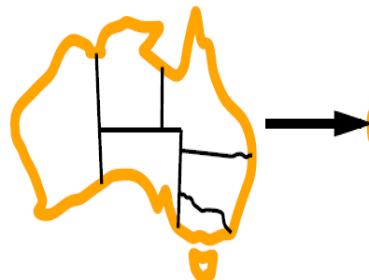


# 排序



# 变量排序

- `var`  $\leftarrow$  选择-未赋值的-变量(csp,assignment)
- 变量排序: **最小剩余值 (MRV)**:
  - 先选择其值域中所剩合理可选的值最少的变量

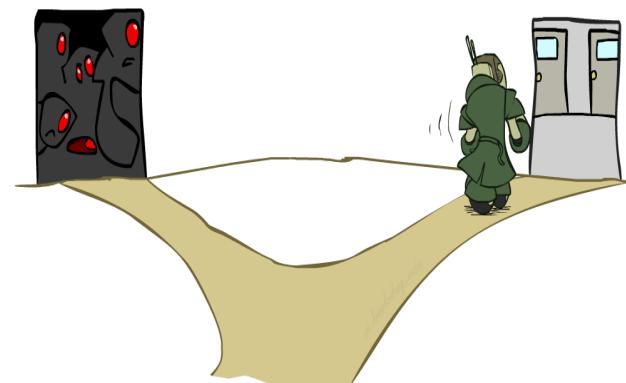
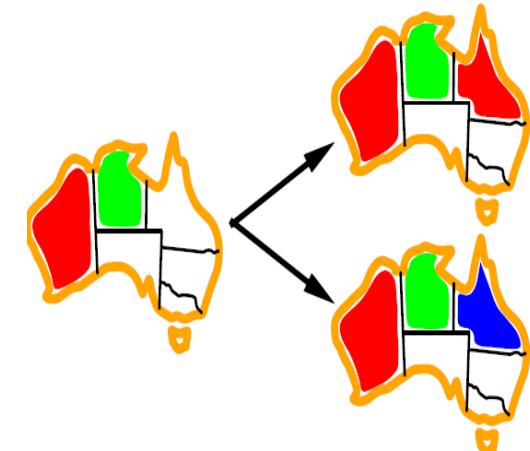


- 为什么是最少而不是最大?
- “快速失败” 排序
- 使用**连接度数启发信息** 打破一样的情况
  - 选择和其他变量连接数最多的变量



# 对值的排序选择

- **for each value in 排序-值域里的值  
(var,assignment,csp) do**
- **选择最小制约的值 (LCV)**
  - 选择对剩下的变量在选值的时候影响最小的值
  - 这可能需要花费些计算时间!
- 为什么最小而不是最大制约的?
- 结合这些排序上的改进，能够解决  
1000-皇后问题



# 过滤

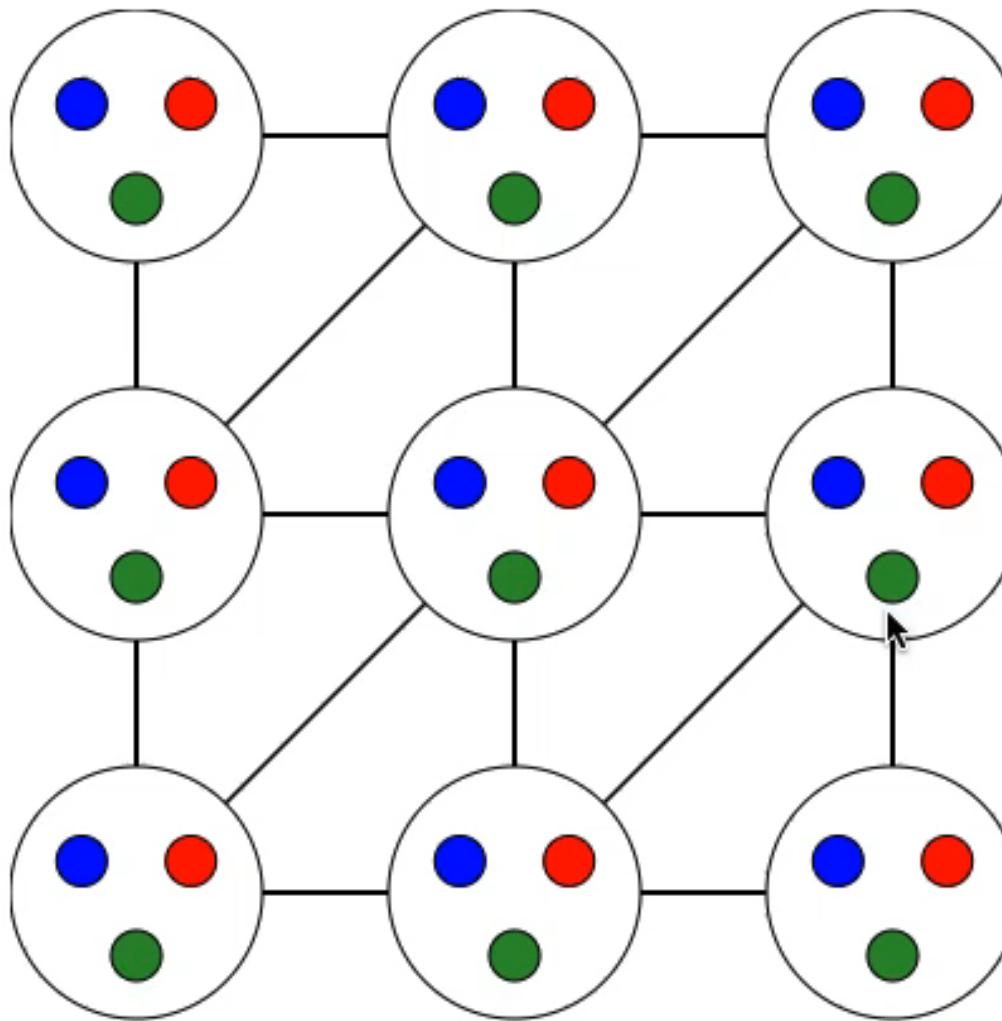


# 过滤: 前向检查法

- 过滤: 搜索中持续观测未赋值的变量的值域, 去掉违反约束条件的选项
- 简单的过滤: **向前检查法**
  - 当添加对一个变量的赋值后, 划掉剩下变量值域中违反约束条件的值



