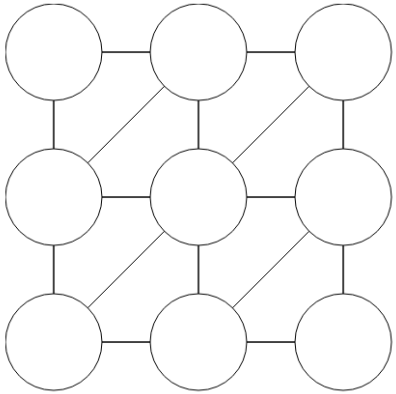
小练习

填充 红, 绿, 或 蓝
相邻颜色必须不同

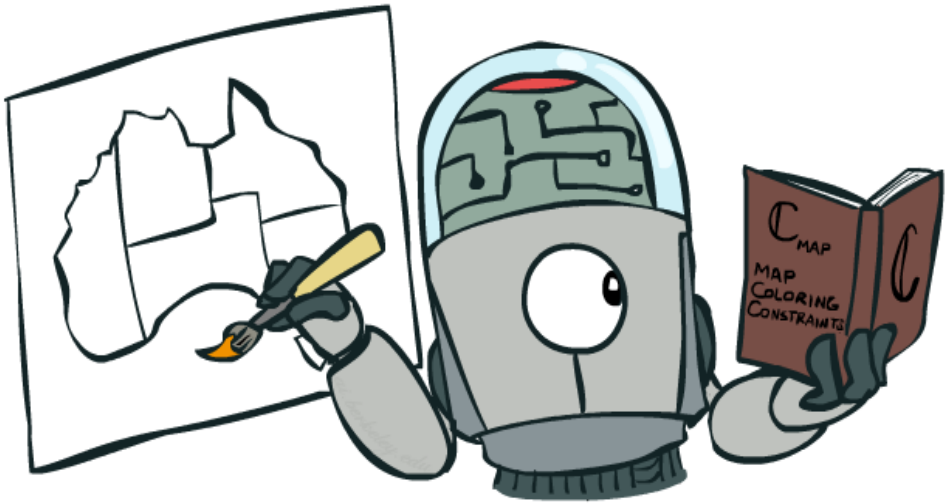


Sudoku

1			
	2	1	
		3	
			4

1) 你的大脑是如何来解决这些问题的?
2) 你如何用搜索方法来解决这些问题 (BFS, DFS, 等)?

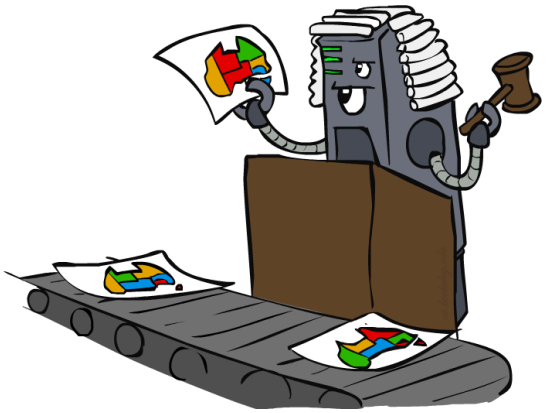
人工智能导论：约束满足问题



标准的（一般的）搜索问题

标准的搜索问题:

- 状态是一个 ***黑盒***: 任意的数据结构（不知道是什么结构）
- 目标检测 是一个在状态上的黑盒检测
- 行动是黑盒数据结构
- 转换模型是一个黑盒函数

