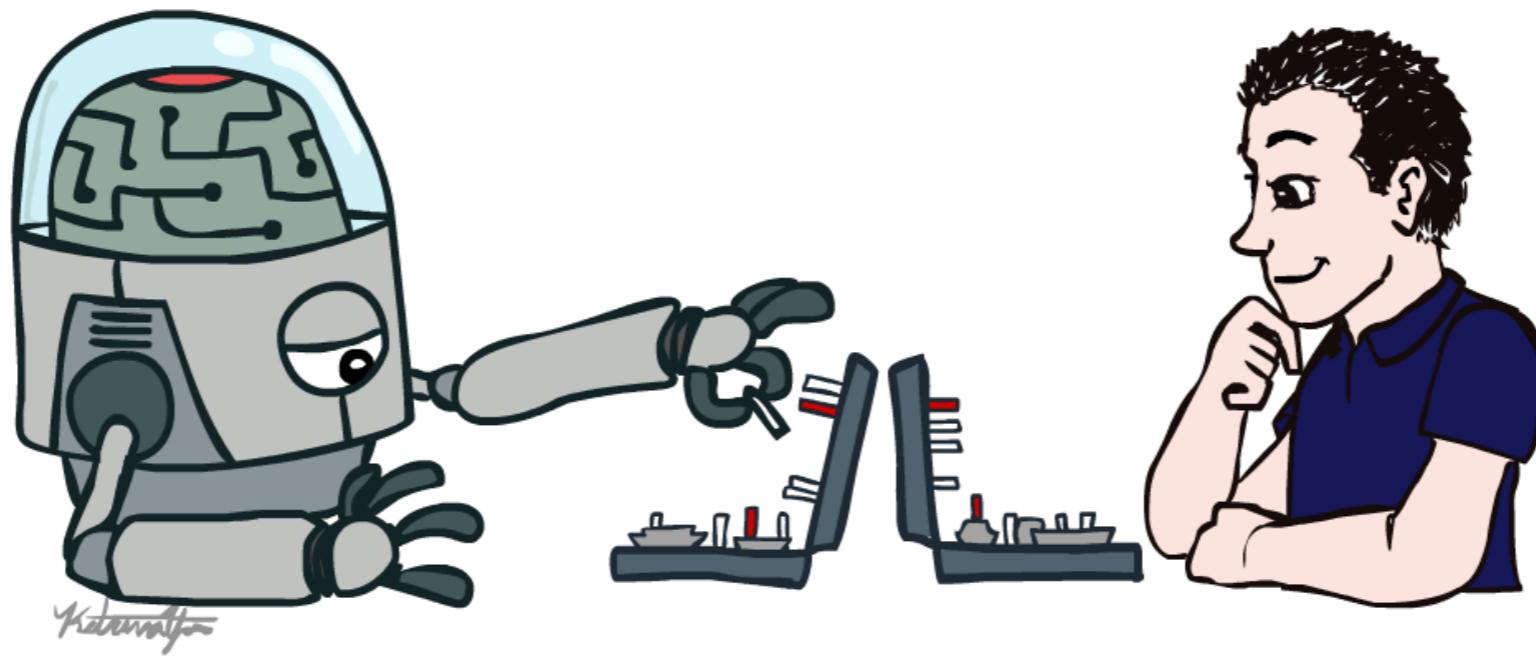
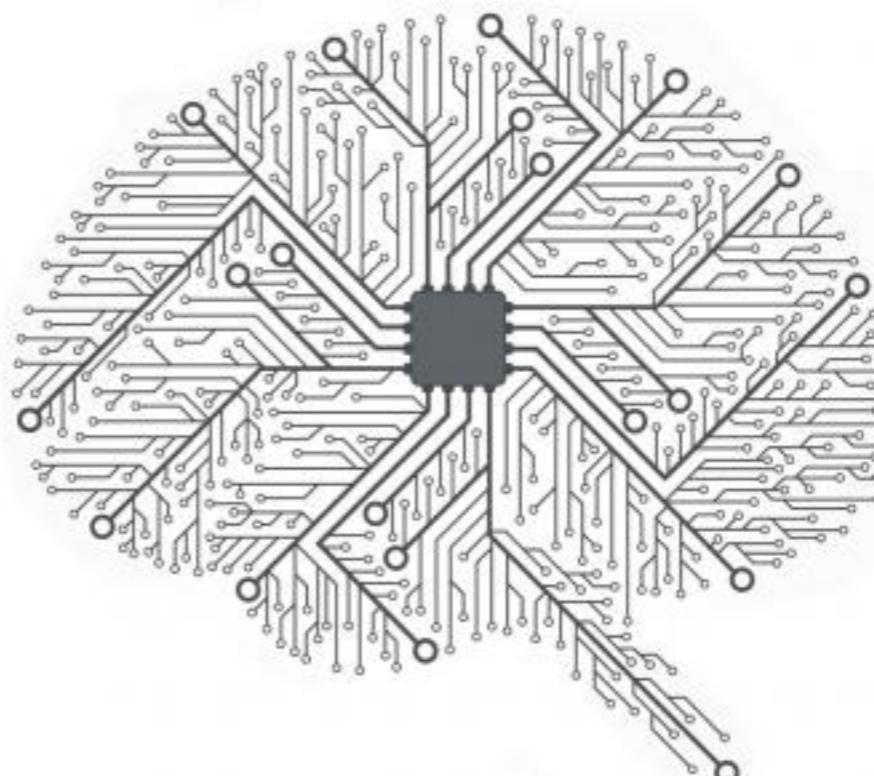


人工智能



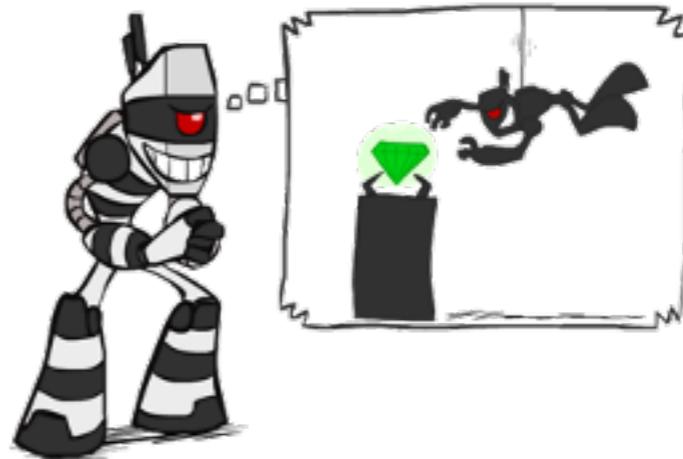
第四章·约束满足问题

- 约束满足问题
- 求解约束满足问题
- 回溯搜索
- 过滤、排序和结构优化
- 迭代改进

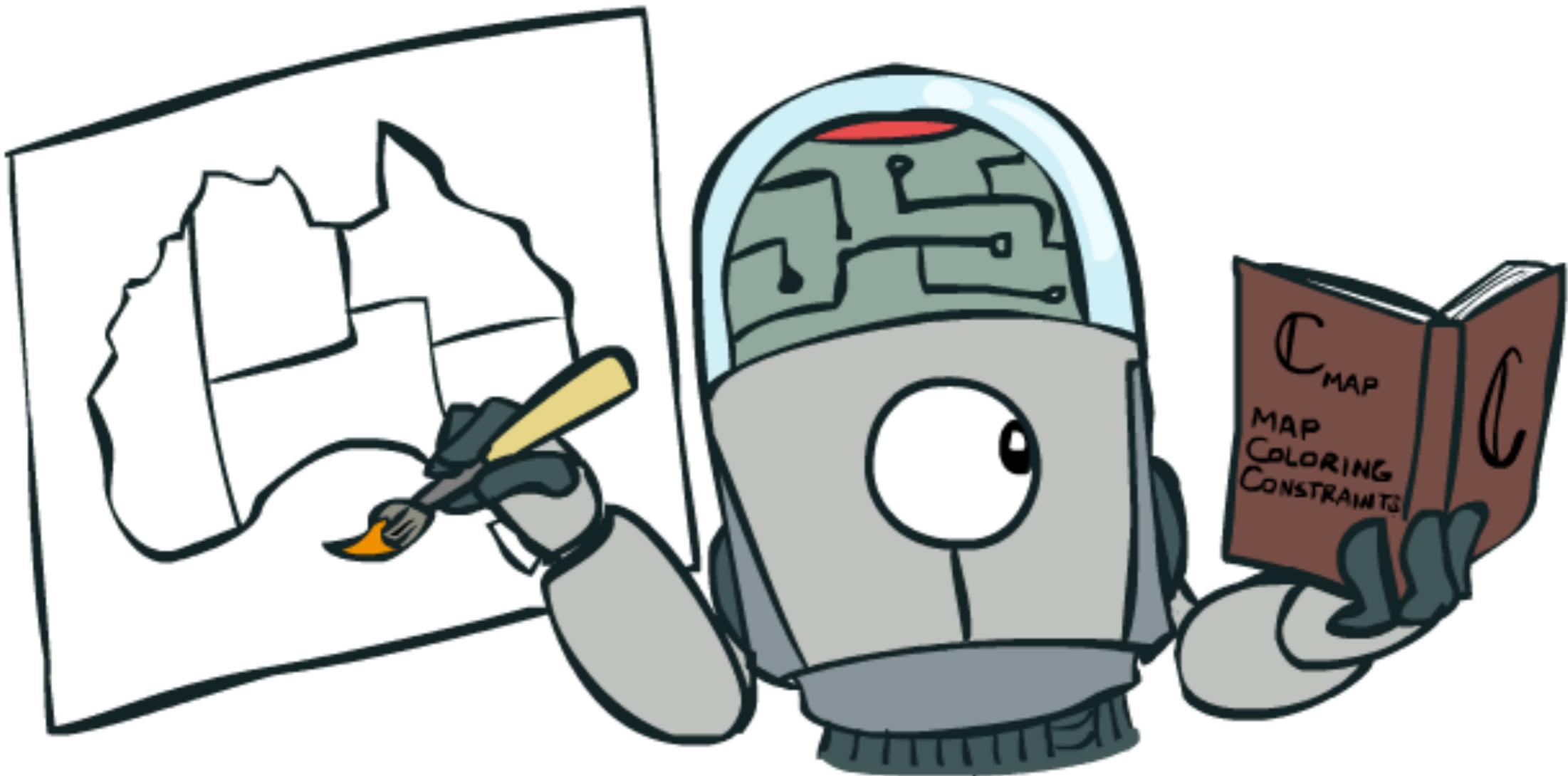


两种问题的求解

- 关于世界的假设：一个单一的智能体，确定性的行动，完全观察到的状态，离散的状态空间
- 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深度
 - 启发式函数给出具体问题的指导
- 识别：变量的特定值
 - 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

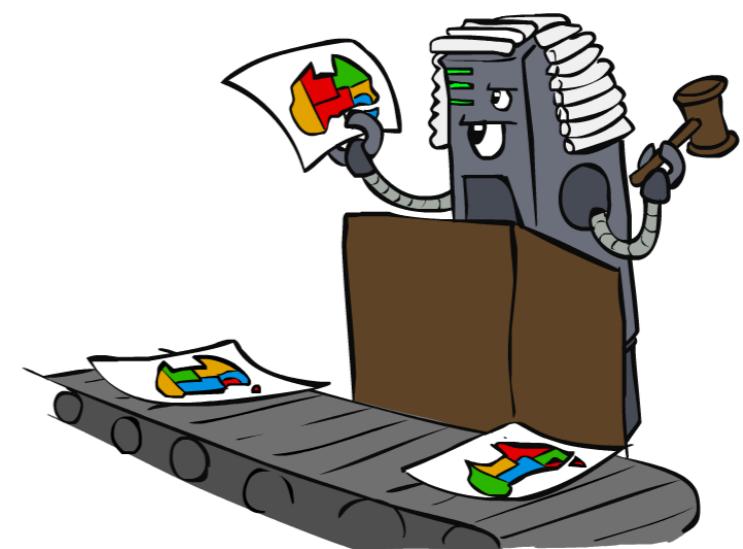


约束满足问题 (Constraint Satisfaction Problems)



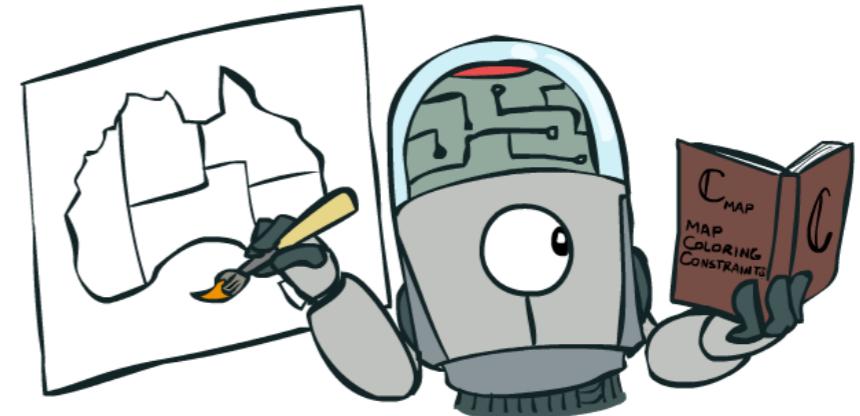
约束满足问题

- 标准的搜索问题:
 - 状态是任意的数据结构
 - 目标测试可以是任何函数
 - 后继函数也可以是任意的
- 约束满足问题:
 - 是搜索问题的一个特殊子集
 - 状态由若干个变量 X_i 组成，每个变量有一个取值域 D (有时 D 与 i 相关)
 - 目标测试是一组约束，指定变量子集的值的允许组合

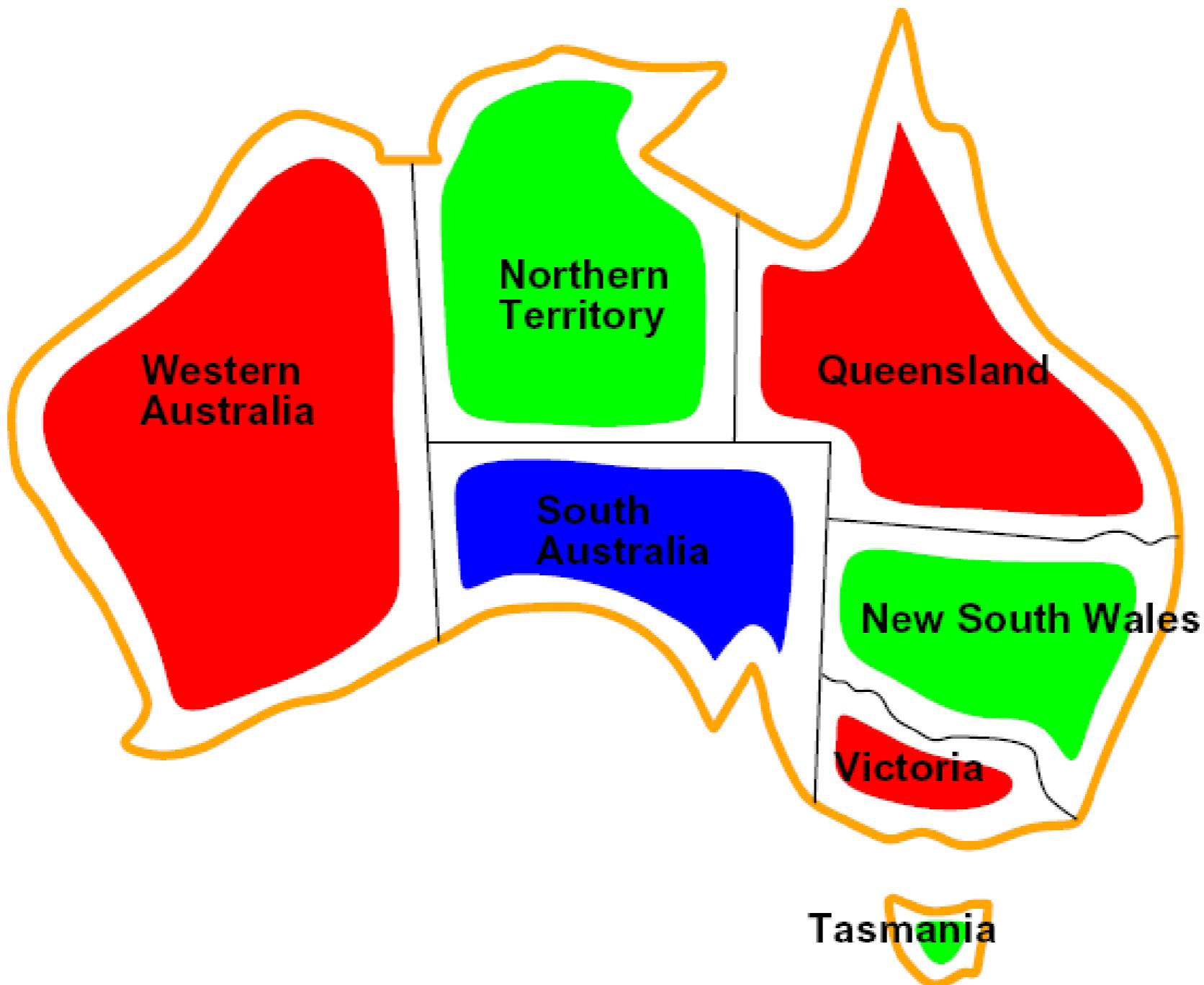


约束满足问题

- 到目标的路径(变量赋值的顺序)不重要
- 约束满足问题算法比标准的搜索问题要快
- 启发信息可用于所有约束满足问题问题

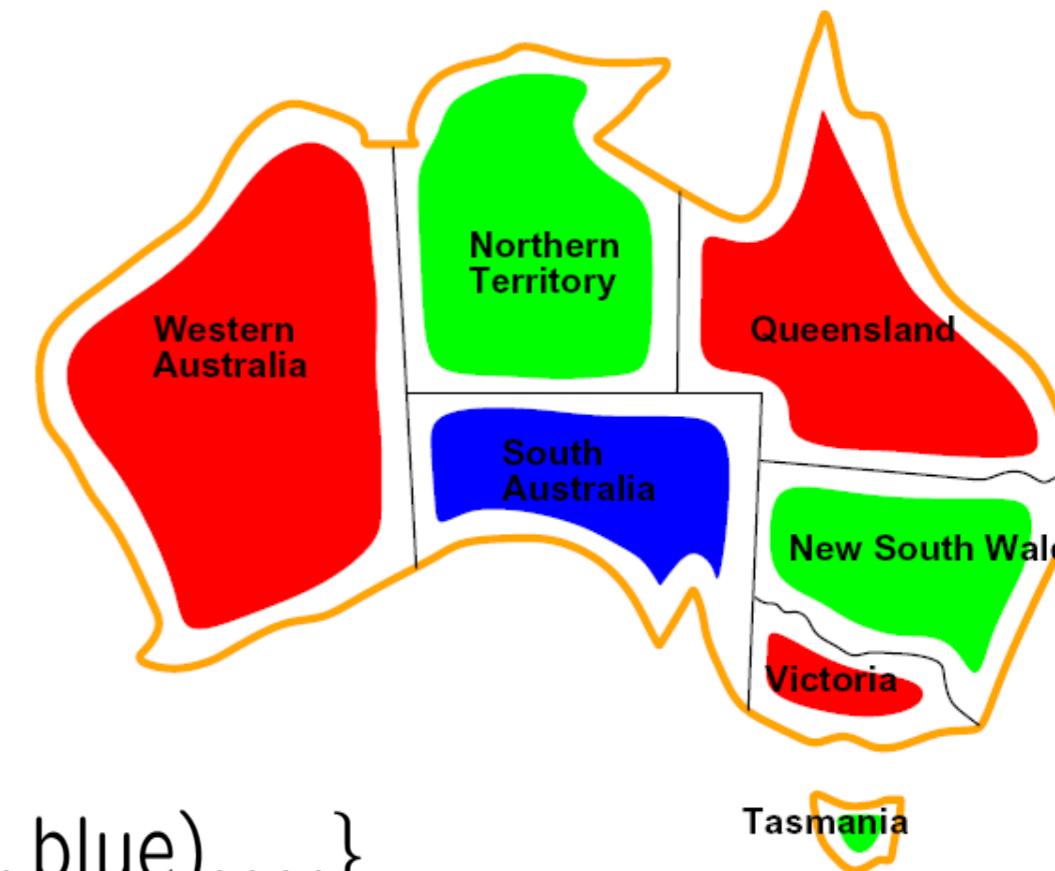


举例：地图着色



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赋值, 满足了所有的约束, 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:

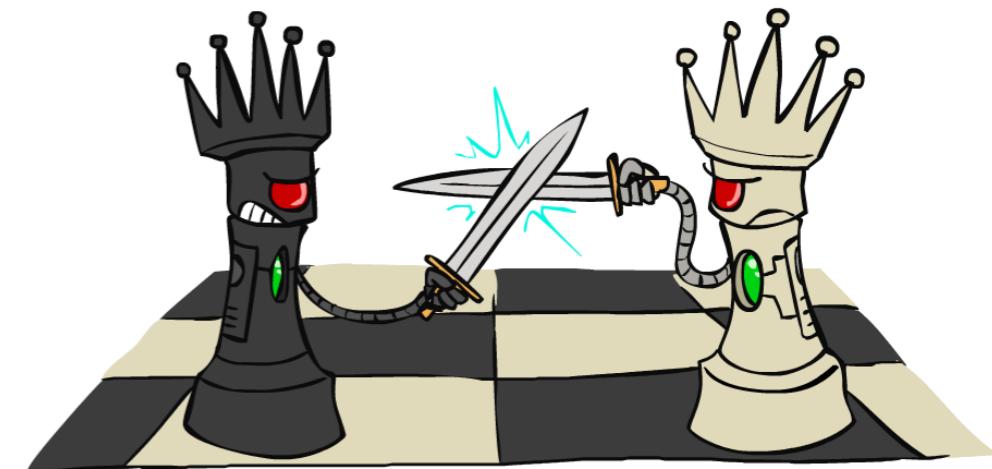
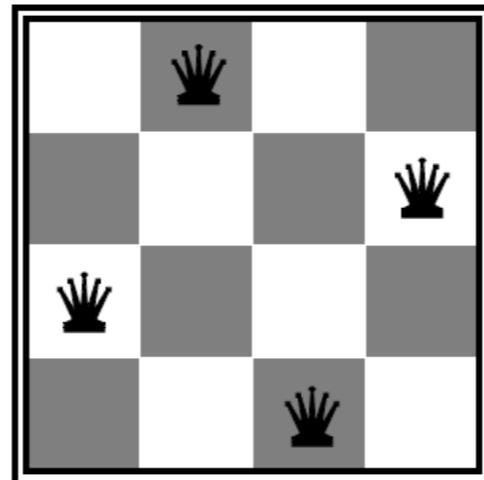
$$\sum_{i,j} X_{ij} = N$$