# 演示一 回溯搜索结合前向检查法



Graph

Simple

Algorithm

Backtracking

Ordering

- None
- MRV
- MRV with LCV

Filtering

- None
- Forward Checking
- Arc Consistency

Speed

Speedup

1 x

Frame Delay

700

Reset

Prev

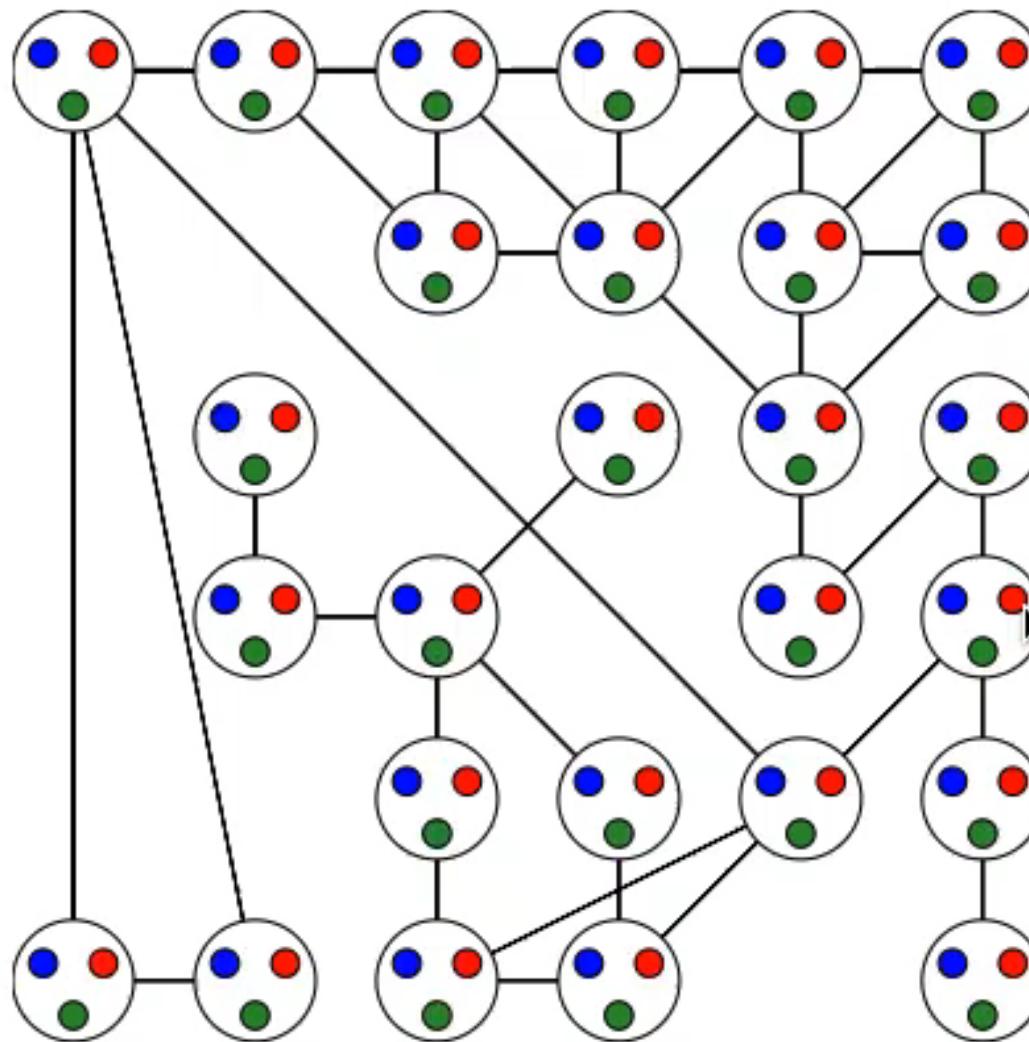
Pause

Next

Play

Faster

# 复杂的图



Reset Prev Pause Next Play Faster

## Graph

Complex

## Algorithm

Backtracking

## Ordering

- None
- MRV
- MRV with LCV

## Filtering

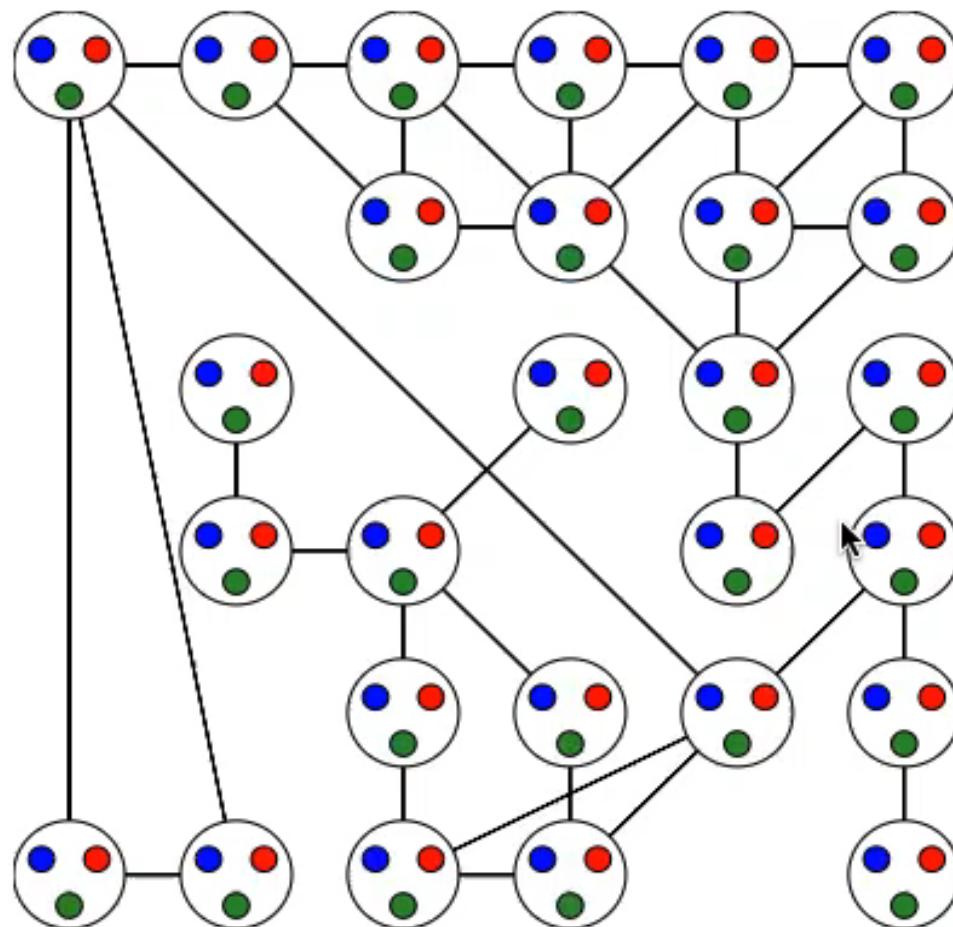
- None
- Forward Checking
- Arc Consistency

## Speed

Speedup  
1 x

Frame Delay  
700

# 演示：着色 – 回溯 + 前向检查 + 排序



Reset Prev Pause Next Play Faster

## Graph

Complex

## Algorithm

Backtracking

## Ordering

- None
- MRV
- MRV with LCV

## Filtering

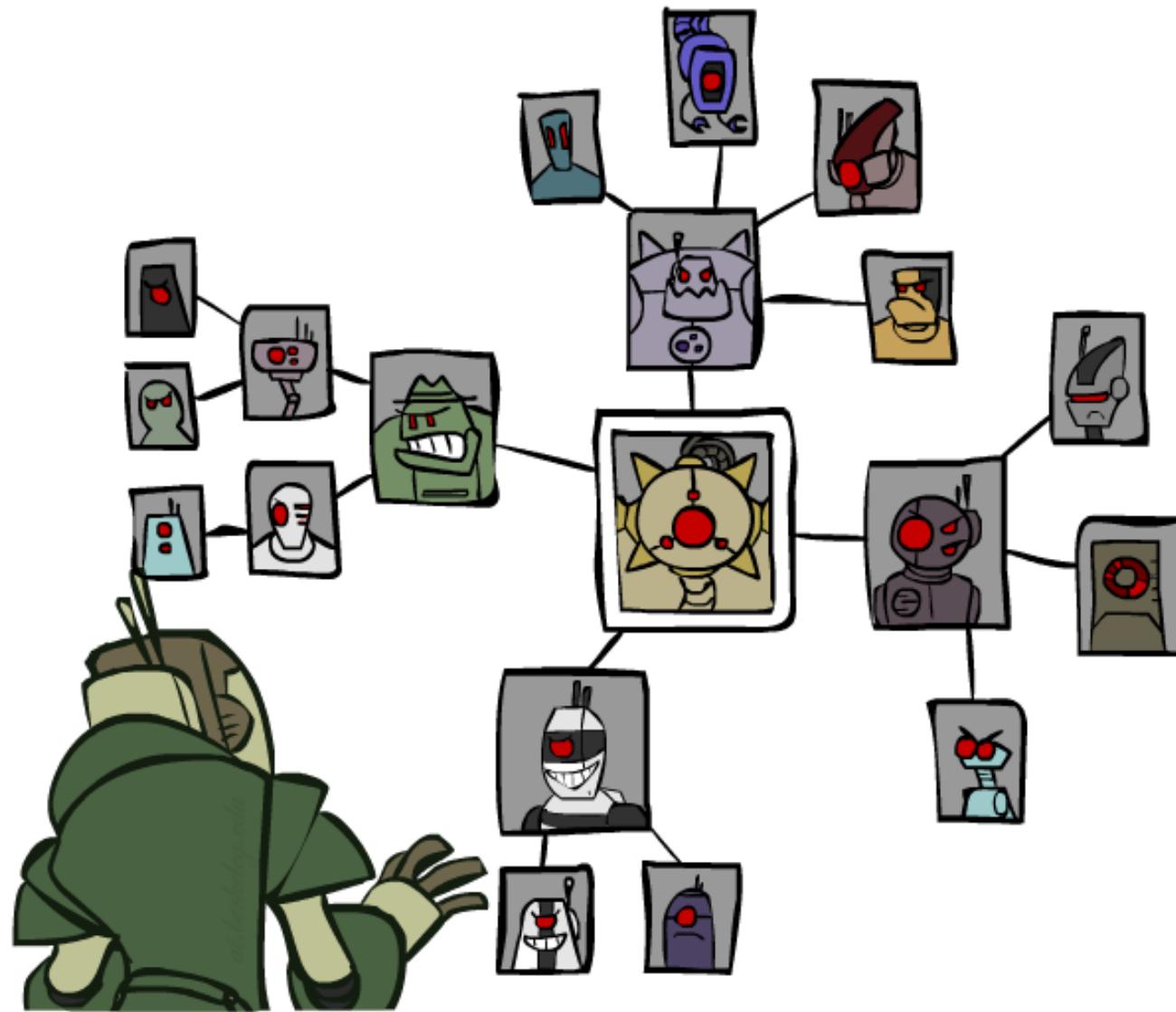
- None
- Forward Checking
- Arc Consistency