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

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

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

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



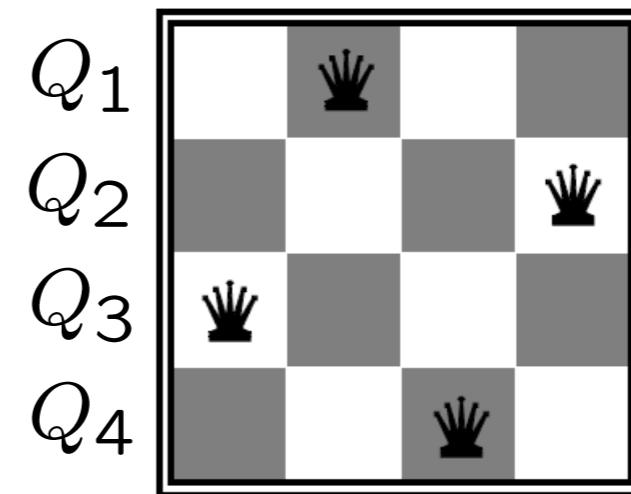
赋值选择空间: 2^{N^2}

举例: N-皇后

- 描述方法 2:

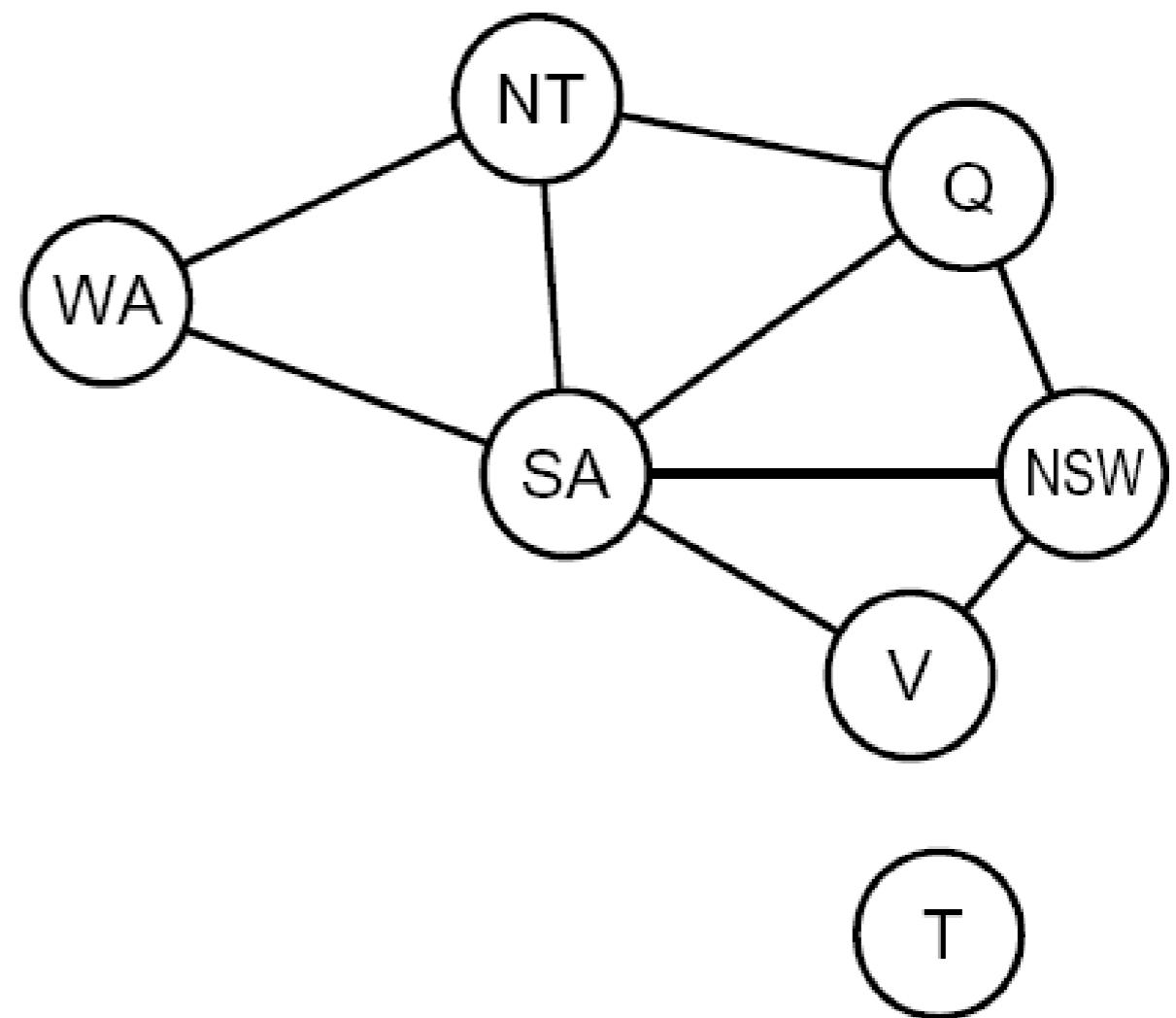
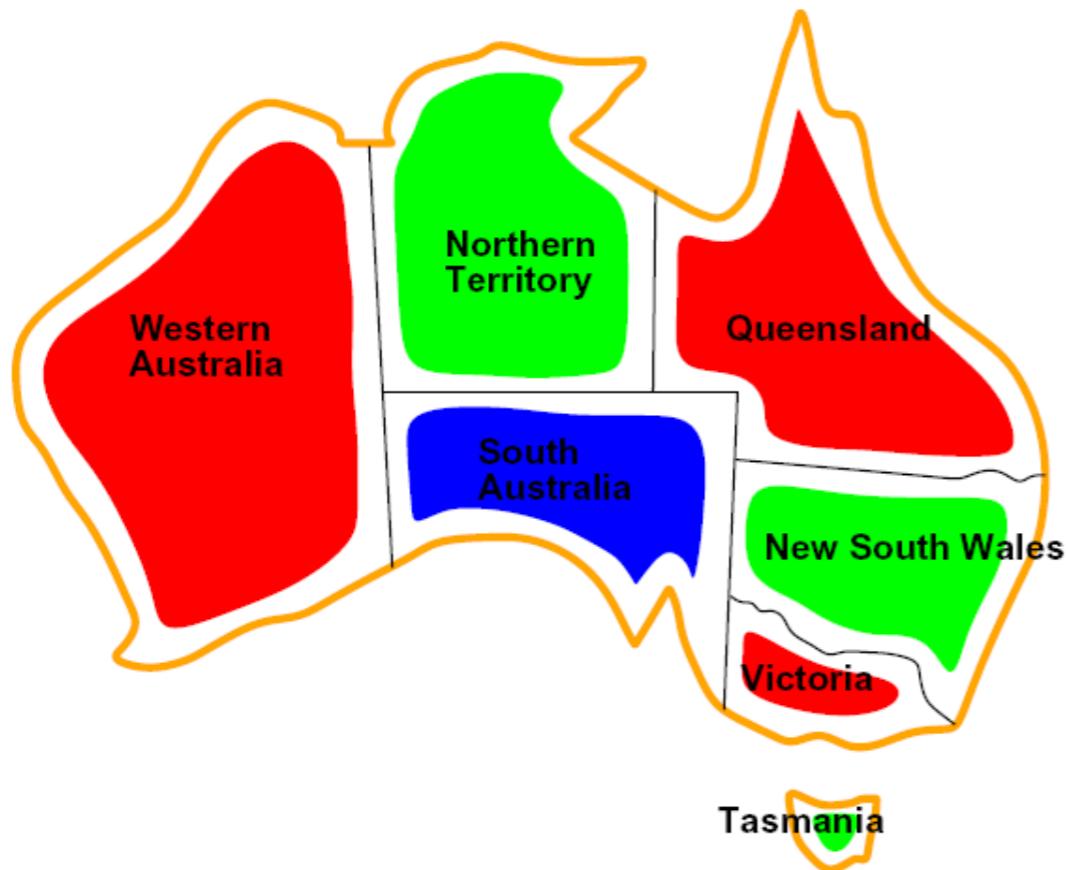
- 变量: Q_k
- 值域: $\{1, 2, 3, \dots, N\}$
- 约束条件:
 - 隐式: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$
 - 显式: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\}$

...



赋值探索空间 N^N

约束图

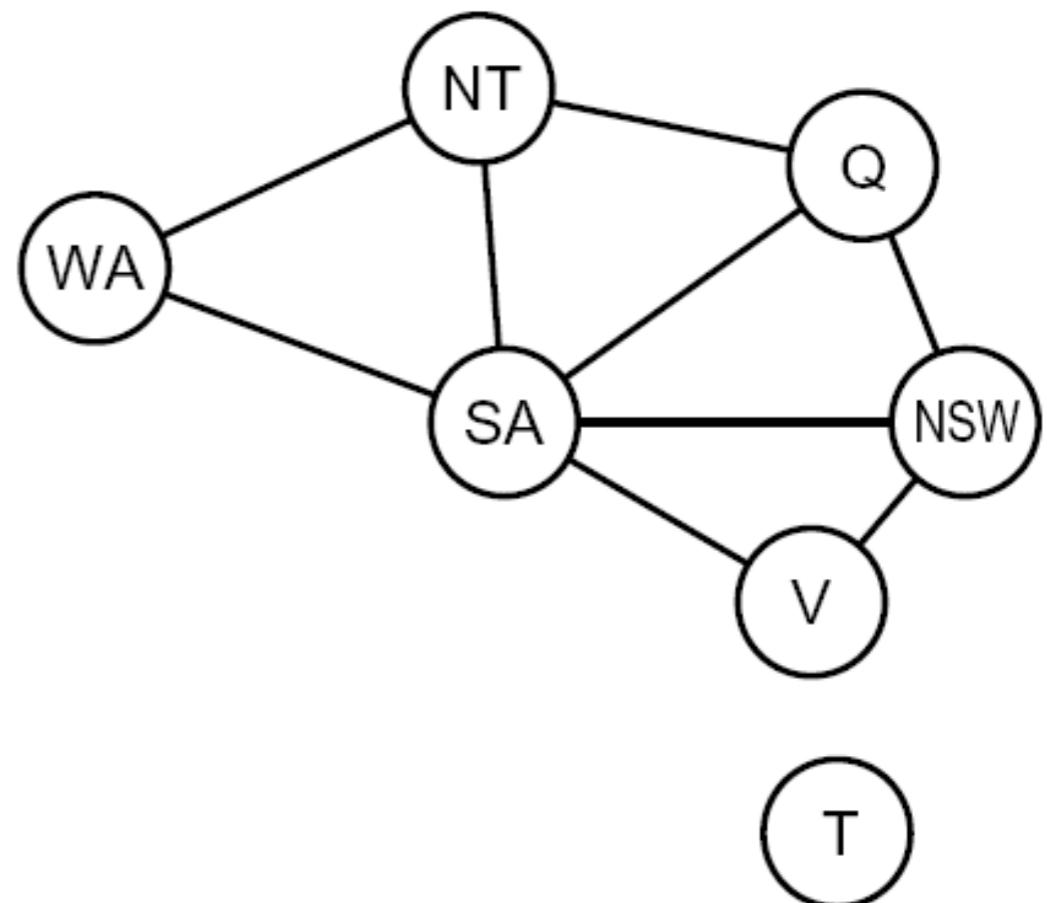


节点：变量

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)
- 每一个非二元CSP可以被转化为一个二元CSP(可能会增添变量)



举例:密码算术

- 变量:

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

- 值域:

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

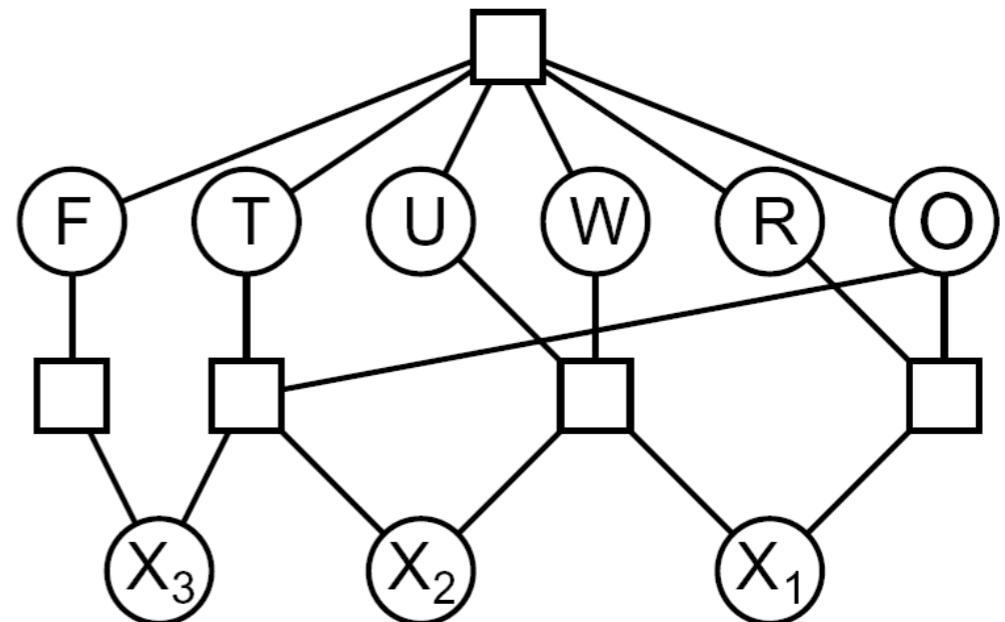
- 约束条件:

$\text{alldiff}(F, T, U, W, R, O)$

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

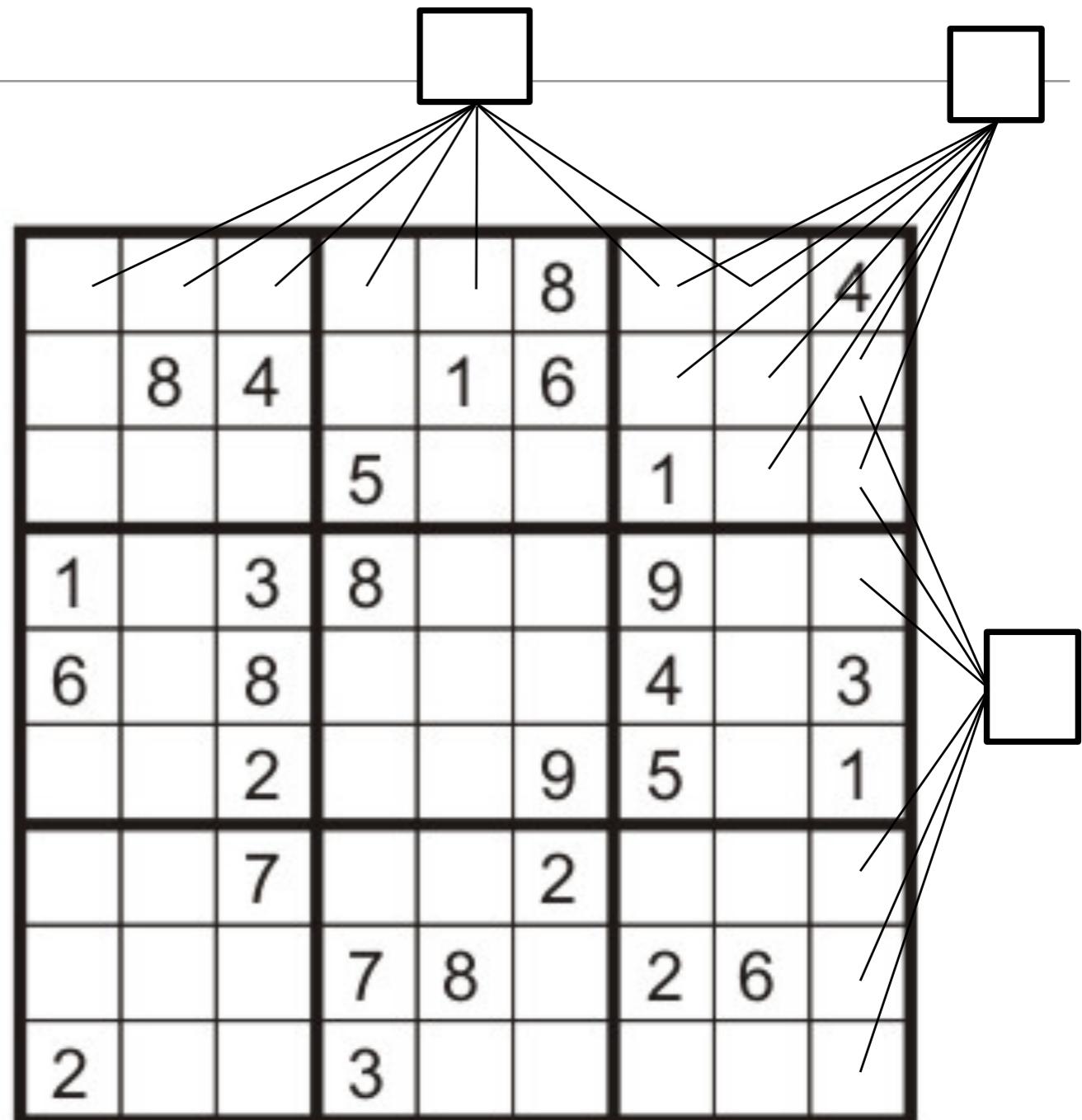
...

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$



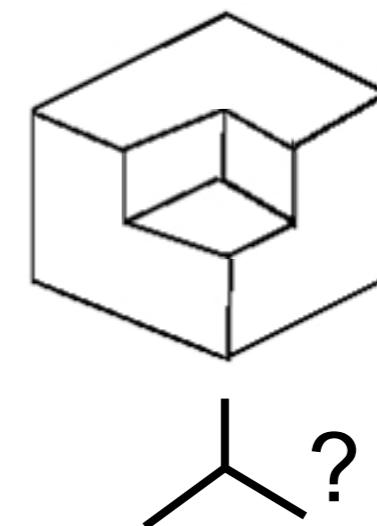
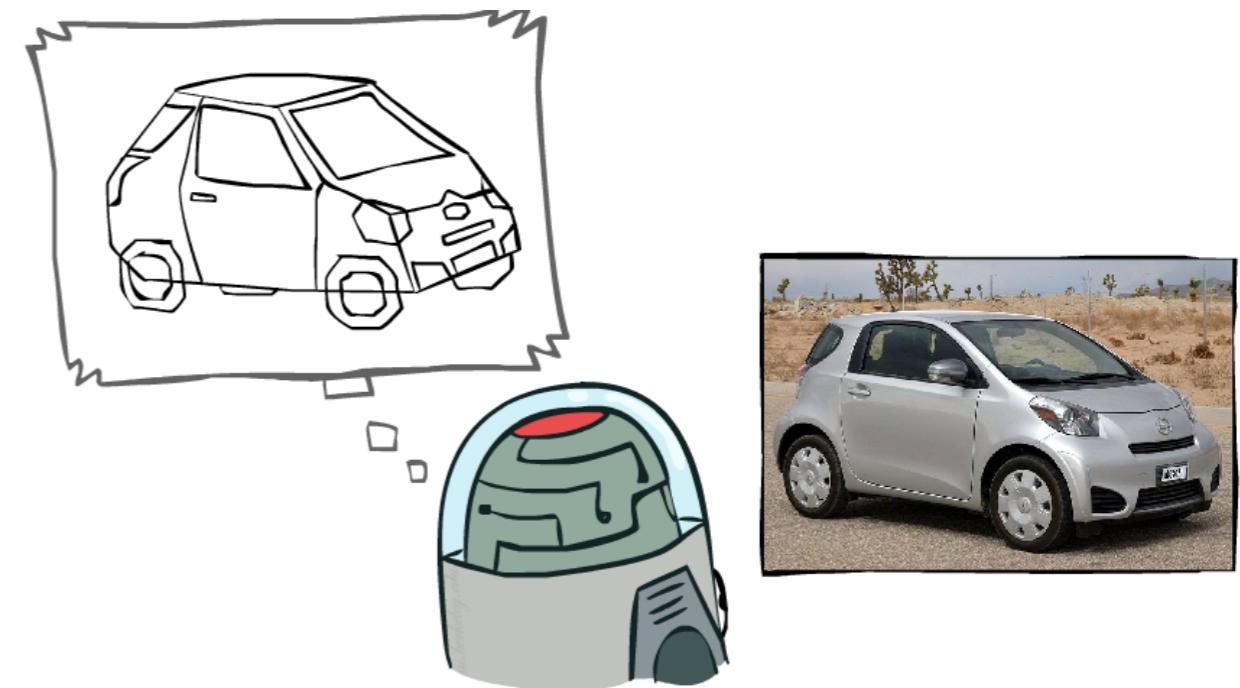
举例: 数独(Sudoku)