## Speed

Speedup  
1 x

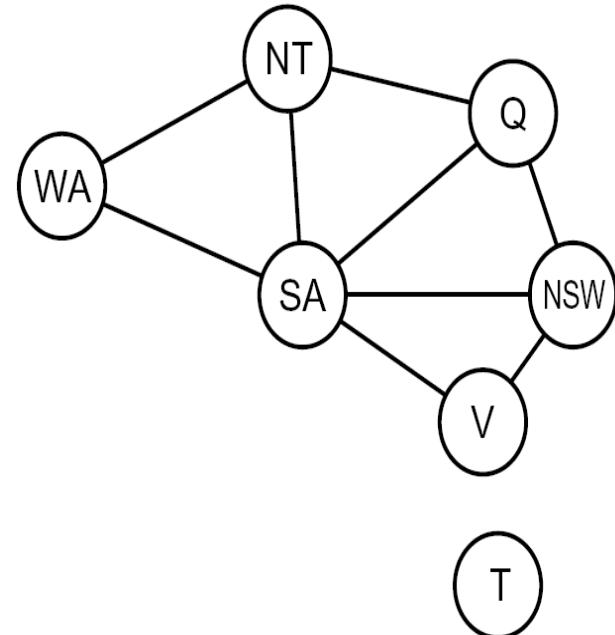
Frame Delay  
700

# 利用问题的结构

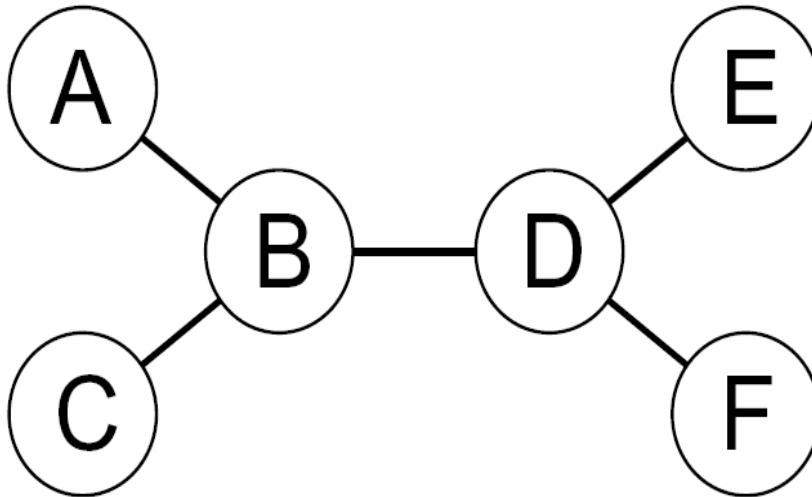


# 问题结构

- 极端情况: 独立的子问题
  - 例如: Tasmania 和大陆不相连
- 独立的子问题可以被认为是约束图中连接的组件:
  - 分治法!
- 假设一个  $n$  变量的图可以被分成 几个子问题, 每个子问题有  $c$  个变量:
  - 最差情况下的求解成本是  $O((n/c)(d^c))$ ,  $n$  的线性关系
  - 比如,  $n = 80$ ,  $d = 2$ ,  $c = 20$ , 搜索 1千万 节点/秒
  - 原始问题:  $2^{80} = 40\text{亿年}$
  - 4个子问题:  $4 \times 2^{20} = 0.4$  秒



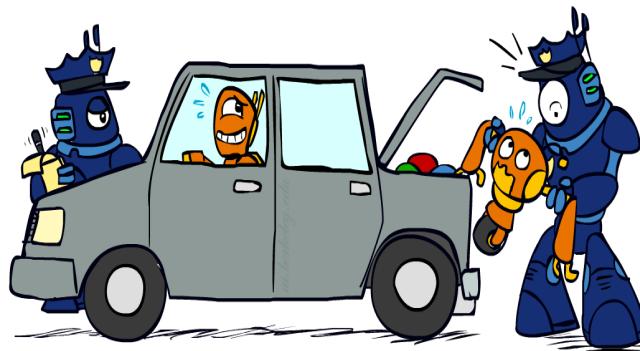
# 树结构的 CSPs



- 定理: 如果约束图无环, 则其对应的约束满足问题的求解时间复杂度是  $O(n d^2)$ 
  - 比较一般的 CSPs, 最差时间复杂度是  $O(d^n)$

# 有用的概念: 弧的一致性 (连贯性; Consistency)

- 一个弧  $X \rightarrow Y$  是 **一致的** 当且仅当 对于X中的 **每一个**  $x$  ,  $Y$  中存在 **某个**  $y$  值 不违背任何一个约束条件