- 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 3×3 大方格里的9个数字都不同

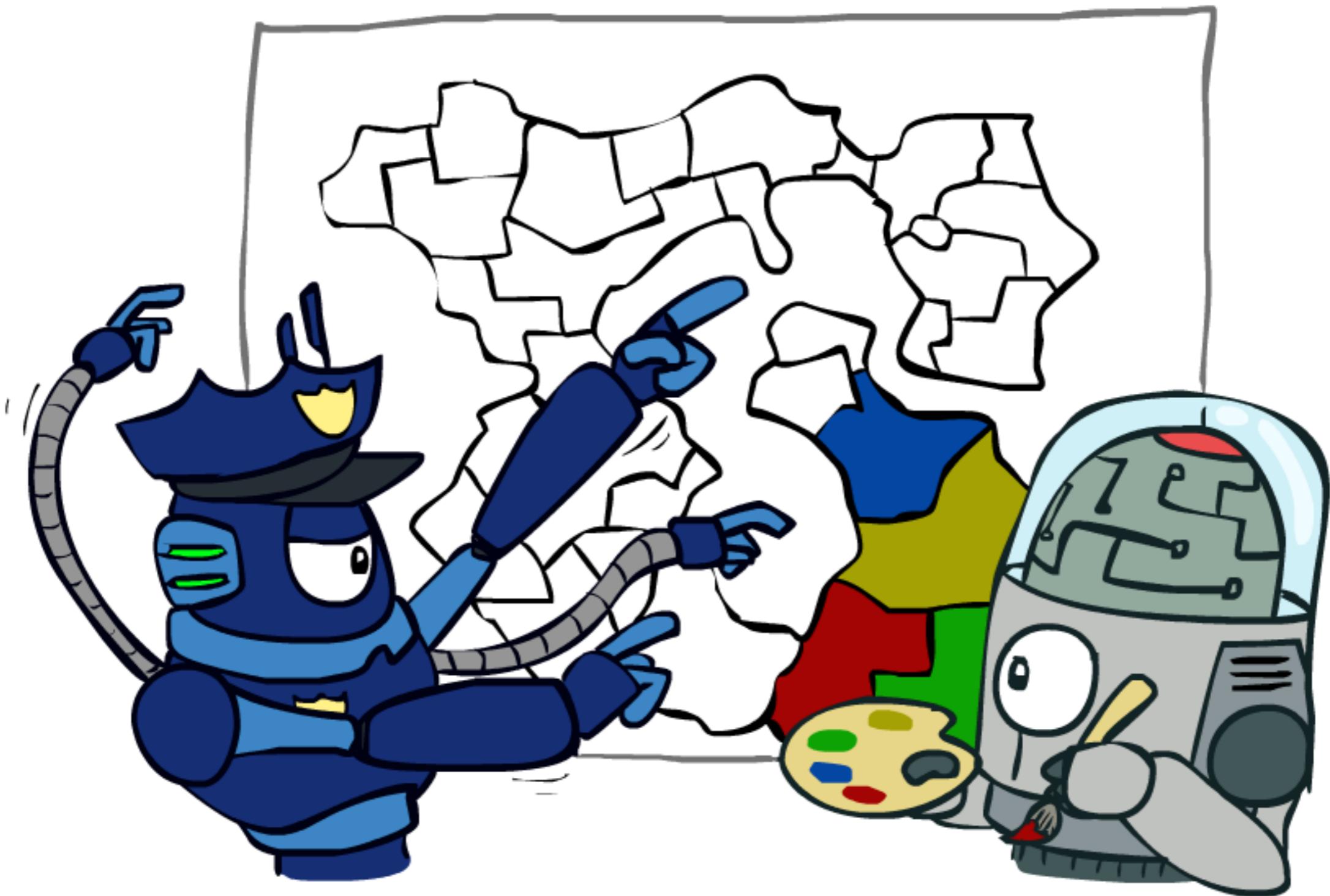


举例：华尔兹算法

- 华尔兹算法是用于将实心多面体的线条图解释为三维对象
- 这个早期的人工智能的实例导致了CSP的提出
- 具体的：
 - 每个交叉点是一个变量
 - 相邻的交叉点相互之间互为限制
 - 解决方案是给出物理上可实现的一个三维解释

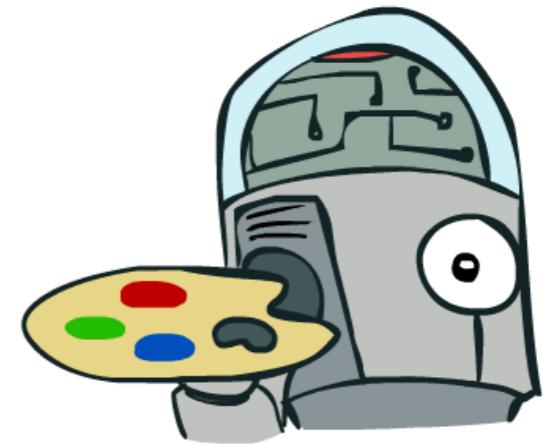


约束满足问题和约束条件的多样性



约束满足问题的多样性

- 离散变量
 - 有限值域, n 变量, 值域大小是 d
 - $O(d^n)$ 完整的赋值复杂度
 - 比如, 布尔可满足性问题(SAT), $d=2$, 约束是逻辑子句
 - 无限值域 (整数, 字符串, 等.)
 - 比如, 任务调度, 变量是每个任务的开始时间
- 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



约束条件的多样性

- 多样的约束

- 一元约束, 涉及一个变量(相当于缩减值域), 比如:

$$SA \neq \text{green}$$

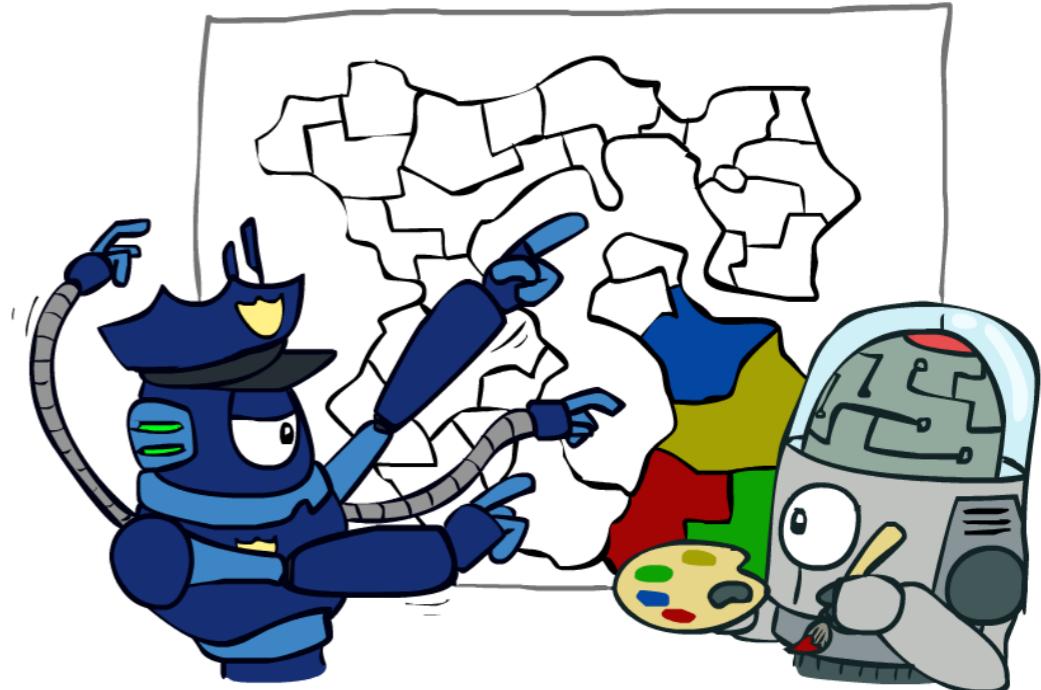
- 二元约束, 涉及成对的变量, 例如:

$$SA \neq WA$$

- 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

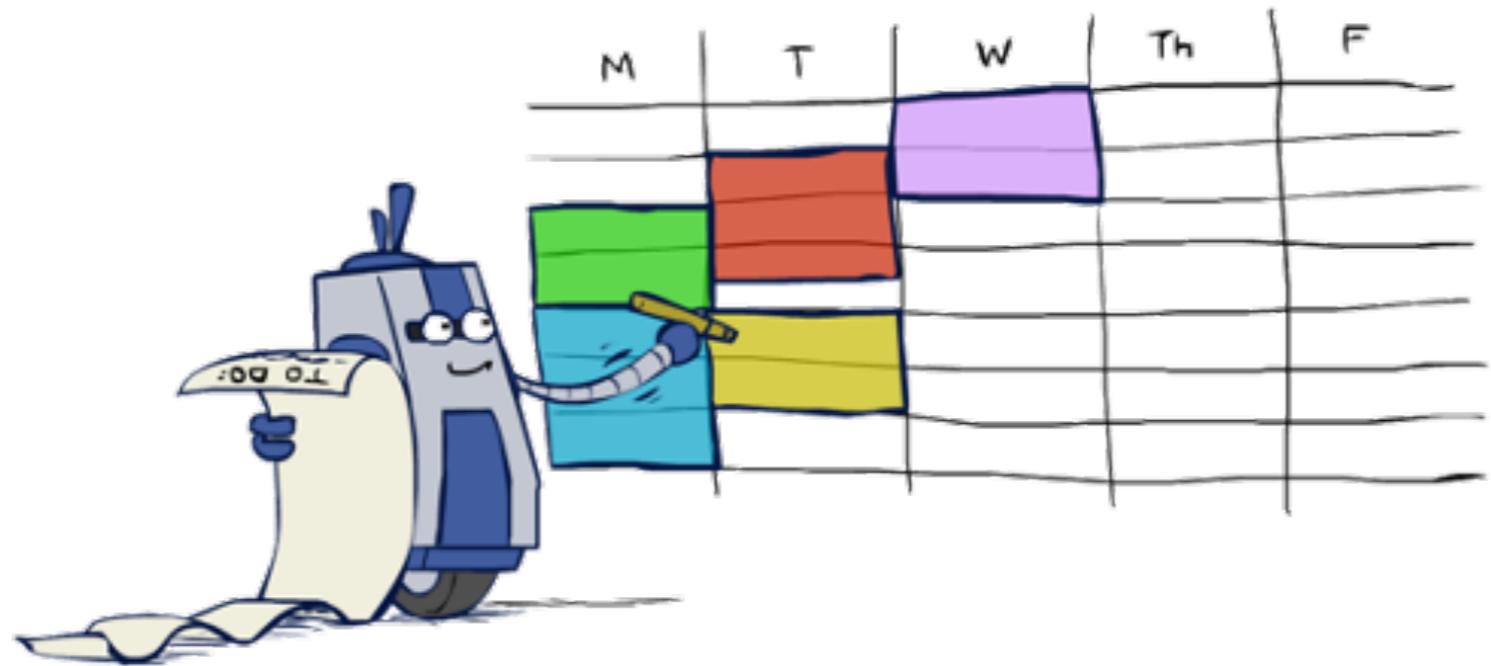
- 偏好约束 (软约束):

- 比如, 红色比绿色好
 - 可用成本函数对赋值组合进行评估
 - 这种情况也叫约束性的优化问题



真实世界中的约束满足问题

- 课程计划问题: 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
 - 公交时间调度
 - 工厂时间调度
 - 电路设计规划
 - 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...

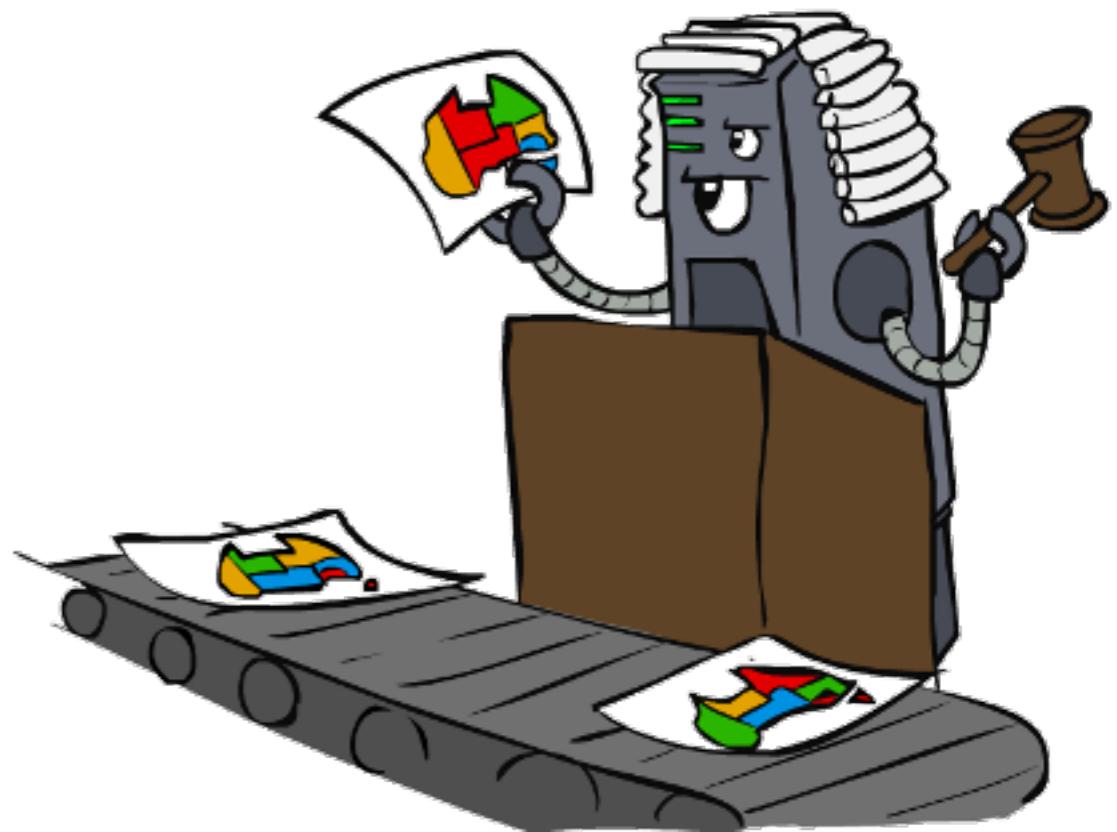


求解约束满足问题



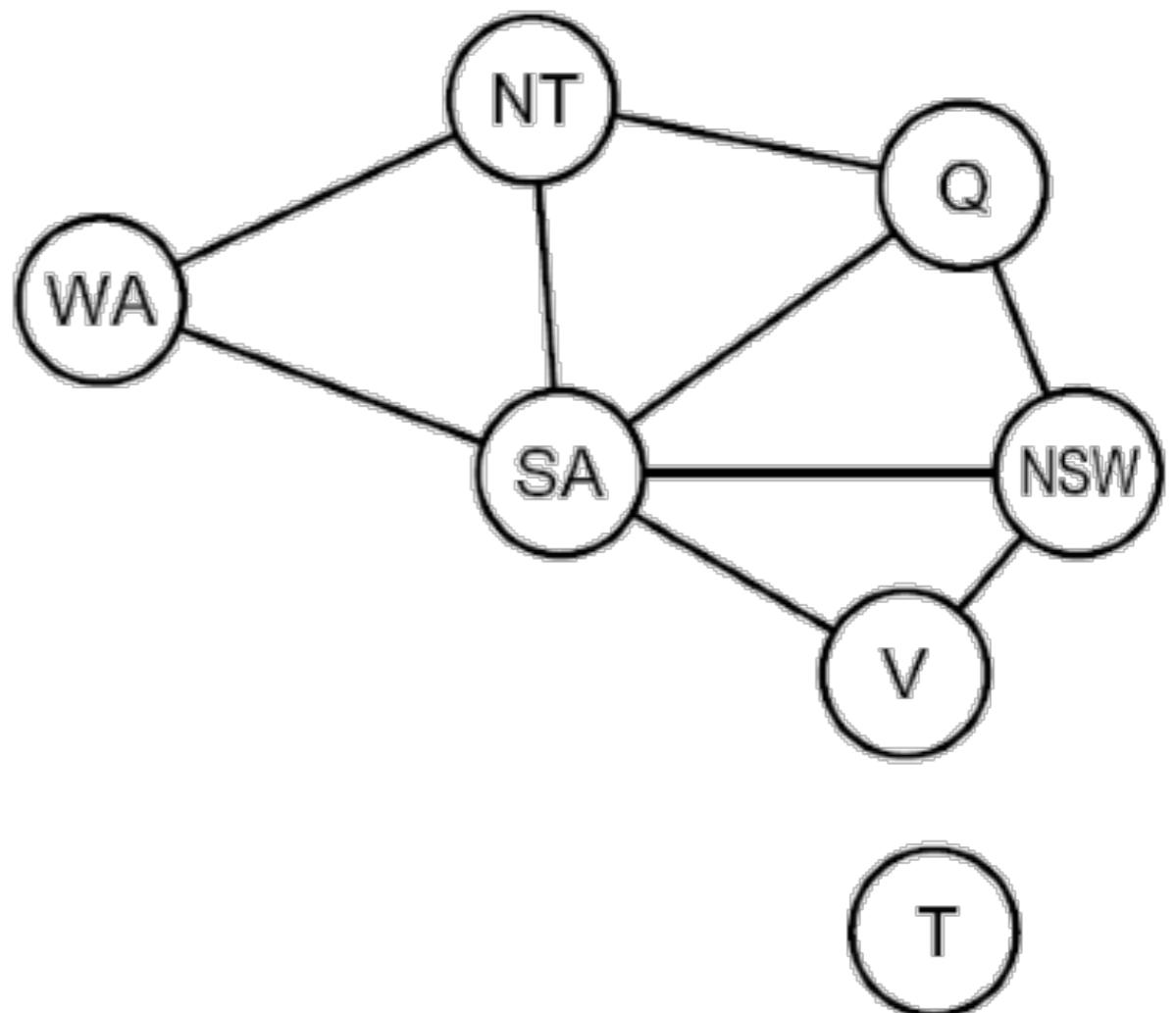
按标准的搜索问题来解决CSP

- 状态反映了变量赋值的当前情况(部分赋值)
 - 初始状态: 没有赋值, {}
 - 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)(即转换模型): 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

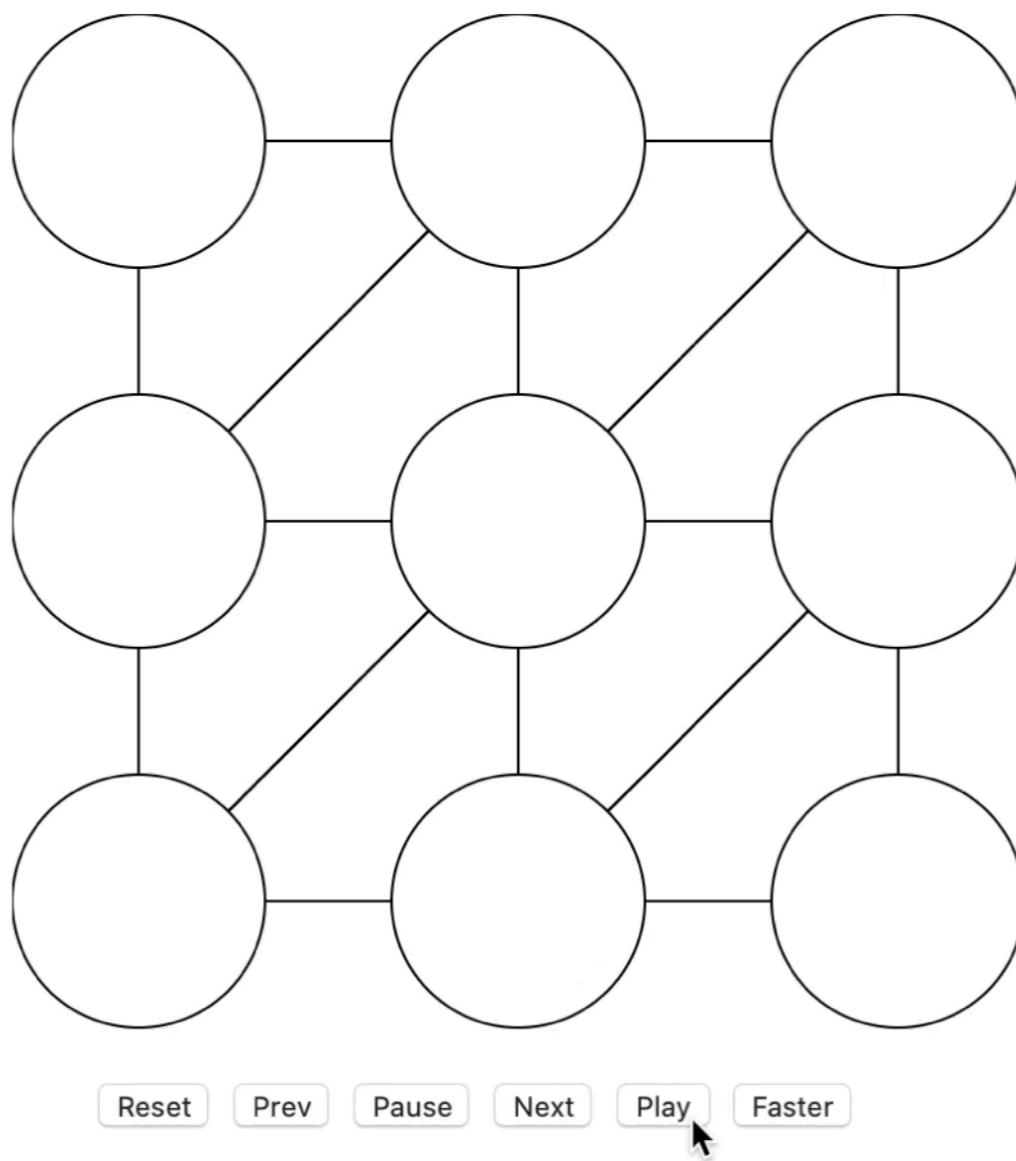


一般搜索方法

- 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?