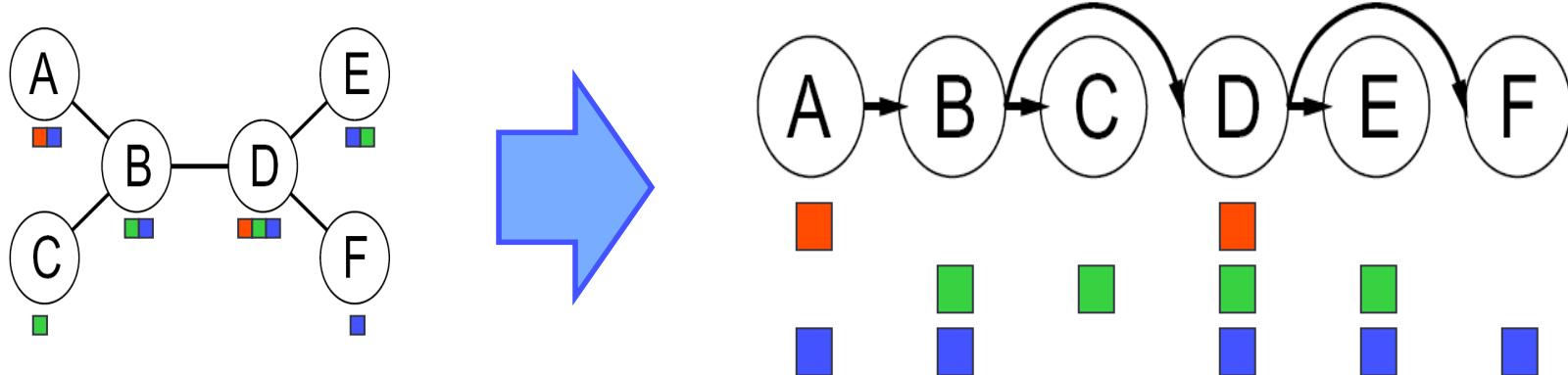


从弧的尾部删除

- 前向检查: 强制检查剩余未赋值变量指向新赋值变量的弧的一致性

# 树结构的 CSPs

- 树结构的CSPs的求解算法：
  - 排序：选一个根节点，把变量线性排序，使得父节点排在子节点之前

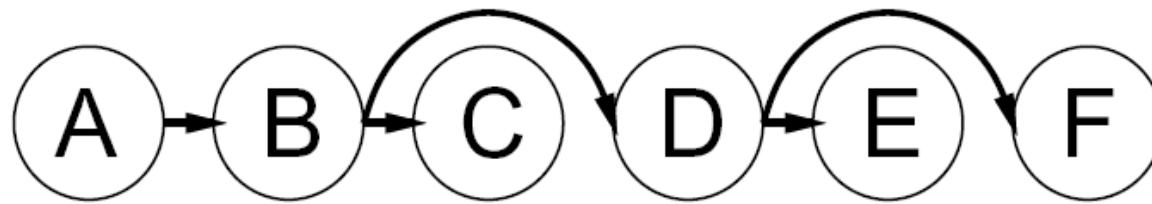


- 从后向前删除: For  $i = n : 2$ , 应用 删除不一致的值(父节点( $X_i$ ), $X_i$ )
- 从前向后赋值: For  $i = 1 : n$ , 赋值  $X_i$  和父节点 Parent( $X_i$ ) 相一致的值
- 运行时间:  $O(n d^2)$  (为什么? )