应用简单的DFS解决图着色问题



Graph
Simple

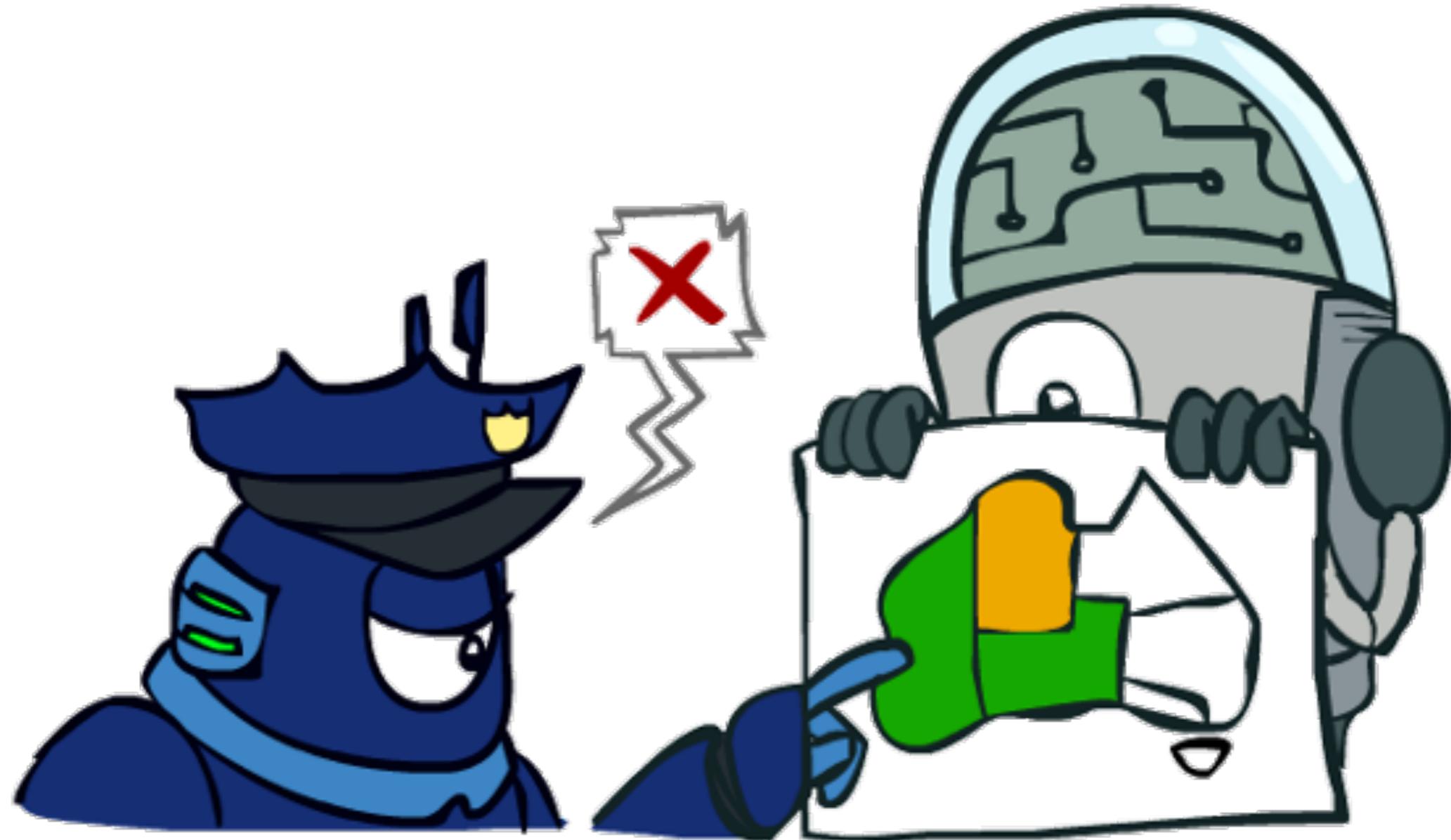
Algorithm
Naive Search

Ordering
 None
 MRV
 MRV with LCV

Filtering
 None
 Forward Checking
 Arc Consistency

Speed
Speedup Frame
1 x Delay
700

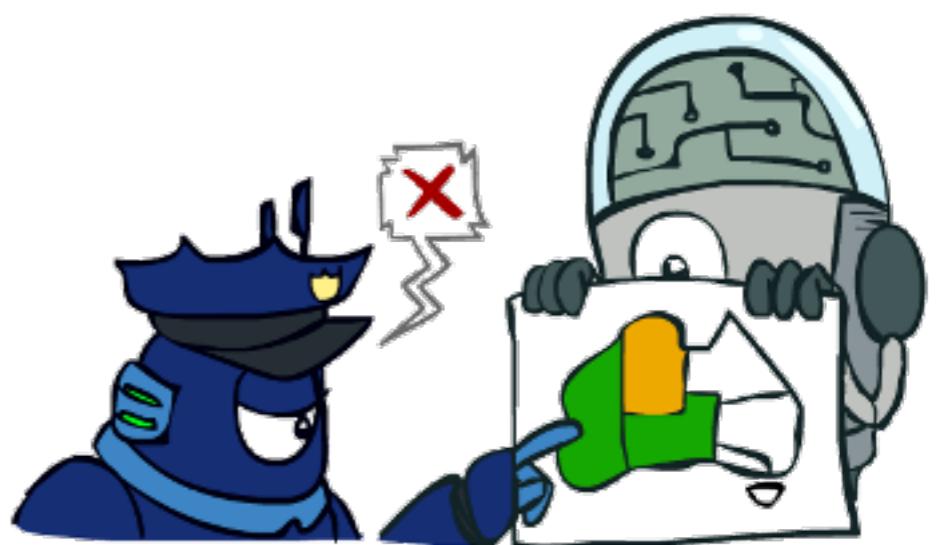
回溯搜索



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
 - 变量赋值是可交换的，所以选择一个顺序固定下来
 - 例如， $[WA = \text{red} \text{ then } NT = \text{green}]$ 和 $[NT = \text{green} \text{ then } WA = \text{red}]$ 是一样的
 - 在每一步只需考虑给一个变量配值：减少分支因子数 b 从 nd 到 d
- 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赋值是否满足约束条件，和之前已赋值的不冲突
 - 也许需要花费一些计算来检查约束条件是否满足
 - 相当于“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 $n \approx 25$

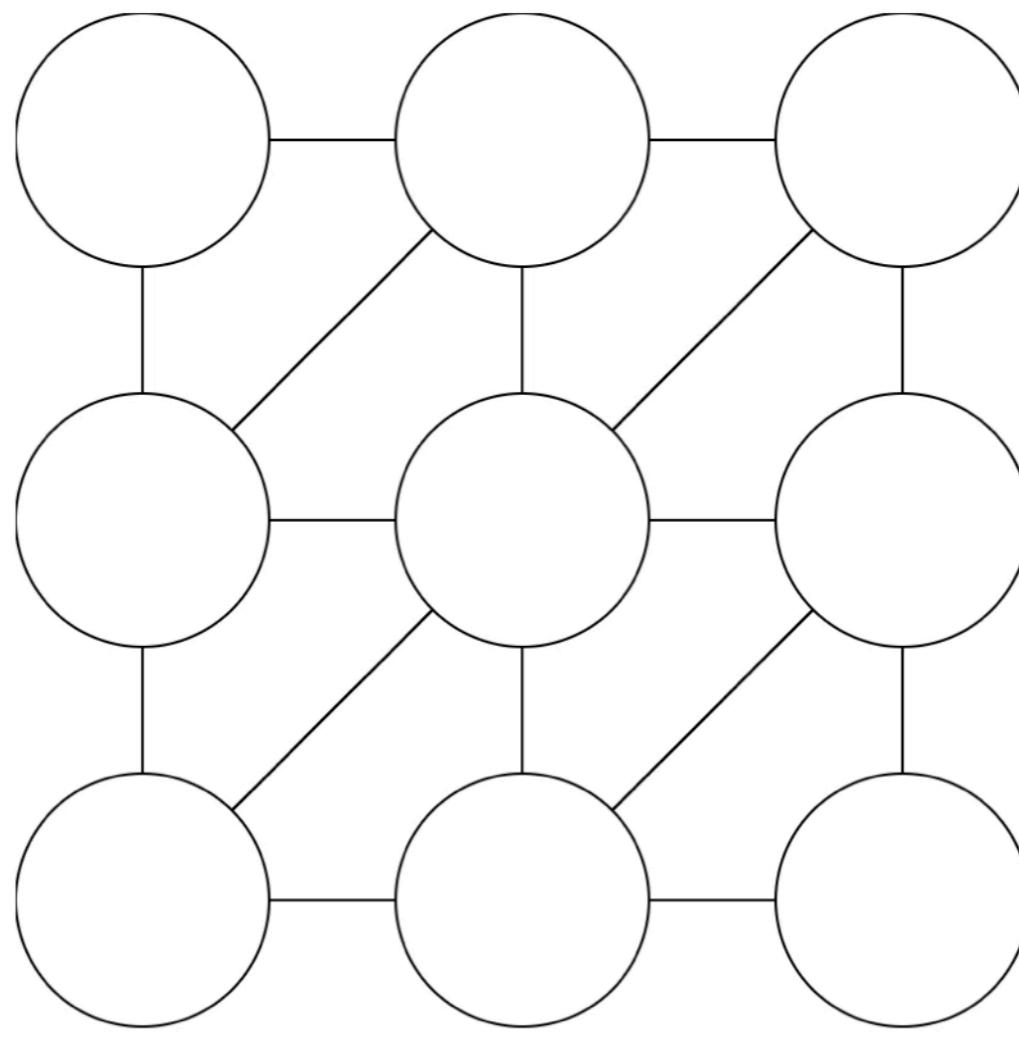
回溯搜索举例



回溯搜索

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add  $\{var = value\}$  to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove  $\{var = value\}$  from assignment
    return failure
```

回溯搜索演示—着色问题



Reset Prev Pause Next Play Faster

Graph

Simple

Algorithm

Backtracking

Ordering

- None
- MRV
- MRV with LCV

Filtering

- None
- Forward Checking
- Arc Consistency

Speed

Speedup Frame
1 x Delay 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(ordering):
 - 下一轮挑选哪个变量进行赋值？
 - 挑选值时，有什么顺序上的考虑？
- 过滤(filtering): 我们能否提前预测不可避免的失败？
- 结构(structure): 我们能否利用问题的结构？

过滤(Filtering)

推理 \leftarrow INFERENCE (csp, var, assignment)

if 推理 \neq 失败 then

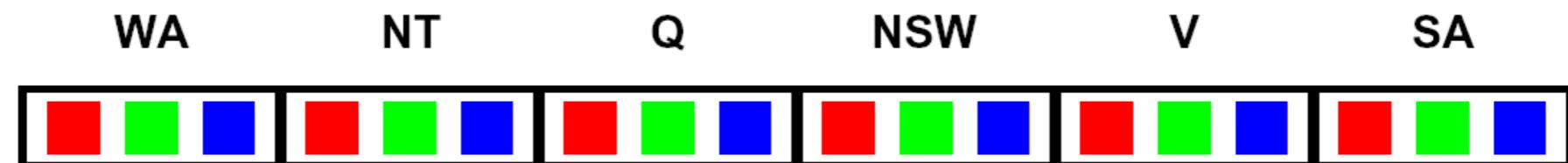
添加 推理 到 assignment

...

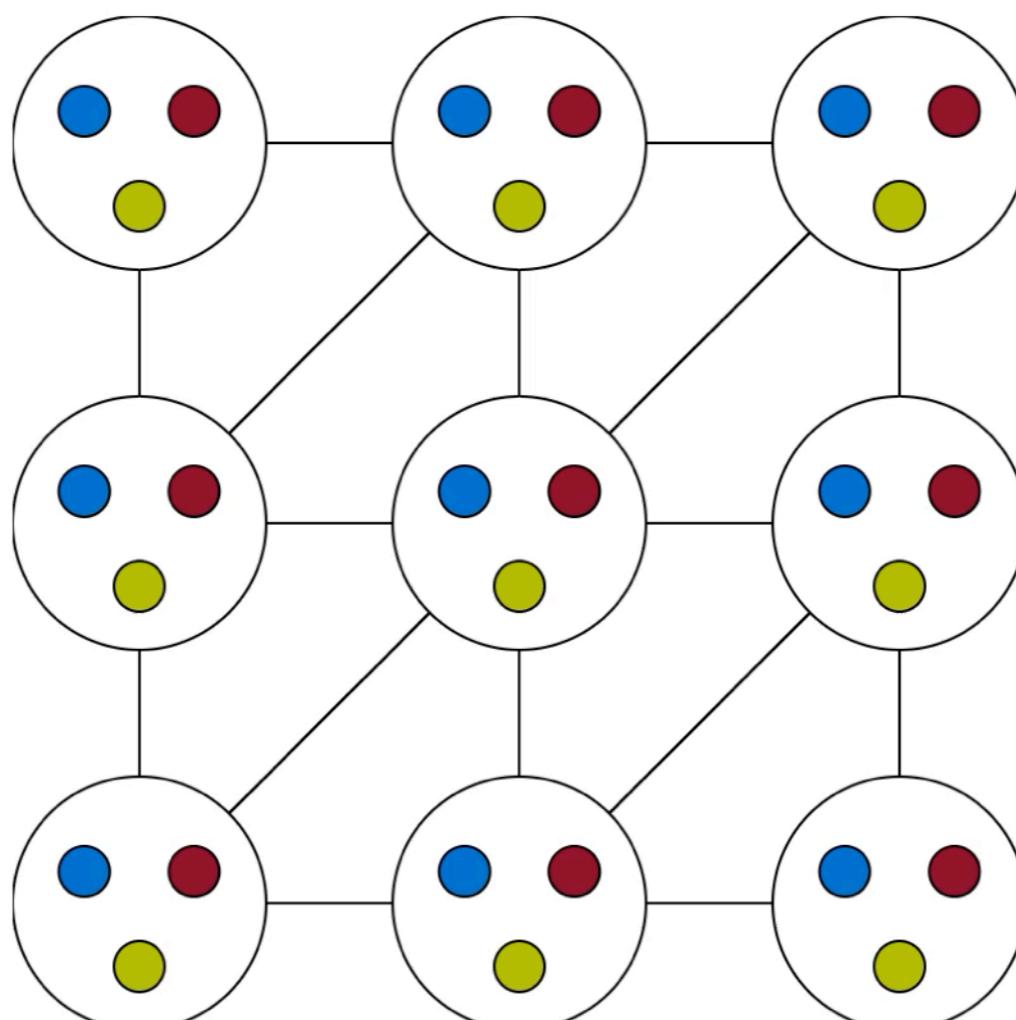


过滤: 前向检查法(Forward Checking)

- 过滤: 搜索中, 持续检测未赋值变量的值域, 去掉违反约束条件的值
- 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



图着色问题演示 - 回溯搜索+前向检查法



Reset Prev Pause Next Play Faster

Graph
Simple

Algorithm
Backtracking

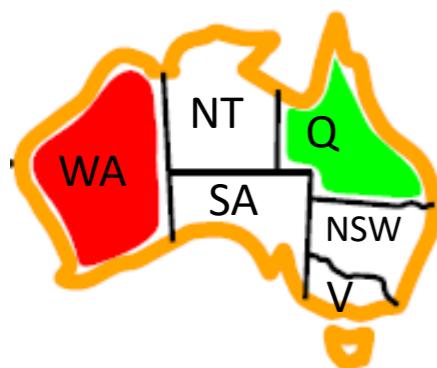
Ordering
 None
 MRV
 MRV with LCV

Filtering
 None
 Forward Checking
 Arc Consistency

Speed
Speedup Frame
1 x Delay
700

过滤:约束传播(Constraint Propagation)

- 前向检查传播已分配到未分配变量的信息，但不提供对所有故障的早期检测：

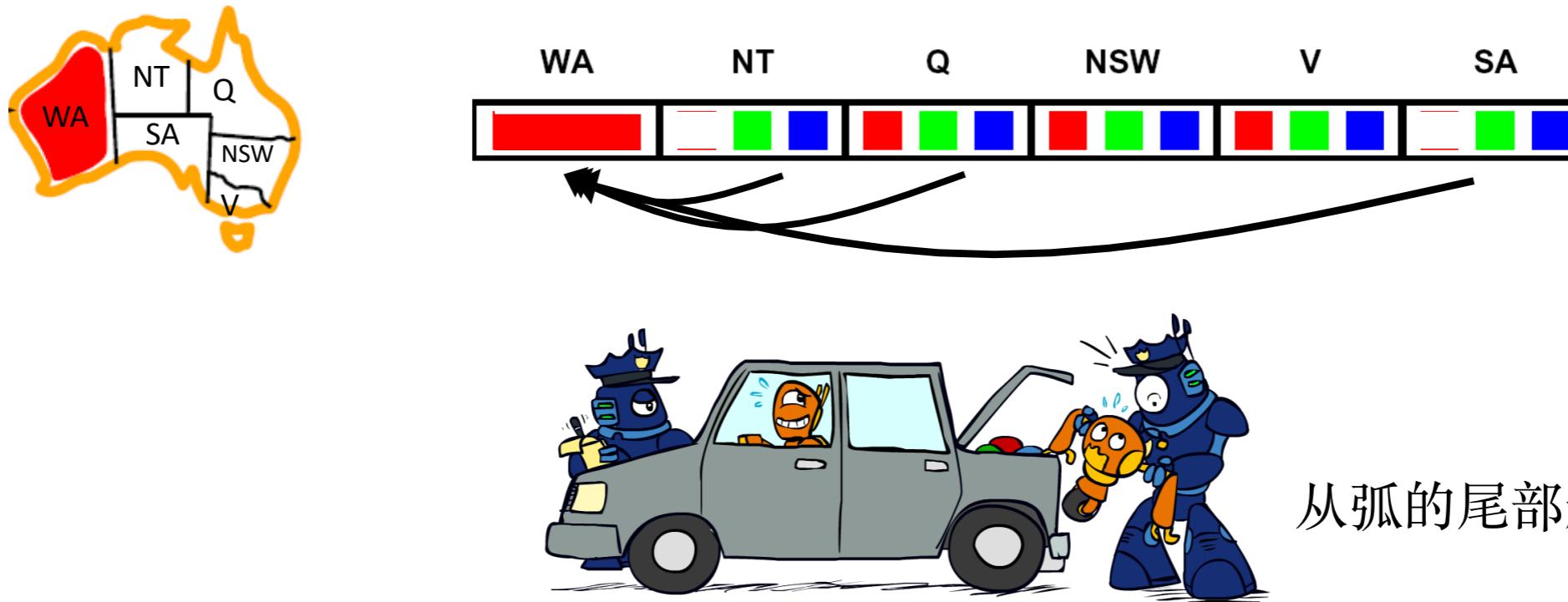


| WA | NT | Q | NSW | V | SA |
|-----|-------|-------|------|-------|-------|
| Red | Green | Blue | Red | Green | Red |
| Red | | Green | Blue | Red | Green |
| Red | | Blue | | Red | |

- NT和SA不可能都是蓝色的!
- 为什么我们还没发现呢?
- 约束传播:从约束到约束的推理

弧的一致性(Arc consistency)

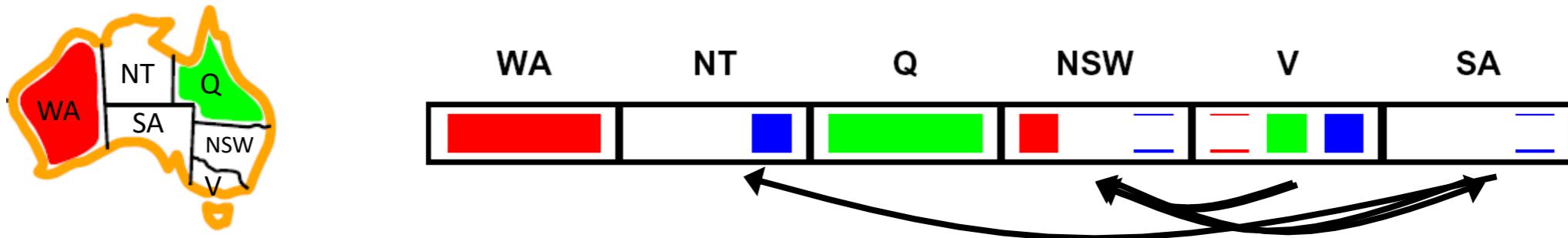
- 一个弧 $X \rightarrow Y$ 是一致的 当且仅当对于X中的每一个 x 值， Y 中存在某个 y 值不违背任何一个约束条件



- 前向检查(Forward checking): 强制检查剩余未赋值变量指向新赋值变量的弧的一致性

整个CSP的弧一致性

- 弧一致性：通过约束传播确保**所有的弧**是一致的：



注意：从弧的尾部删除！

- 重要提示：如果X丢失了一个值，则需要重新检查X的邻居！
- 弧一致性检测故障早于前向检查
- 强制弧一致性的缺点是什么？

在CSP中确保弧一致性

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \dots, X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 

if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**

for each X_k **in** NEIGHBORS[X_i] **do**

 add (X_k, X_i) to *queue*

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds