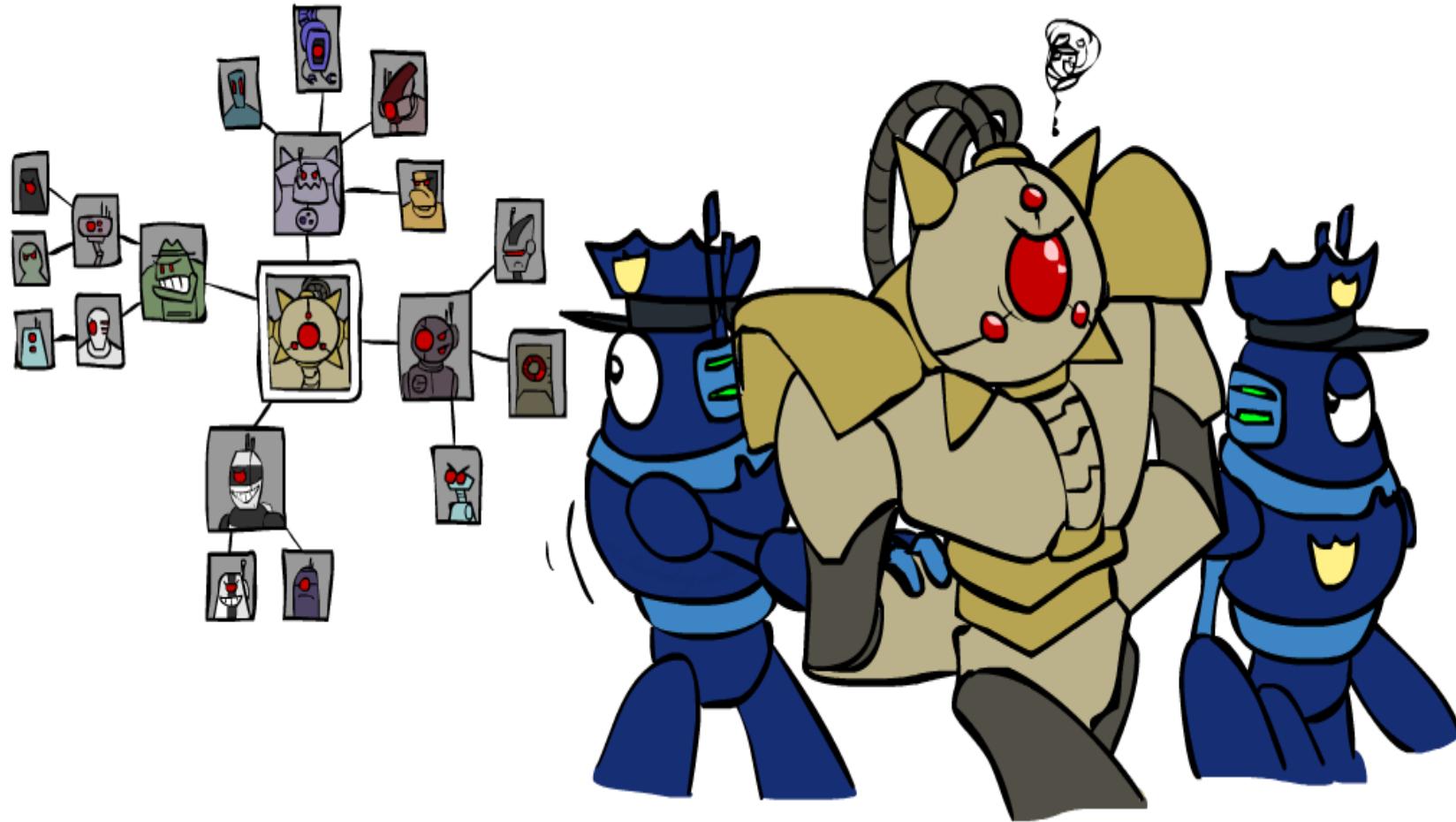
# 树结构的 CSPs

- 声明 1: 在从后向前的一致性检查后, 所有根到叶的弧都是一致性的
- 证明: 每个  $X \rightarrow Y$  如果是一致性的, 那么  $Y$  的值域以后不会被减小 (因为  $Y$  的子节点在  $Y$  之前先被处理过)

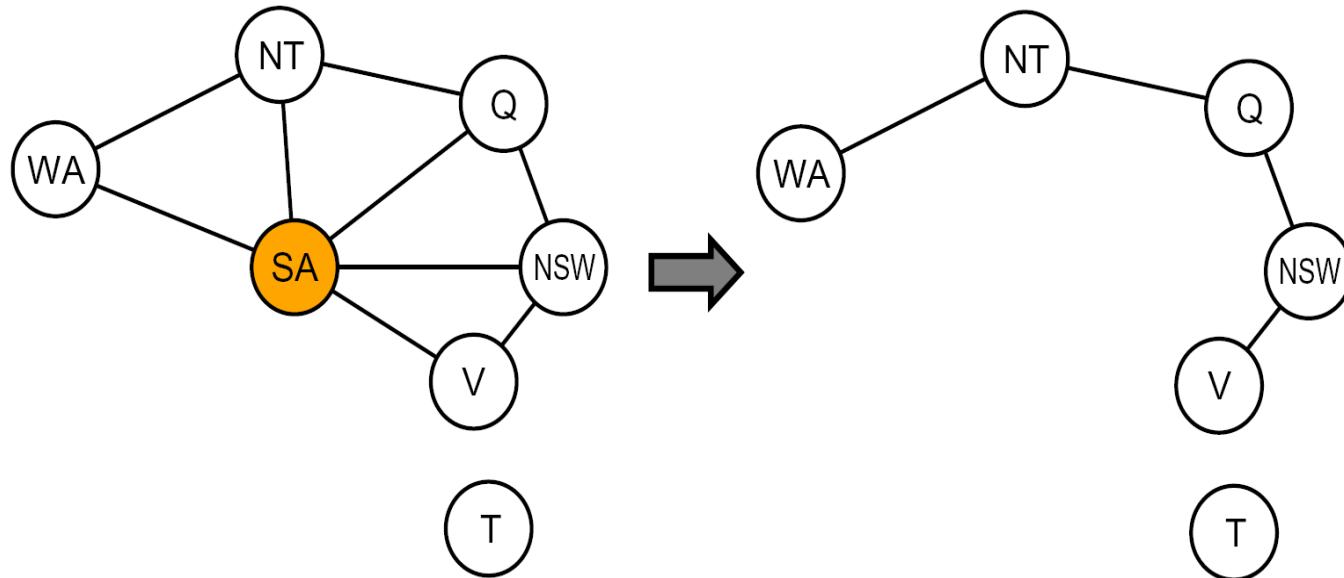


- 声明 2: 如果从根到叶的弧都是一致性的, 那么从前向后的赋值过程将不会有回溯
- 证明: 归纳法
- 为什么这个算法不适用于约束图中有环的情况?

# 改进结构



# 几乎是树结构的 CSPs



- **条件制约**: 赋值一个变量，剪裁这个变量的邻居变量的值域
- **切集条件制约**: 对切集变量赋值 (所有可能的组合)，使得 剩下的约  
束图变成一个树
- 切集大小是 $c$ , 时间复杂度为  $O( (d^c) (n-c) d^2 )$ ,  $c$  很小时算法很快
- 例如, 80 个变量,  $c=10$ , 40亿 years -> 0.029 秒

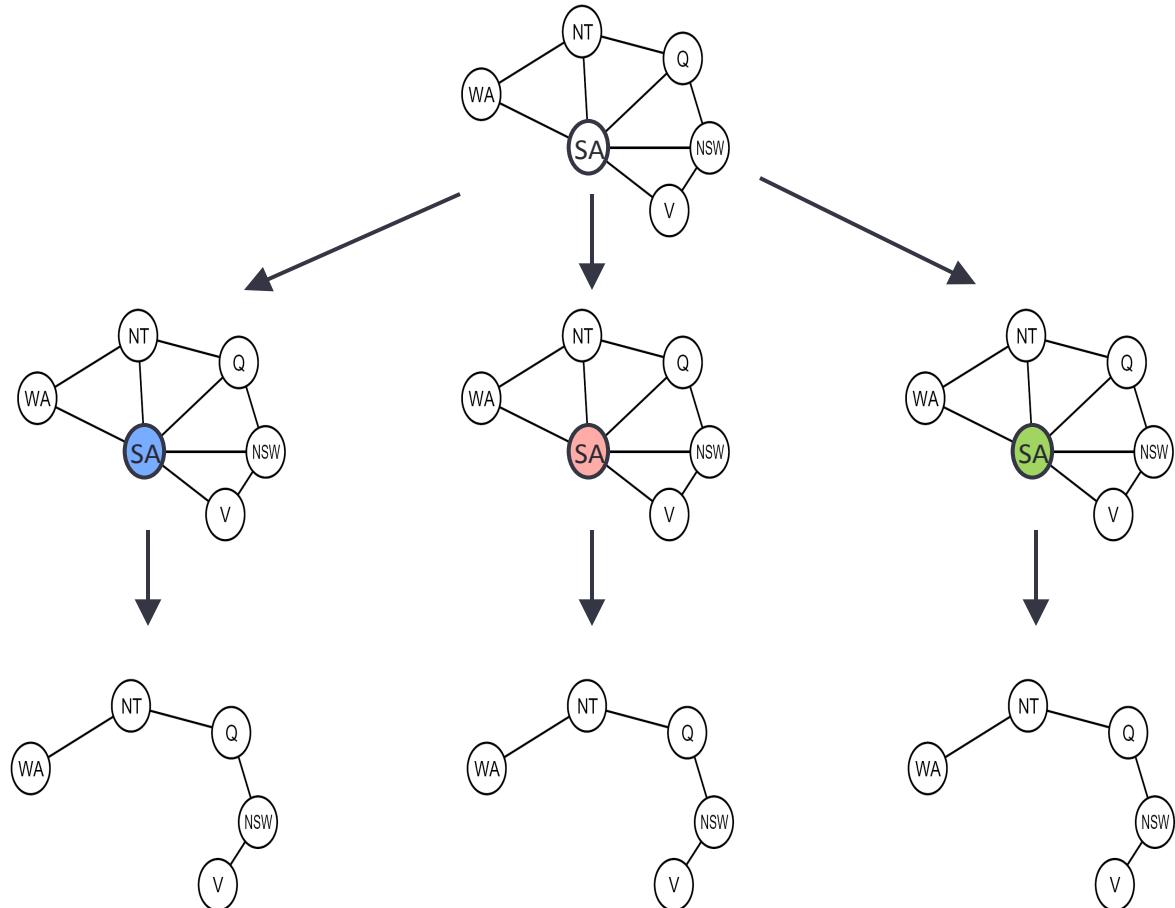
# 切集条件制约

选择一个切集

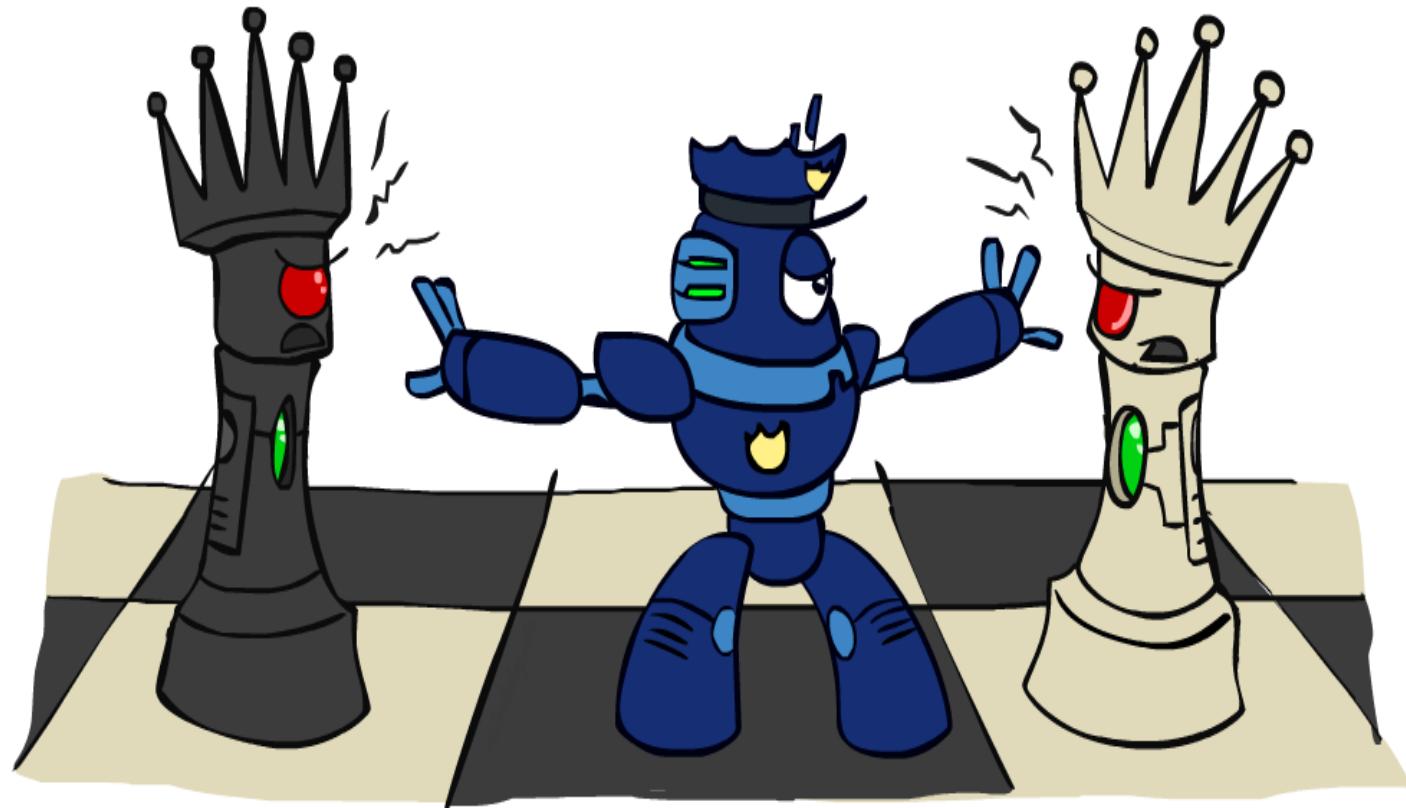
对切集变量赋值  
(所有可能组合)