removed \leftarrow false

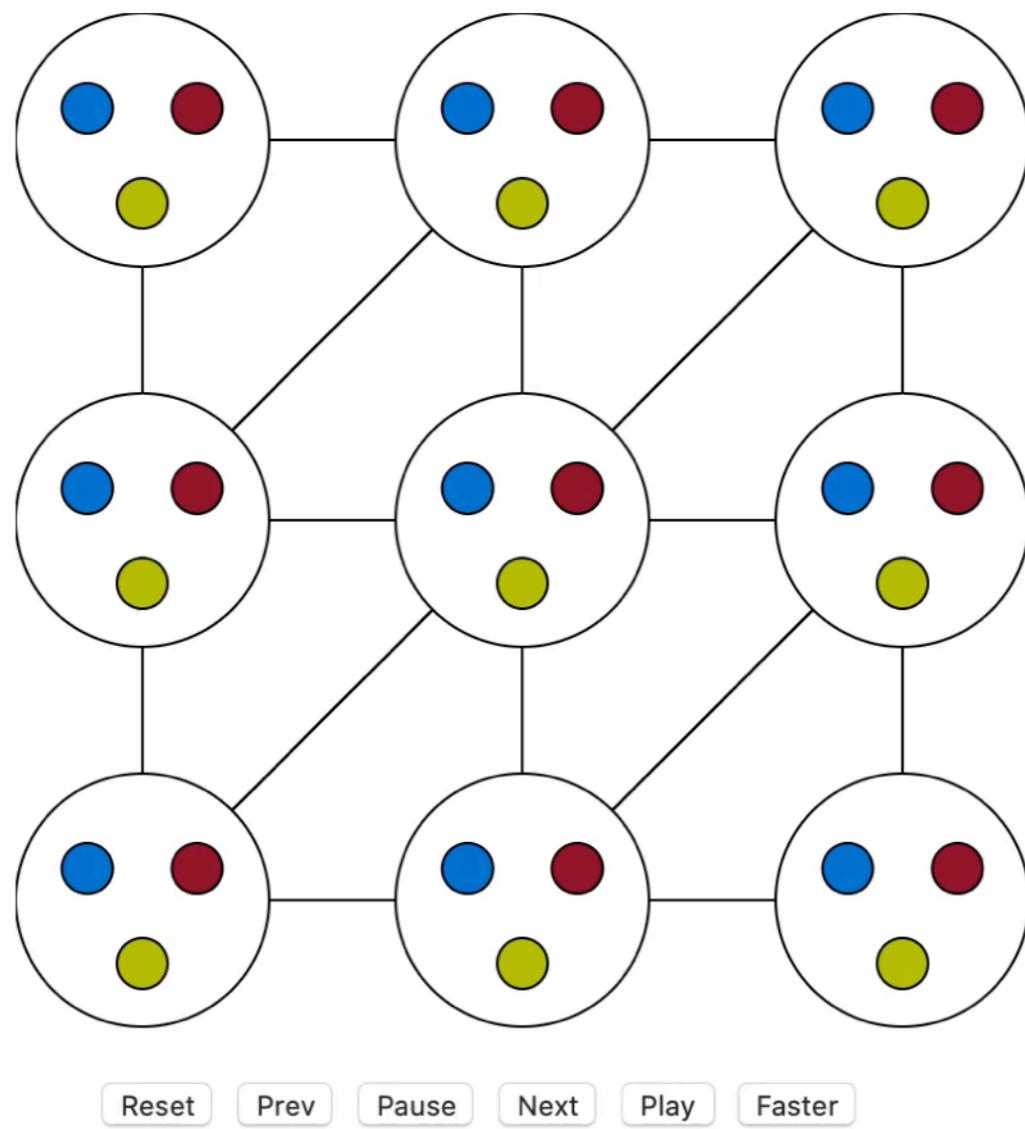
for each x **in** DOMAIN[X_i] **do**

if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint $X_i \leftrightarrow X_j$

then delete x from DOMAIN[X_i]; $\text{removed} \leftarrow \text{true}$

return *removed*

演示：弧一致性



Graph

Simple

Algorithm

Backtracking

Ordering

- None
- MRV
- MRV with LCV

Filtering

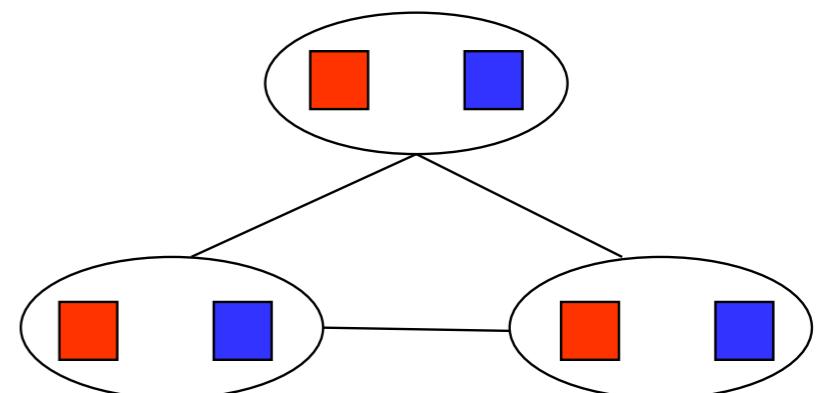
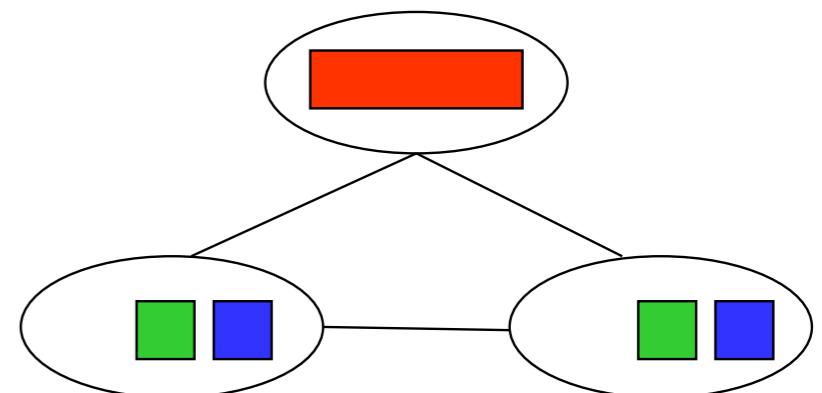
- None
- Forward Checking
- Arc Consistency

Speed

Speedup Frame
1 x Delay
700

弧一致性的局限

- 在确保弧一致性后：
 - 还能有一个解
 - 还能有多个解
 - 没有解（而且不知道）
- 弧一致性仍然需要回溯搜索



为什么会出现问题？

K一致性

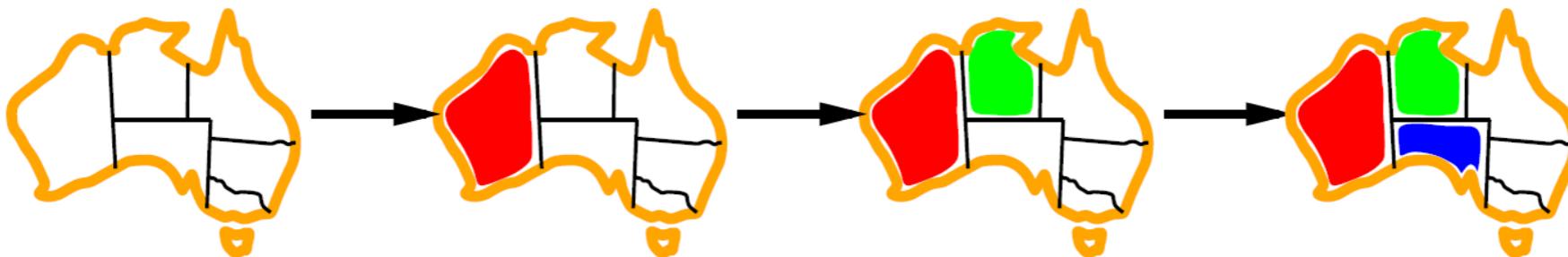
- 提高一致性程度
 - 1一致性（节点一致性）：每个节点的域都有一个值满足该节点的一元约束
 - 2一致性（弧一致性）：对于每对节点，对节点的任何一致分配都可以扩展到另一个
 - K一致性：对于每个k节点，对k-1的任何一致分配都可以扩展到第k个节点。
- K越高，计算成本越高
- 强k一致性：K一致的同时，也k-1、k-2、...1一致

排序



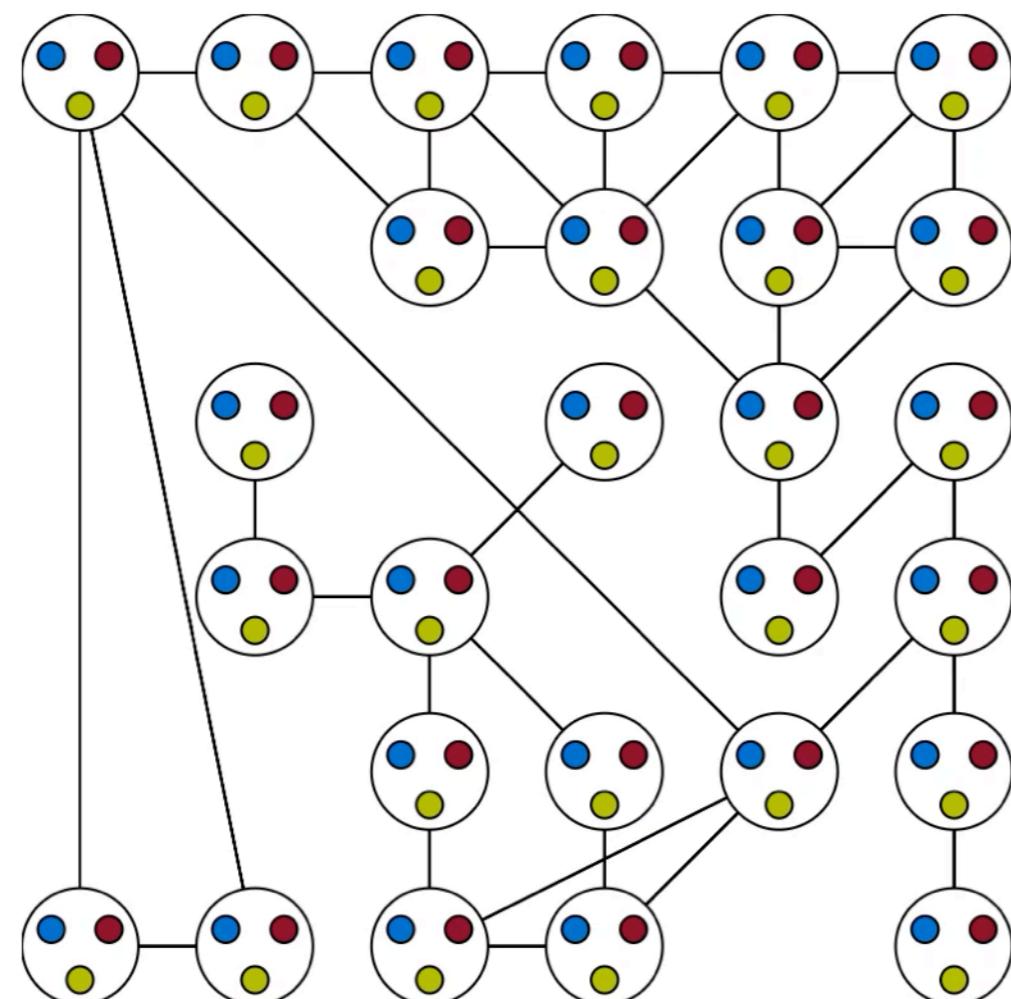
变量排序

- 变量排序: 最小剩余值 (Minimum remaining values -- MRV) 原则:
 - 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(受约束最多的)

演示：大图上的前向检查



Graph
Complex

Algorithm
Backtracking

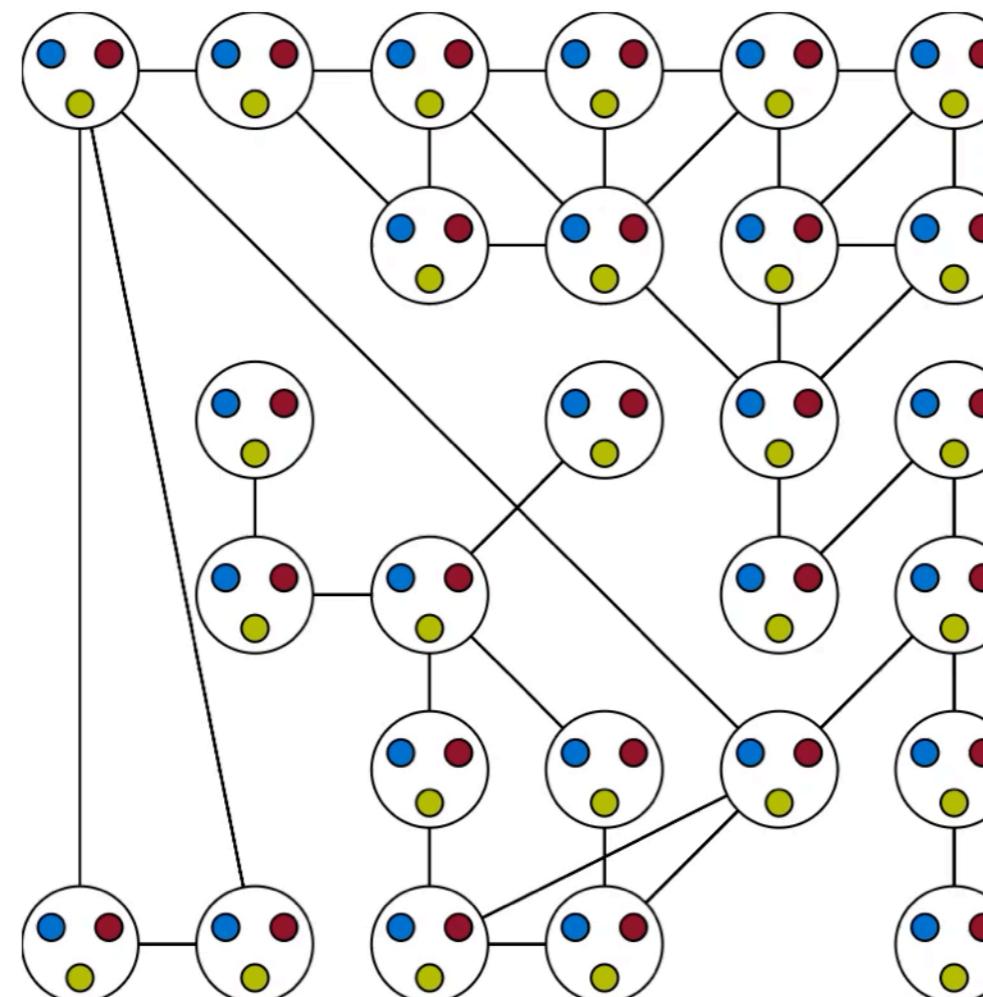
Ordering
 None
 MRV
 MRV with LCV

Filtering
 None
 Forward Checking
 Arc Consistency

Speed
Speedup
Frame
1 x Delay
700

ResetPrevPauseNextPlayFaster

演示：“快速失败”排序



Reset

Graph

Algorithm

Ordering

- None
- MRV
- MRV with LCV

Filtering

- None
- Forward Checking
- Arc Consistency

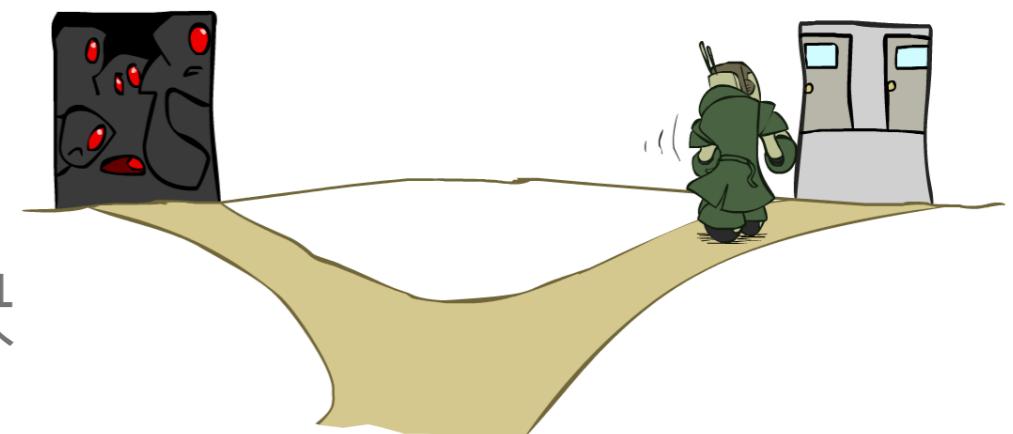
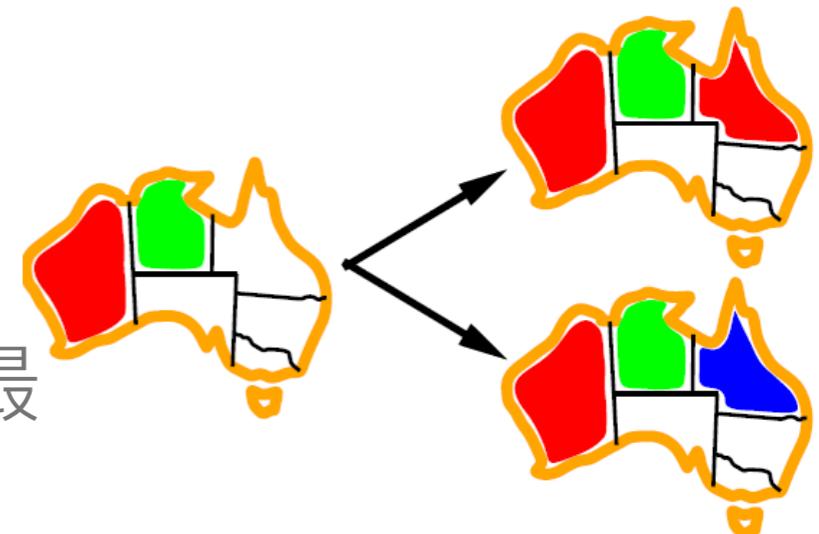
Speed

| | |
|---------|---------|
| Speedup | Frame |
| 1 | x Delay |

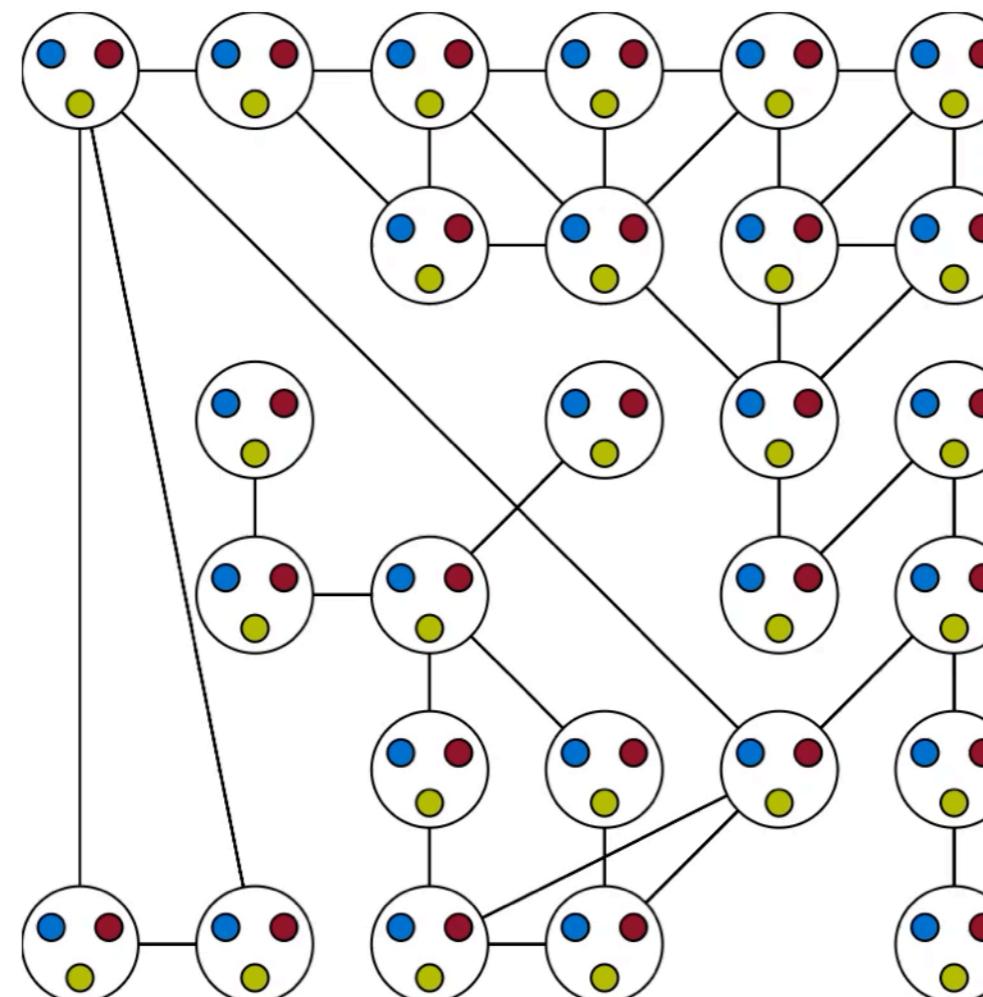
700

对值的排序选择

- 选择最小制约的值 (Least Constraining Value -- LCV)
 - 选择对剩下的变量在选值的时候约束最小的值
 - 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



最小约束值



Graph

Algorithm

Ordering

- None
- MRV
- MRV with LCV

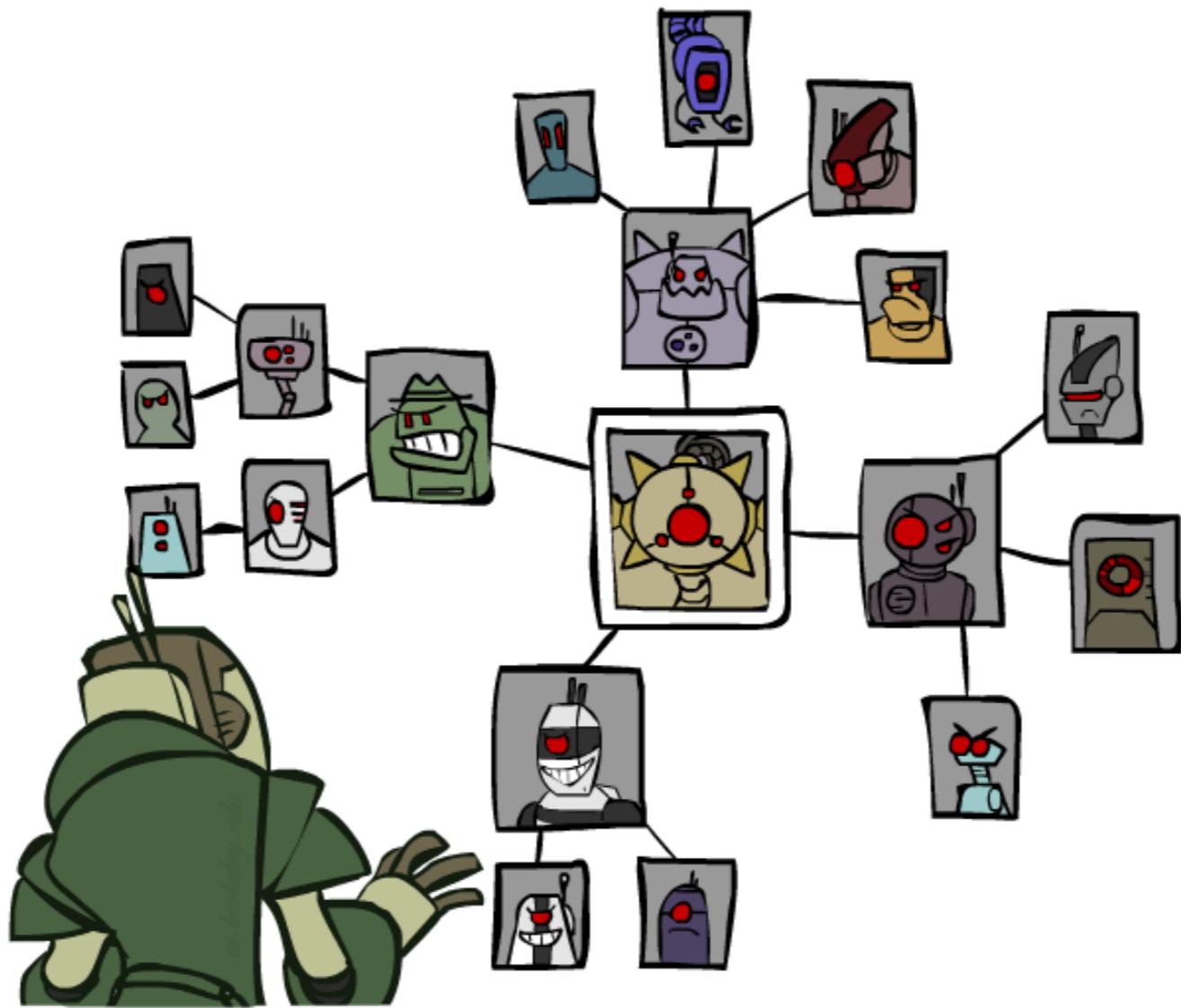
Filtering

- None
- Forward Checking
- Arc Consistency

Speed

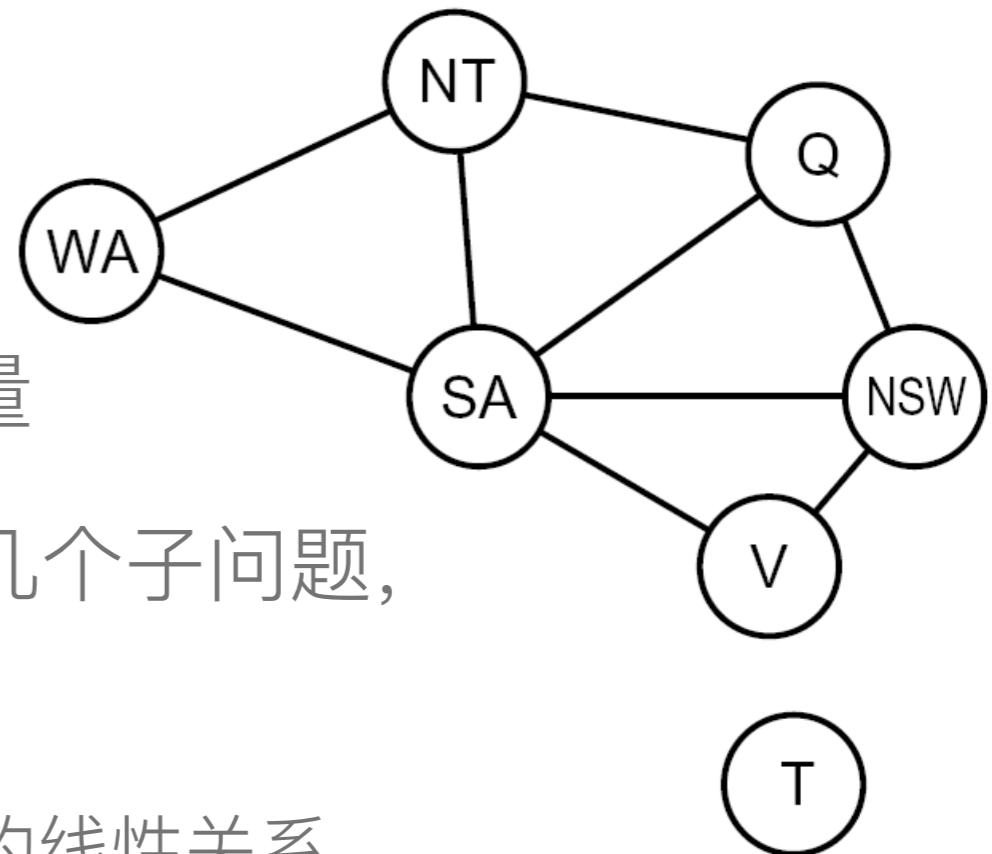
| | |
|---------|---------|
| Speedup | Frame |
| 1 | x Delay |
| 700 | |

结构

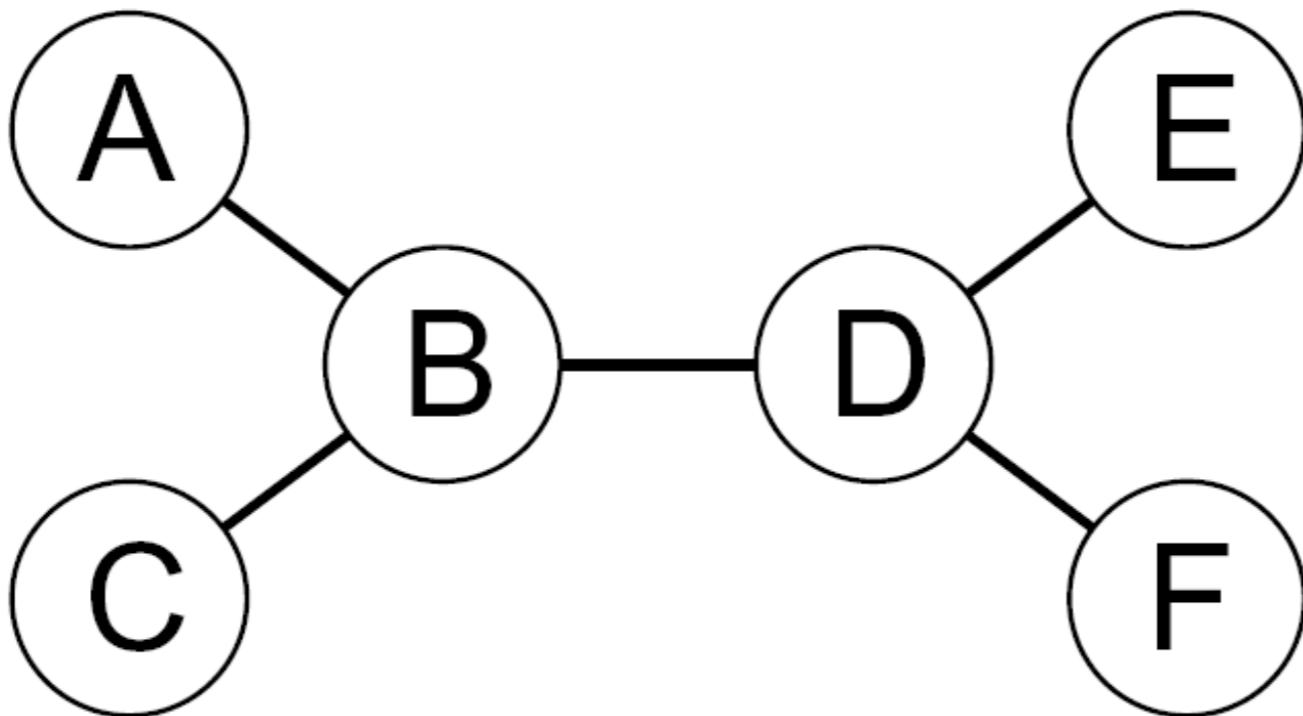


问题结构

- 极端情况: 独立的子问题
 - 例如: Tasmania 和大陆不相连
- 独立子问题可识别为约束图的连通分量