计算剩余的CSP相  
对于每一组切集  
赋值

求解剩余的 CSPs  
(树结构的)

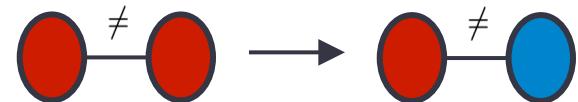


# 局部方法求解 CSPs



# 最小冲突算法

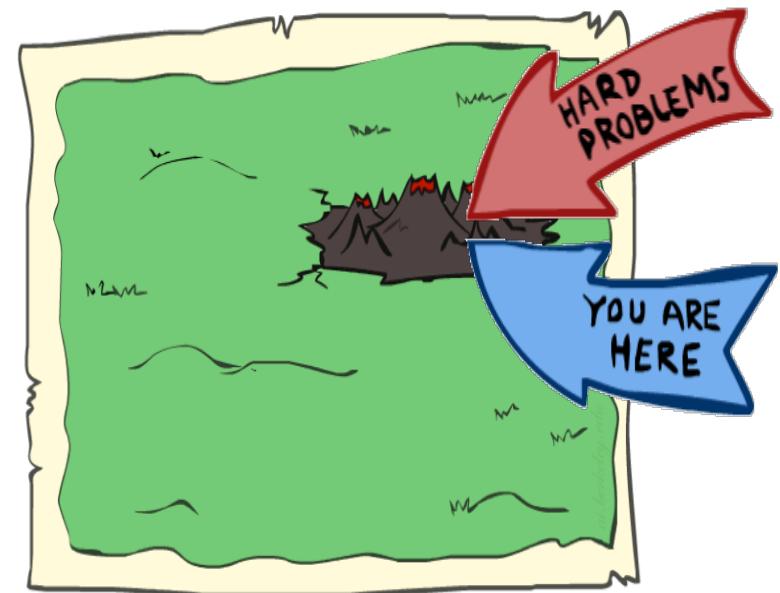
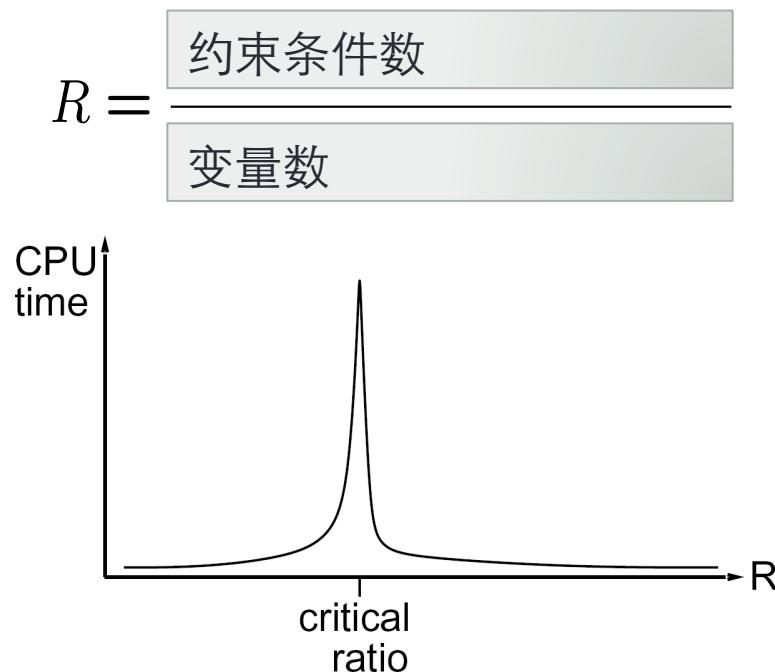
- 像爬山算法, 但不完全是!



- 算法: 开始时所有状态都已随机赋值, 迭代改进, 直至问题得到求解,
  - 变量选择: **随机选择** 任何冲突的变量
  - 值选择: **最小冲突启发信息:**
    - 选择一个和约束条件冲突最少的值
    - 相当于爬山算法中  $h(v)$ =冲突的约束条件总数
    - 随机打破平局情况

# 最小冲突算法的表现性能

- 给定随机初始化状态下, 几乎可以在常量时间里以高成功概率求解  $n$  为任意数的  $n$ -皇后问题, (例如,  $n = 10,000,000$ )!
- 同样的表现性能对任意随机产生的 CSP都适用, 除了在一个很窄的比例范围内



# 总结：约束满足问题

- 一类特殊的搜索问题：
  - 状态是变量的(部分)配值
  - 目标检测通过检查约束条件
- 基本算法：回溯搜索
- 提速思路：
  - 排序
  - 过滤
  - 结构
- 实践中，最小冲突法的局部算法经常很有效