- 假设一个 n 变量的约束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: $2^{80} = 40$ 亿年
 - 4 个子问题: $4 \times 2^{20} = 0.4$ 秒



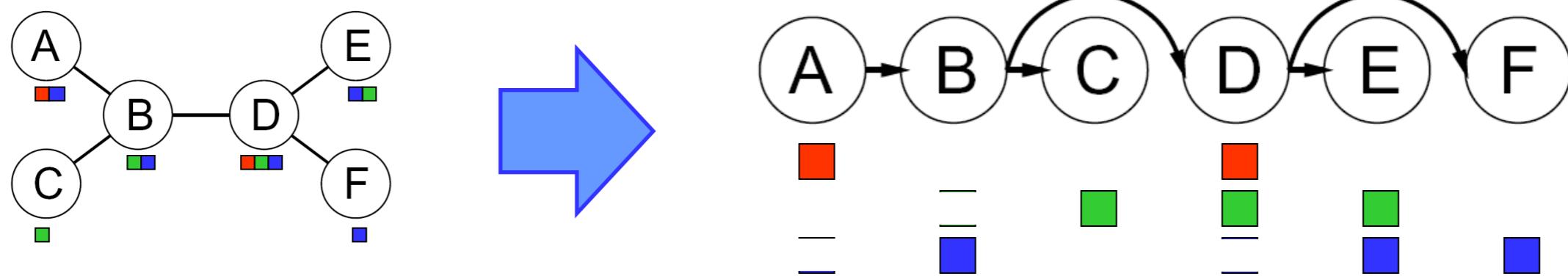
树结构的CSP



- 定理:如果约束图没有循环，则CSP求解时间为 $O(n \cdot d^2)$
 - 与一般csp相比，最坏情况时间为 $O(d^n)$

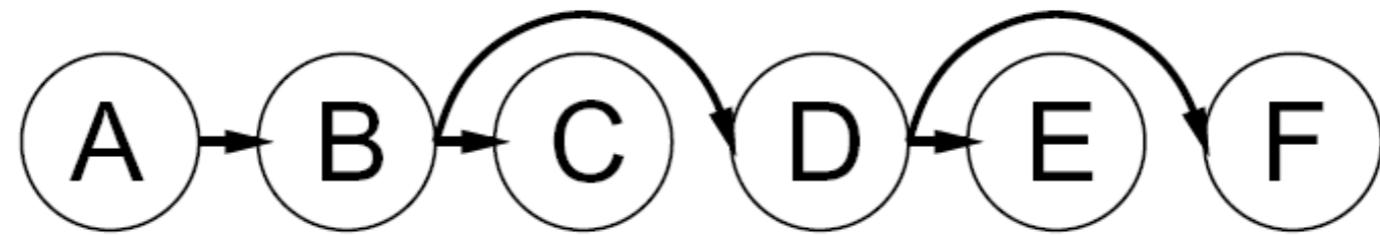
树结构的CSP

- 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 $i = n : 2$, 执行: 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 $\text{Parent}(X_i)$ 相一致的值
- 运行时间: $O(n \cdot d^2)$



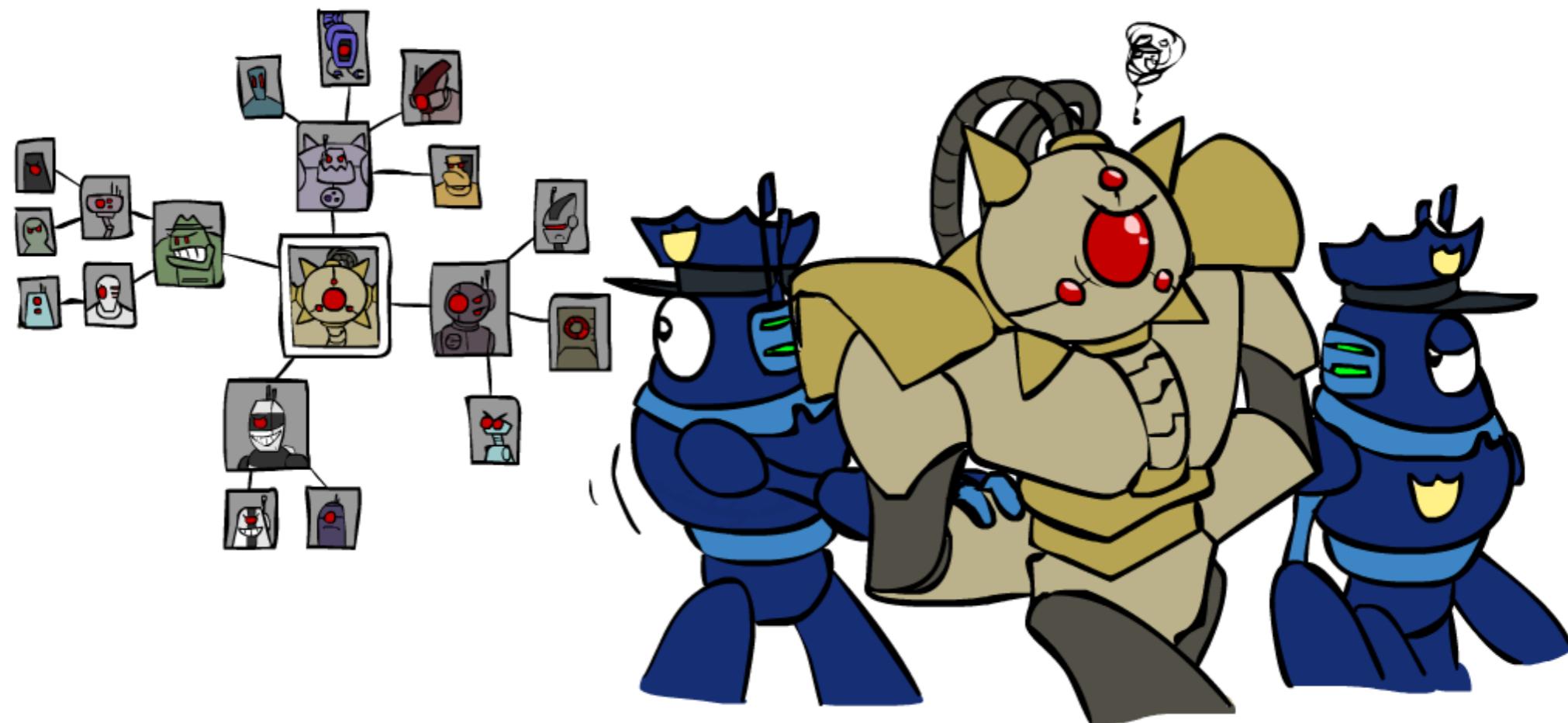
树结构的CSP

- 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)

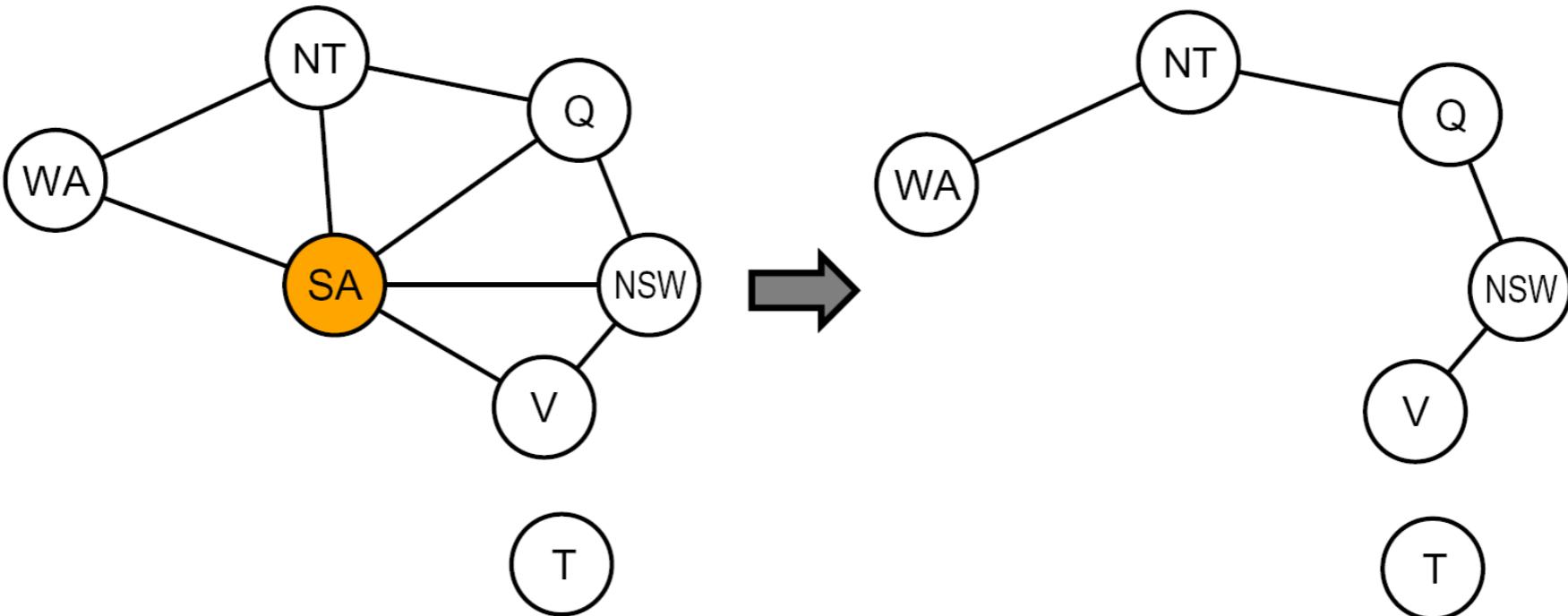


- 声明 2: 如果从根到叶的弧都是一致性的, 那么从前向后的赋值过程将不会有回溯
 - 证明: 归纳法。
 - 为什么这个算法不适用于约束图中有环的情况?

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
- 割集条件制约: 对割集变量赋值 (所有可能的组合), 使得剩下的约束图变成一个树
- 割集大小是 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 $\rightarrow 0.029$ 秒

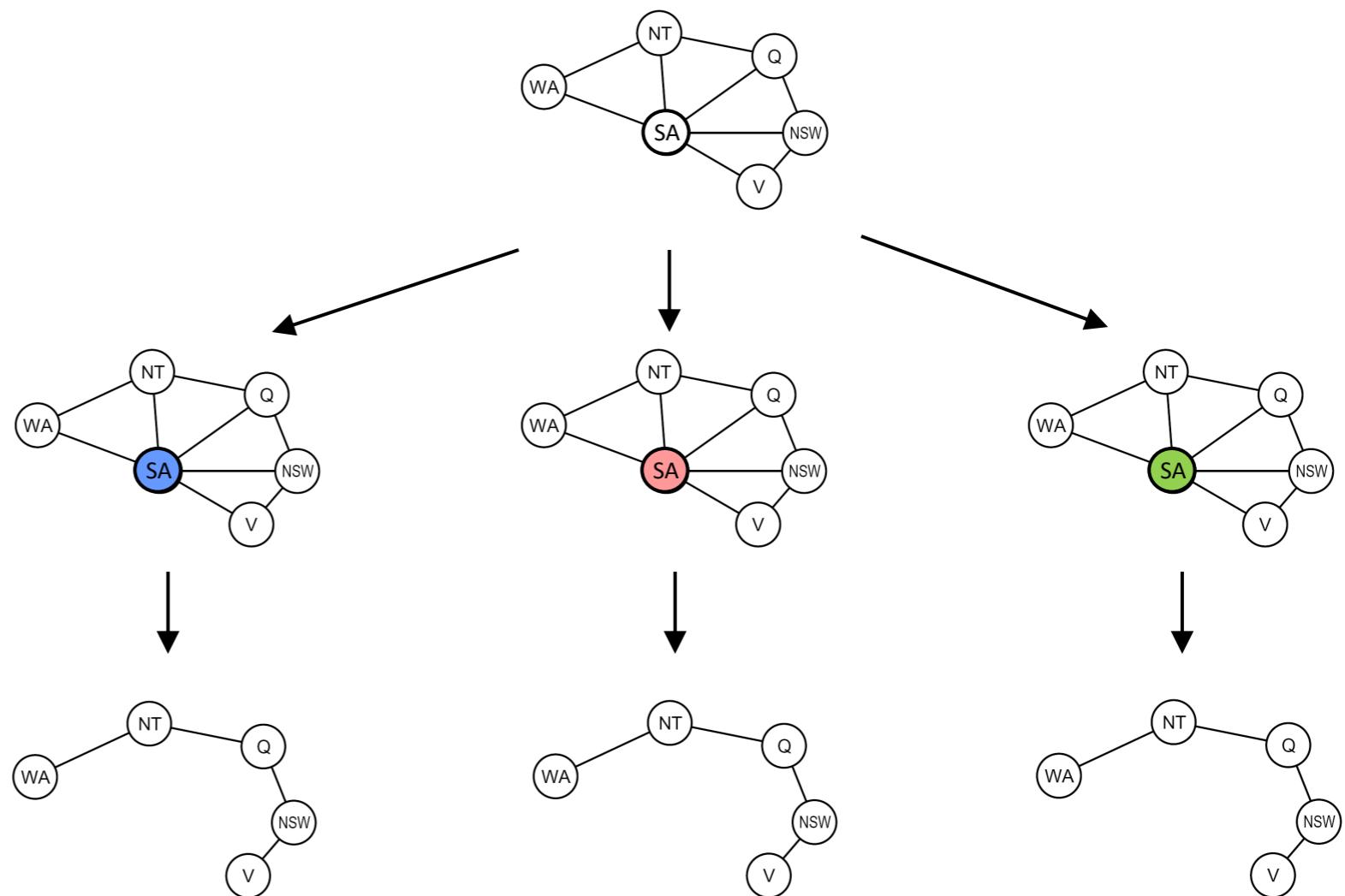
割集条件化制约

选择一个割集

对割集变量赋值(所有可能组合)

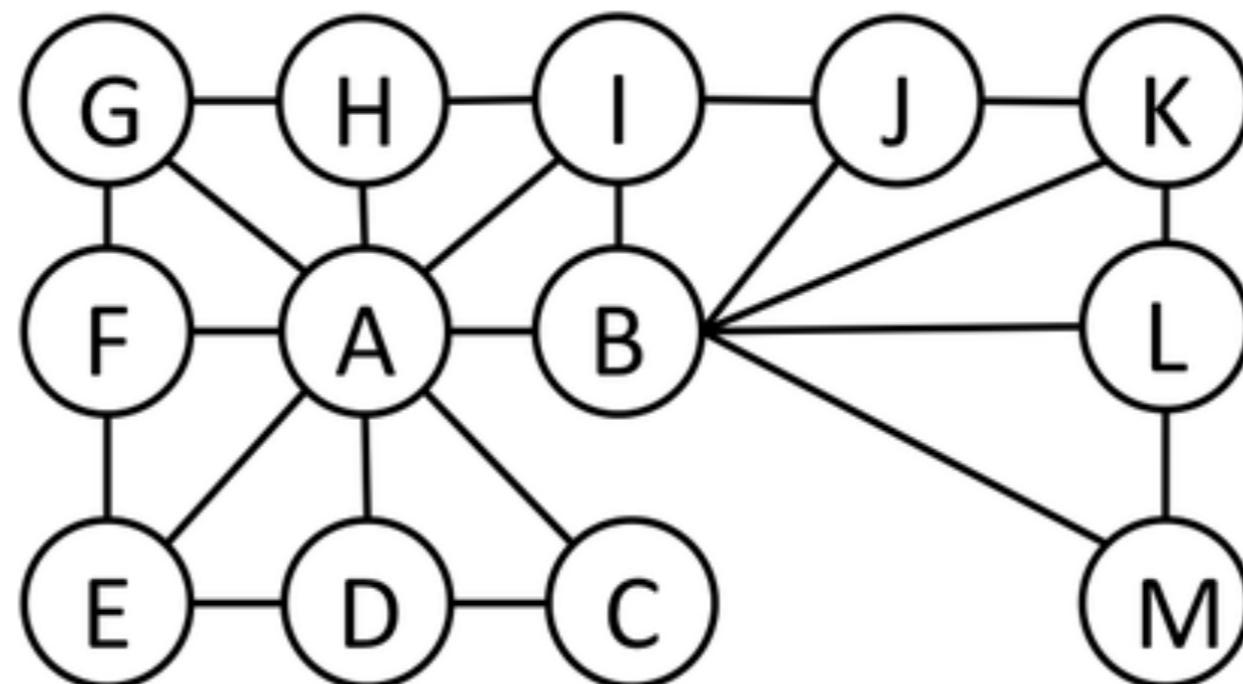
计算剩余的CSP相对于每一组割集赋值

求解剩余的CSPs(树结构的)



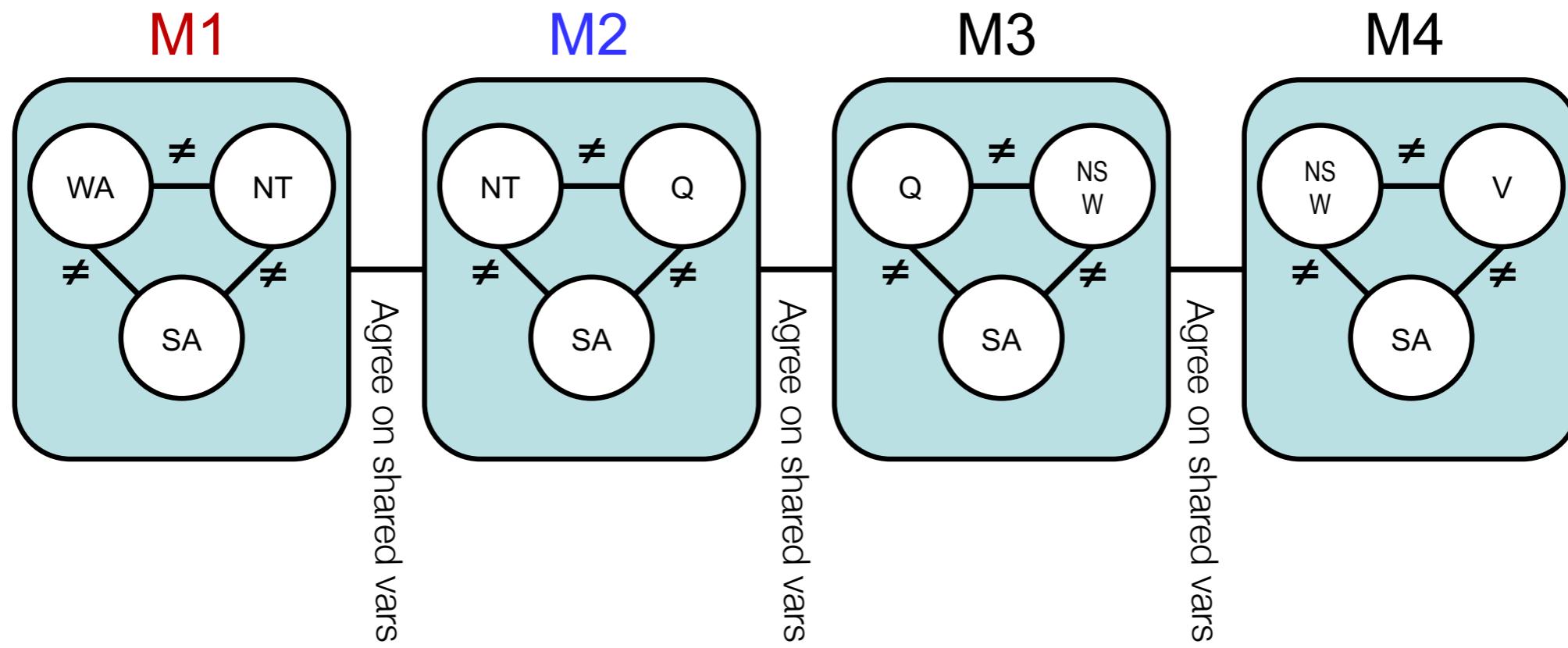
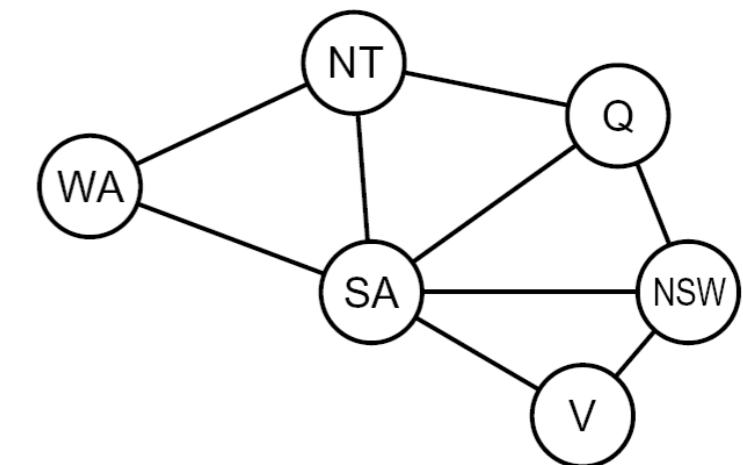
割集示例

- 求下列图的最小割集。

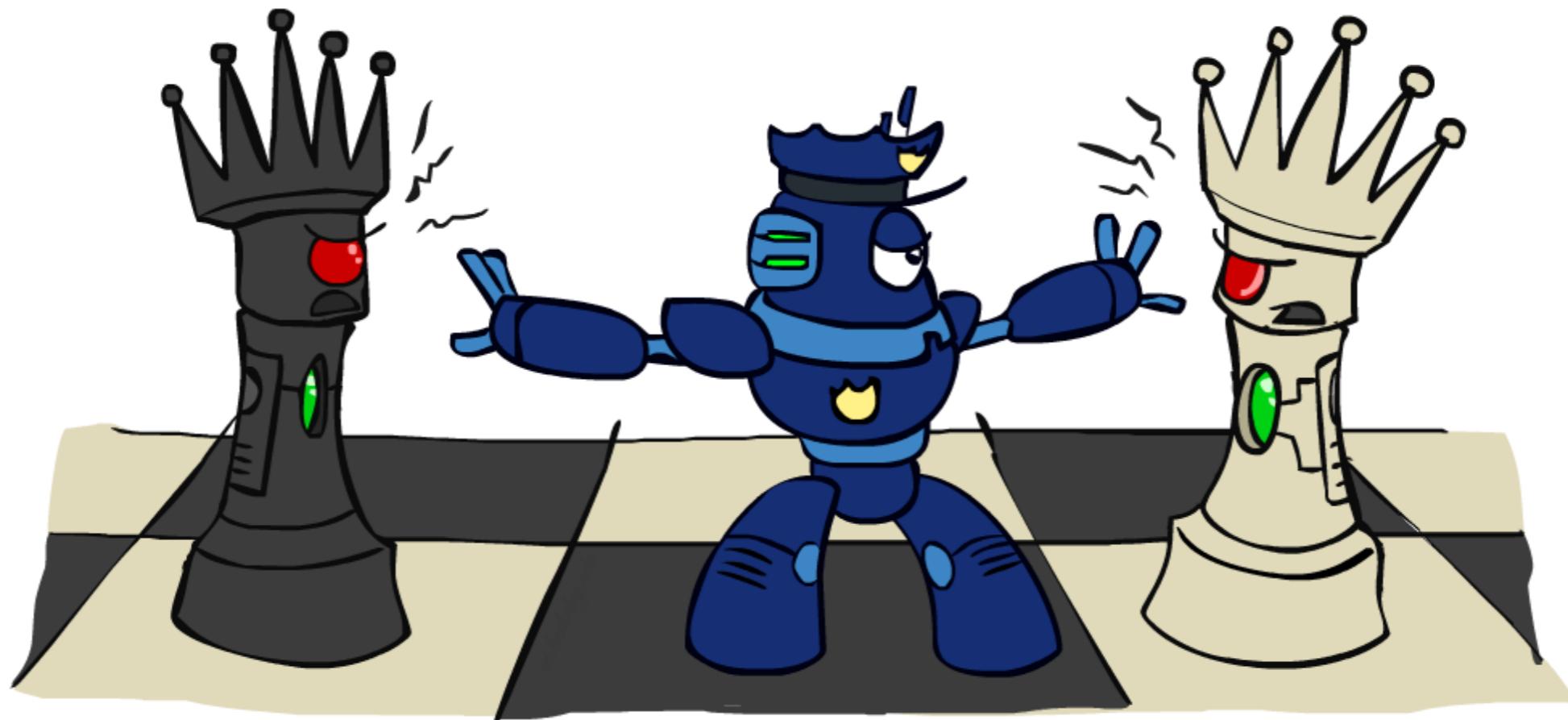


树分解

- 想法: 创建一个“巨变量”的树形结构图
- 每个“巨变量”对原始CSP的一部分进行编码
- 子问题重叠以确保一致的解决方案

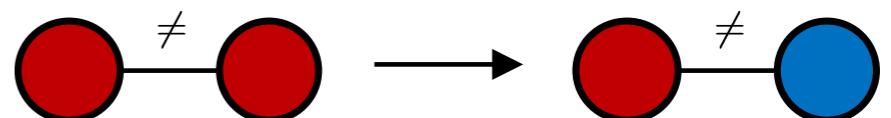


迭代改进

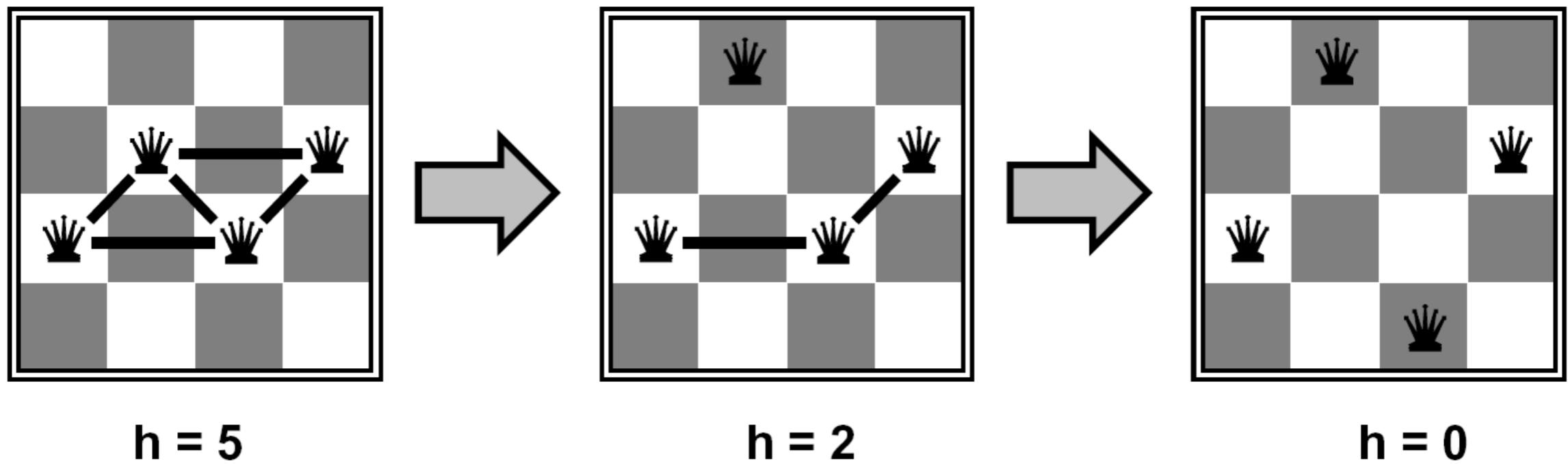


CSP的迭代改进算法

- 随机赋值一组不满足约束条件的变量
- 迭代更新变量值，直到满足约束条件
 - Algorithm: While not solved,
 - 变量选择:随机选择任何冲突的变量
 - 值选择:最小冲突(Min-Conflicts)的启发式方法
 - 选择违背最少约束的值
 - 即 $h(n)$ =违反约束的总数

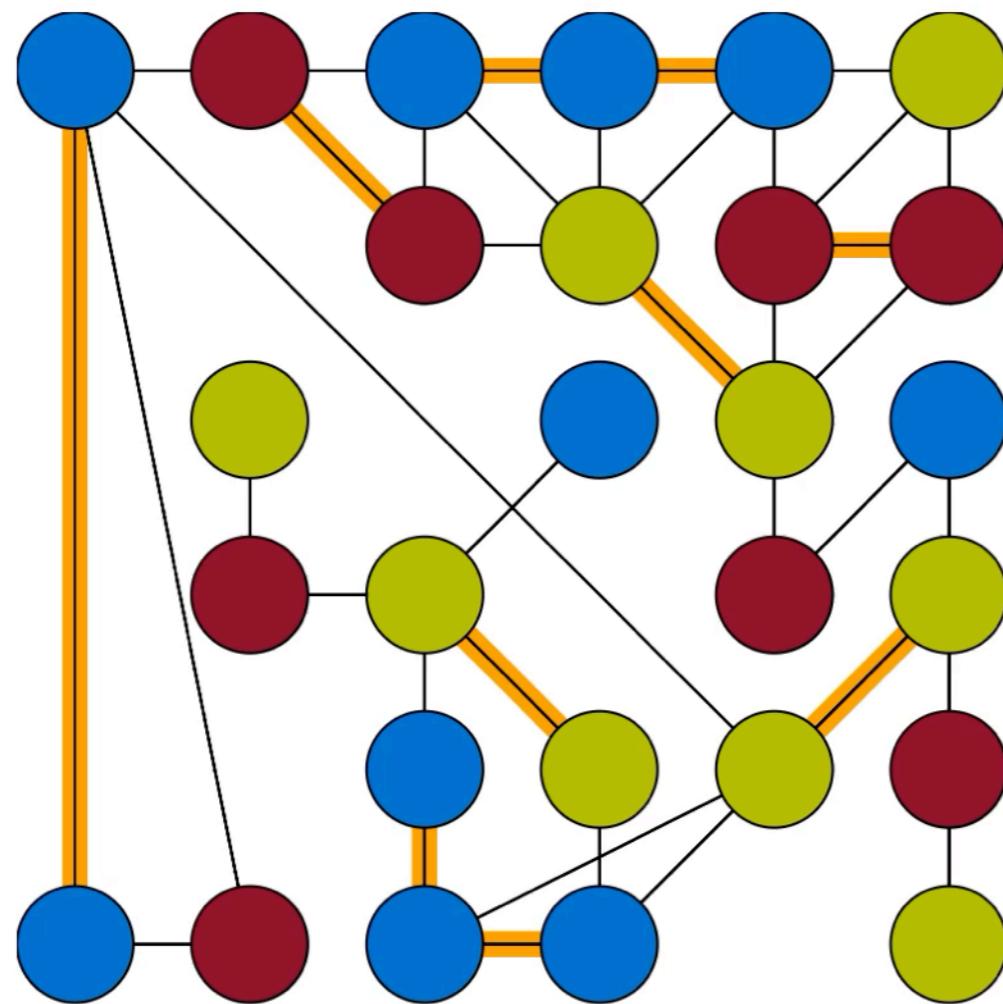


迭代求解4皇后



- 状态： 4x4的方格中放置了4个皇后($4^4=256$ 个状态)
- 操作： (在列上) 移动皇后
- 目标测试： 任意两个皇后不在同一行、列、斜线上

演示：迭代改进



Reset Prev Pause Next Play Faster

Graph
Complex

Algorithm
Iterative Improvement

Ordering
None
MRV
MRV with LCV

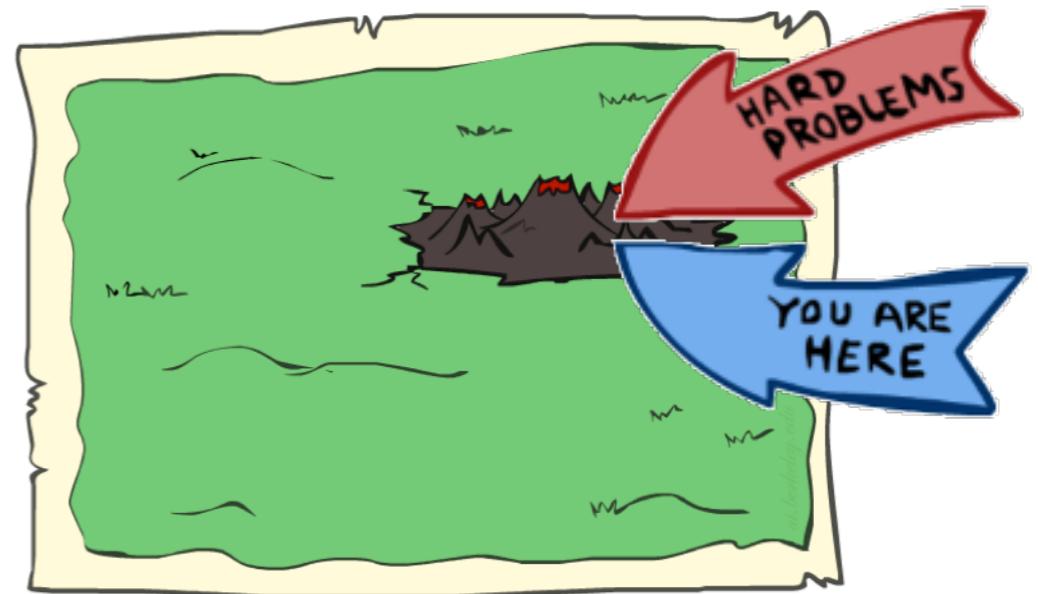
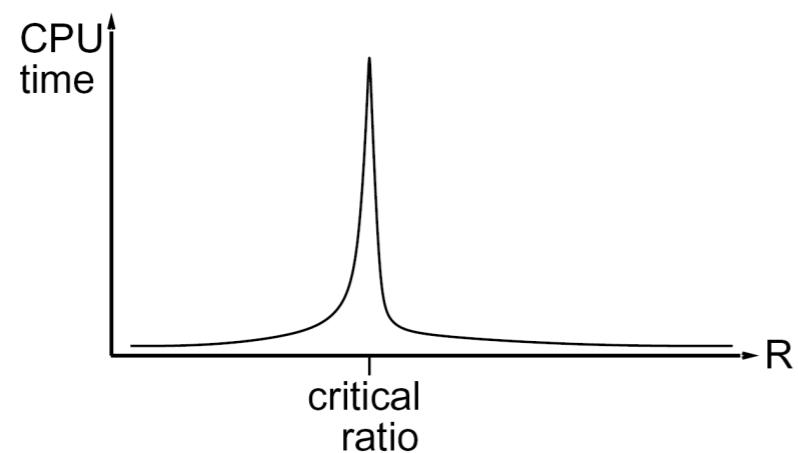
Filtering
None
Forward Checking
Arc Consistency

Speed
Speedup Frame
1 x Delay
700

Min-Conflicts的性能

- 给定随机初始状态，可以在几乎恒定的时间内以高概率解出任意n的n次皇后！(例如， $n = 10,000,000$)
- 对于任何随机产生的CSP，除了在一个狭窄的比率范围内，几乎都是如此

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

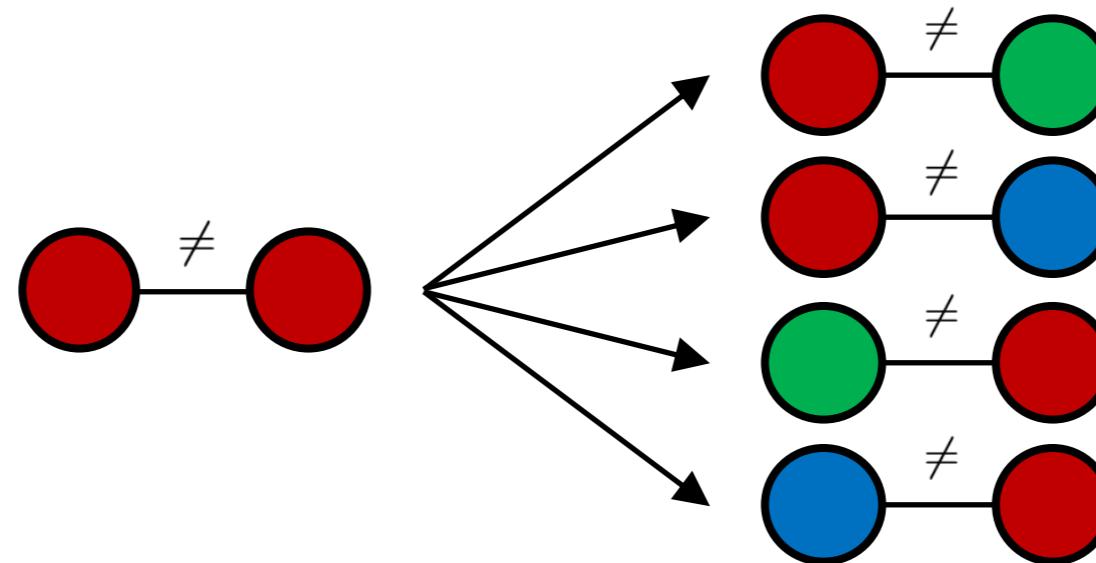


局部搜索



局部搜索 (Local Search)

- 树搜索：将未经探索的备选方案保留在边缘（确保完备性）
- 局部搜索：改进一个选项，直到你无法改进为止（没有边缘！）



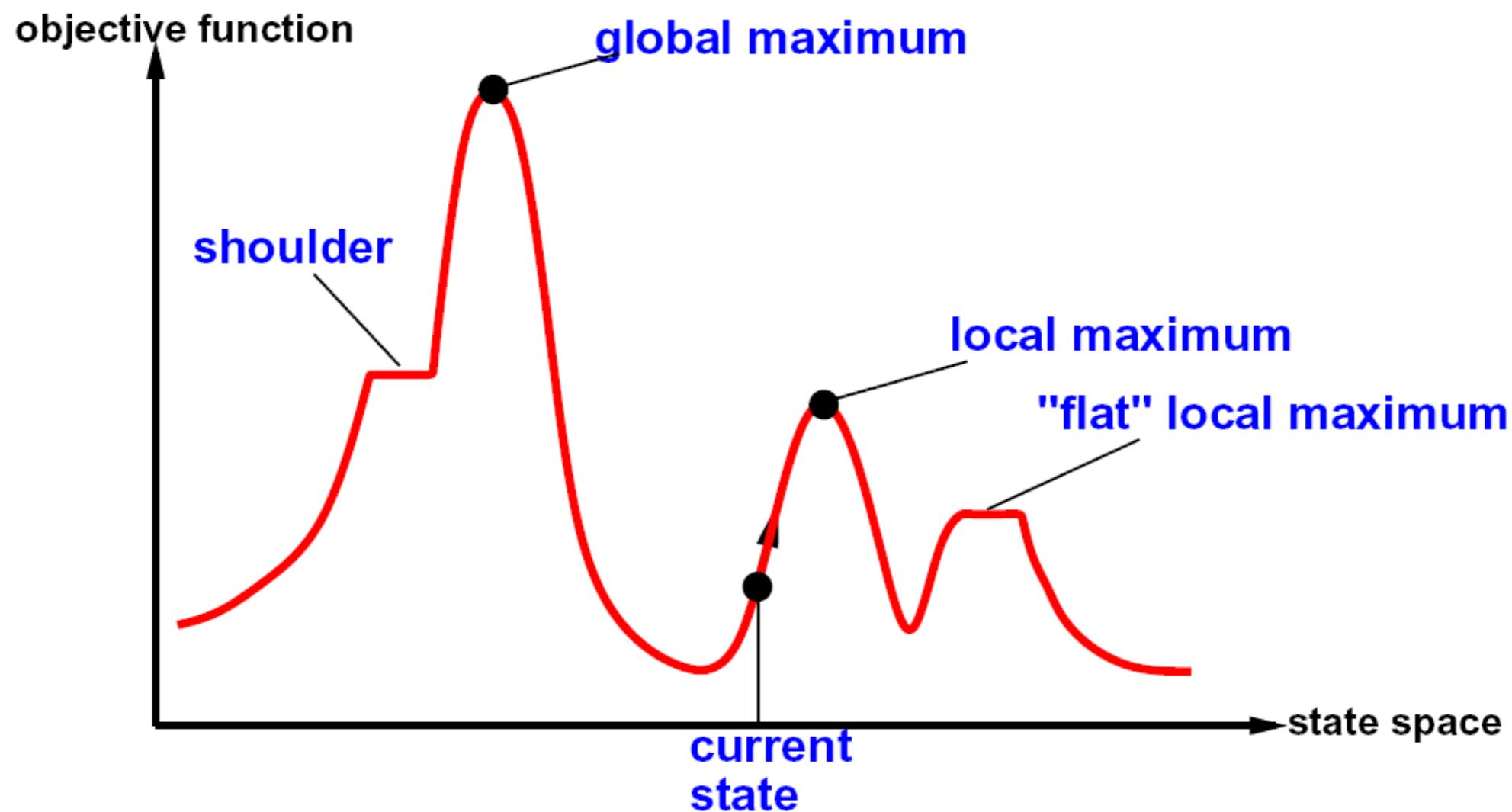
- 局部搜索通常速度更快，内存效率更高（但不完备也不最优）

爬山算法 (Hill Climbing)

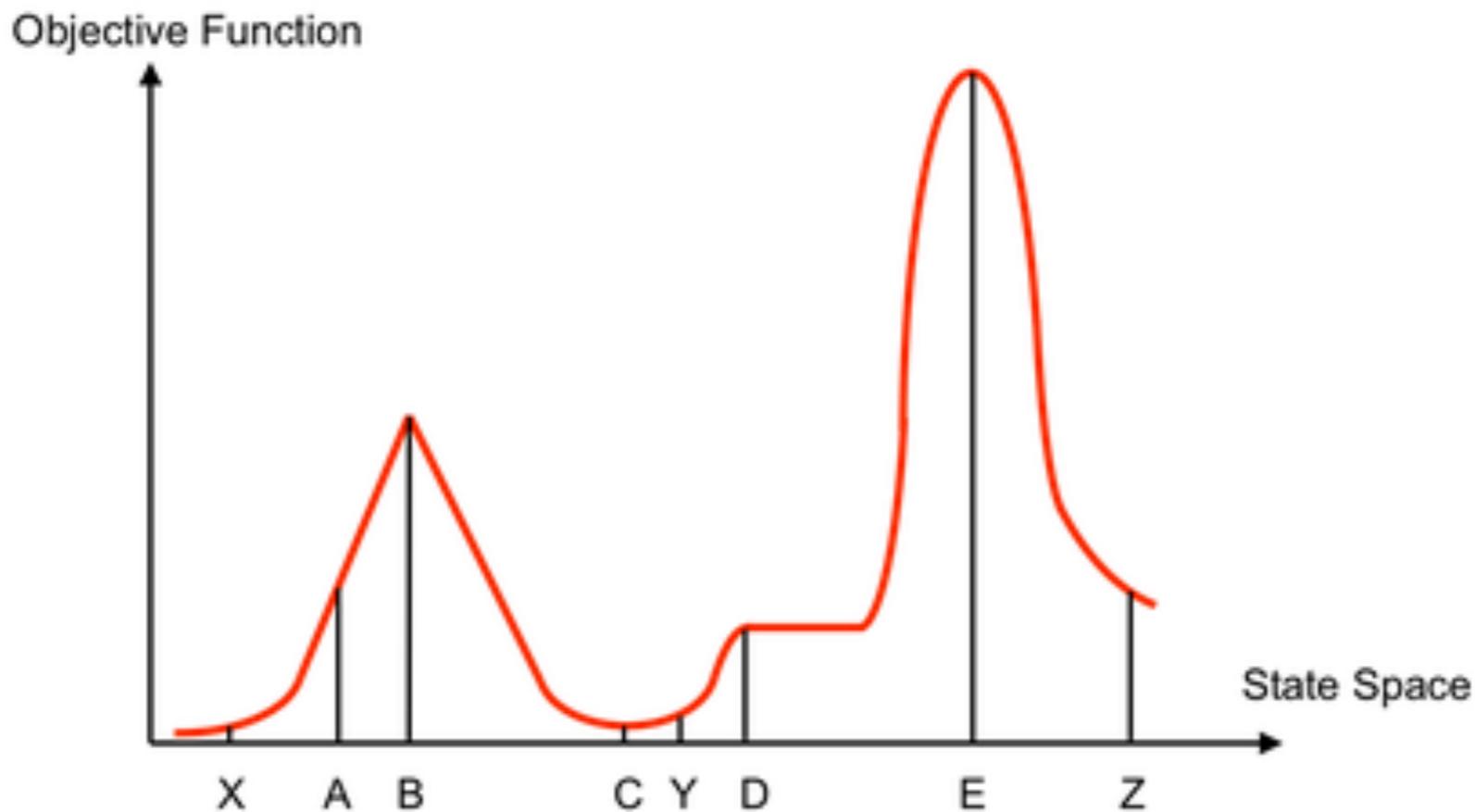
- 算法：
 - 从随机的位置开始
 - 重复：向上爬
 - 直到没有“向上”的路为止。
- 会陷入局部最优解，而不一定能搜索到全局最优解



爬山算法



爬山算法



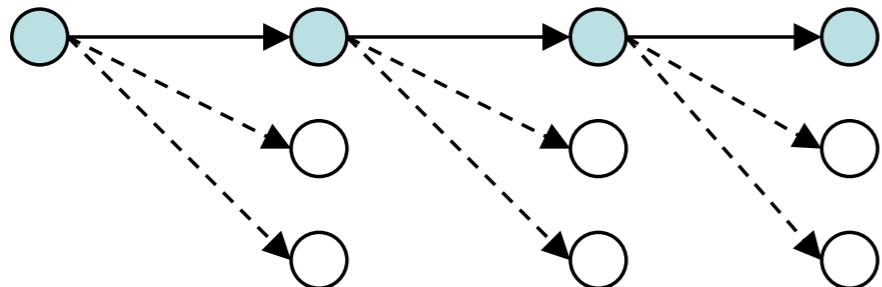
从X开始爬山，会在哪里结束？

从Y开始爬山，会在哪里结束？

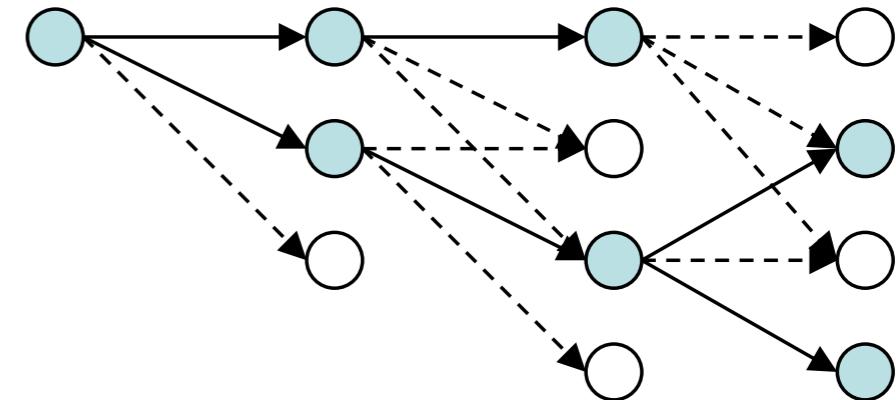
从Z开始爬山，会在哪里结束？

光束搜索 (Beam Search)

- 就像贪心的爬山搜索，但始终保持K状态：



Greedy Search

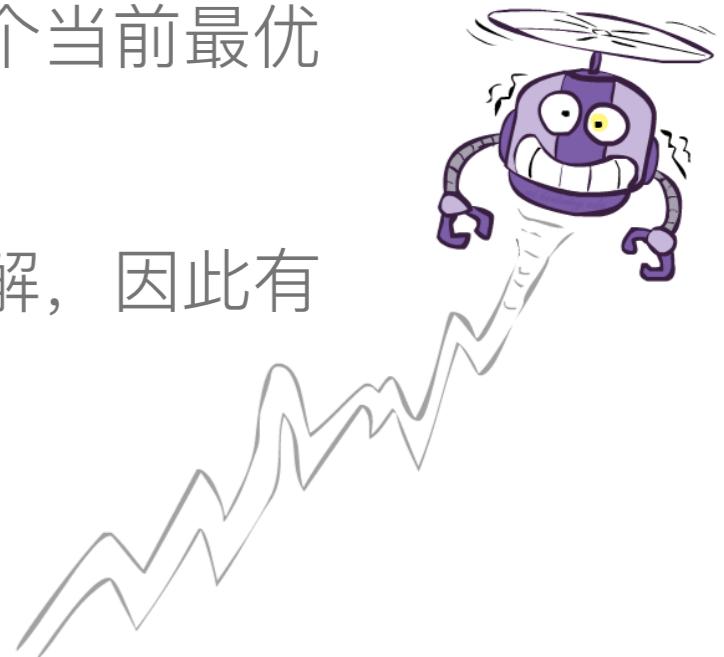


Beam Search

- 变量：光束大小，多样性约束
- 在许多实际问题中（例如机器翻译）是最常用的算法
- 是否完备？最优？

模拟退火算法 (Simulated Annealing)

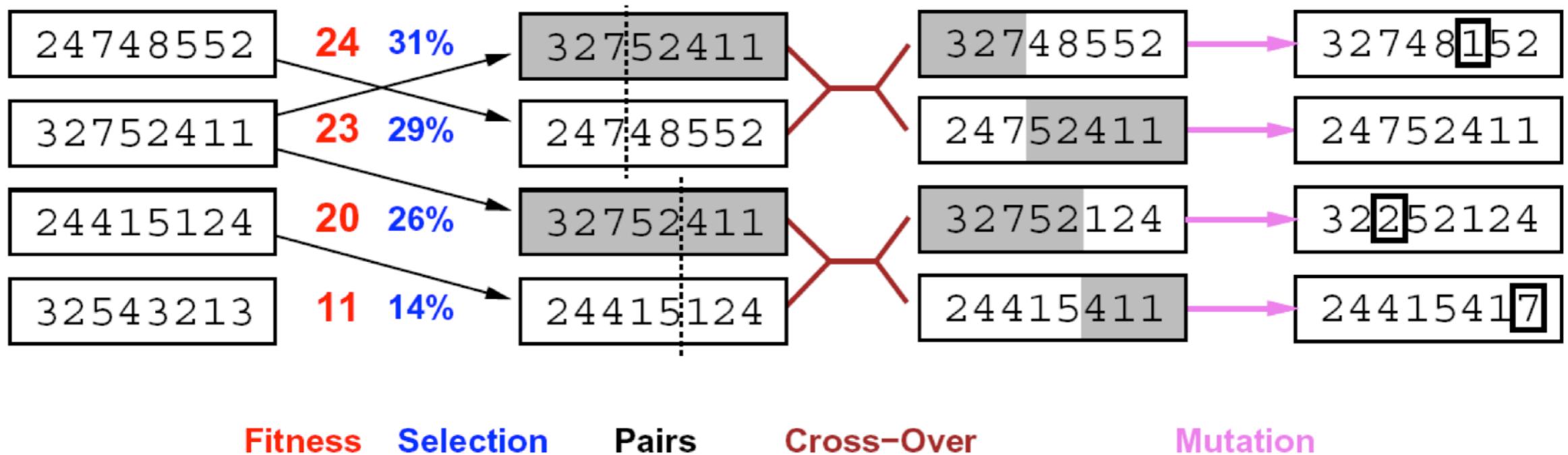
- 爬山法是完完全全的贪心法，每次都鼠目寸光的选择一个当前最优解，因此只能搜索到局部的最优值。
- 模拟退火算法以一定的概率来接受一个比当前解要差的解，因此有可能会跳出这个局部的最优解，达到全局的最优解。
- 算法：
 - 若 $J(Y(i+1)) \geq J(Y(i))$ (即移动后得到更优解)，则总是接受该移动
 - 若 $J(Y(i+1)) < J(Y(i))$ (即移动后的解比当前解要差)，则以一定的概率接受移动，而且这个概率随着时间推移逐渐降低 (逐渐降低才能趋向稳定)
 - 这里的“一定的概率”的计算参考了金属冶炼的退火过程，这也是模拟退火算法名称的由来。



模拟退火算法

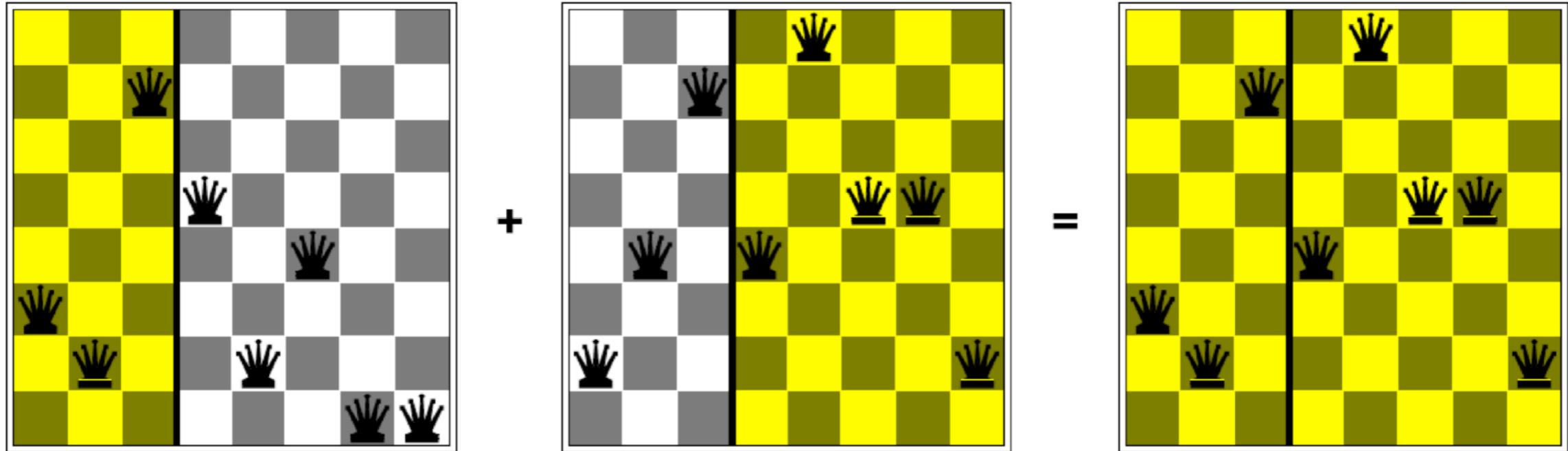
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to “temperature”
  local variables: current, a node
                    next, a node
                    T, a “temperature” controlling prob. of downward steps
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next] – VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

遗传算法 (Genetic Algorithms)



- 遗传算法模拟了自然选择的过程
 - 基于适应度函数，在每个步骤（选择）保持最佳N个假设
 - 使用互换和变异提供多样性

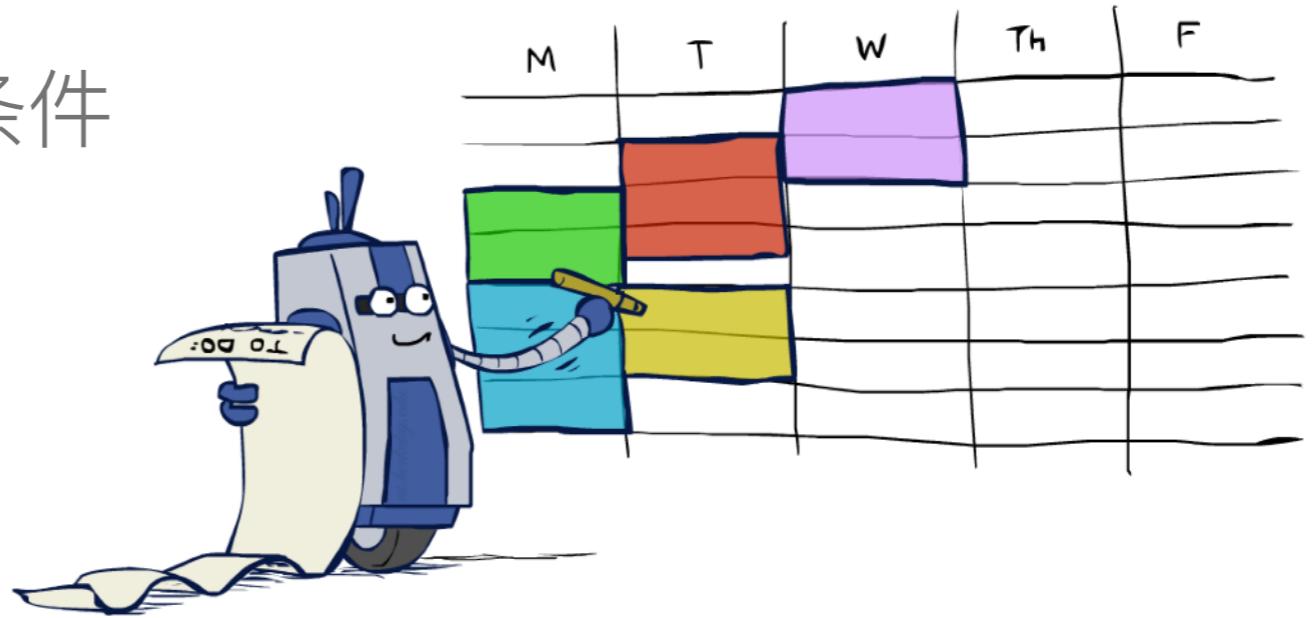
遗传算法



- 为什么互换在这里有意义？
- 突变是什么？
- 什么是好的适应度函数？

总结: 约束满足问题

- 约束满足问题是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
 - 通用的算法和启发信息
- 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



作业

- 假设计算机课程被安排在每周一，三，五。总共有 5 门课程，3 名教师，每名教师在一个时间只能教一门课。
课程包括：
 - 课程 1 – 计算机编程简介，时间 8:00 – 9:00AM
 - 课程 2 – 人工智能导论，时间 8:30 - 9:30AM
 - 课程 3 – 软件工程，时间 9:00 – 10:00 AM
 - 课程 4 – 计算机视觉，时间 9:00 – 10:00 AM
 - 课程 5 – 机器学习，时间 10:30 – 11:30AM
- 教师包括：
 - 教师 A，能够教课程 1, 2, 5
 - 教师 B，能够教课程 3, 4, 5
 - 教师 C，能够教课程 1, 3, 4
- 关于如何安排以上课程的教师，请回答以下问题：
 1. 把以上问题描述为一个约束满足问题(每门课程是一个变量)，指明变量的值域和约束。约束也可以是隐式表达形式。
 2. 画出相应的约束图？
 3. 如果这个约束图看上去近似树状。请找到一个相应的割集，并解释为什么将这个问题转成树状约束图后能有所帮助？
 4. 编程并求解这个问题。