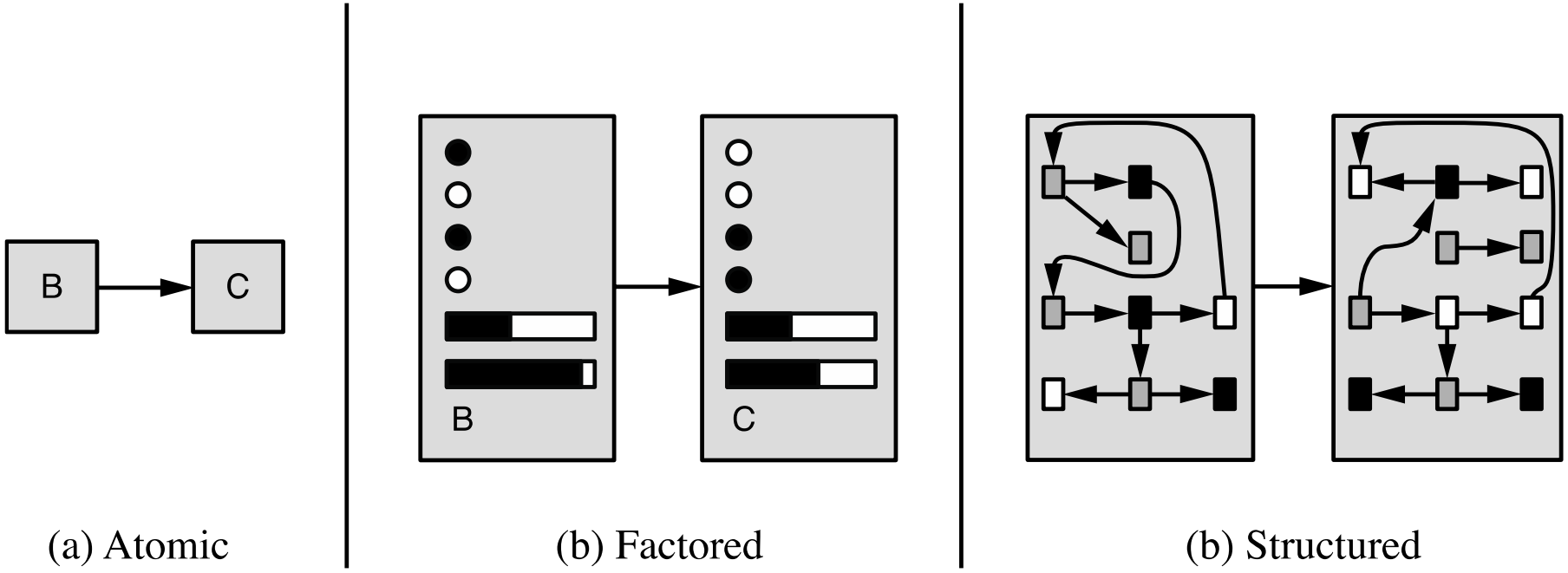


这样的结果，对于每一新的问题，不得不：

- 编写新的程序
- 重新设计新的启发信息，根据特定问题
- 缺乏通用目的的启发式信息

解决方案: 对于状态，行动，目标使用形式化的表达。

表达法



(a) Atomic　　(b) Factored　　(b) Structured

搜索,
博弈问题

CSPs, 规划,
命题逻辑,
贝叶斯网络, 神经网络

一阶逻辑,
数据库,
概率程序

两种主要的问题求解

规划: 解是一个行动序列 (或行动策略)

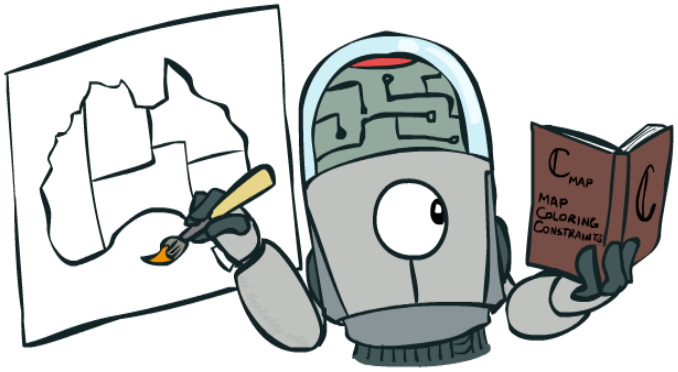
- 重要的是到目标状态的路径
- 路径有不同的成本和探索深度
- 和行动的顺序相关

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）

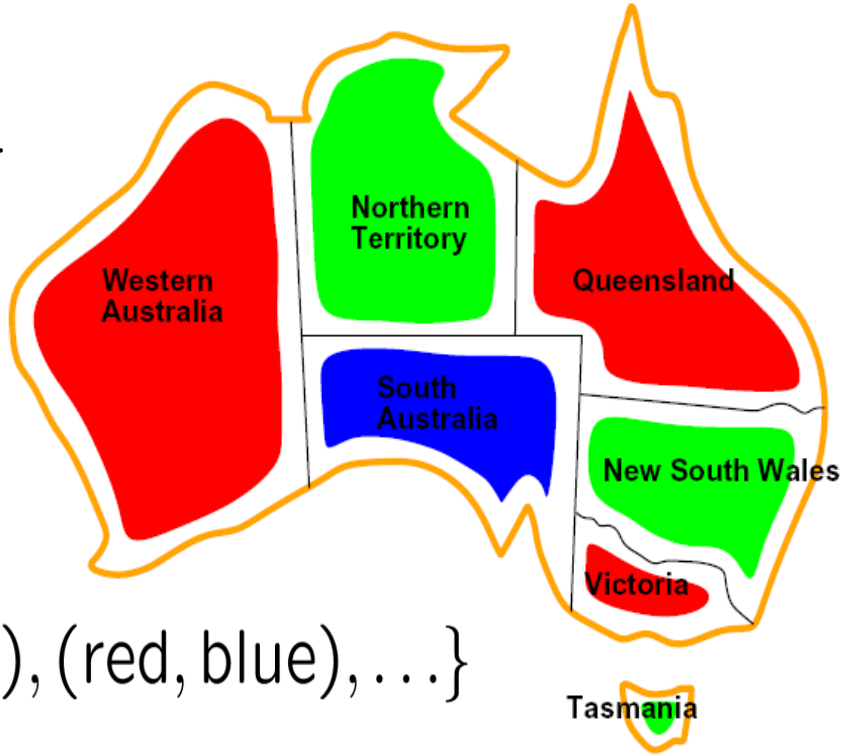
- 状态由若干个 ***变量*** X_i 组成，每个变量有一个 ***取值域*** D_i
- 目标测试是一套约束规则，规定了允许的变量集合的取值组合
- 到目标的路径（变量赋值的顺序）不重要
- 其结果：
  - CSP算法比标准的搜索问题要快
  - 启发信息可用于所有CSP问题
  - 解决新的问题不需要写新的代码



举例: 地图着色

变量: WA, NT, Q, NSW, V, SA, T

值域: $D = \{red, green, blue\}$



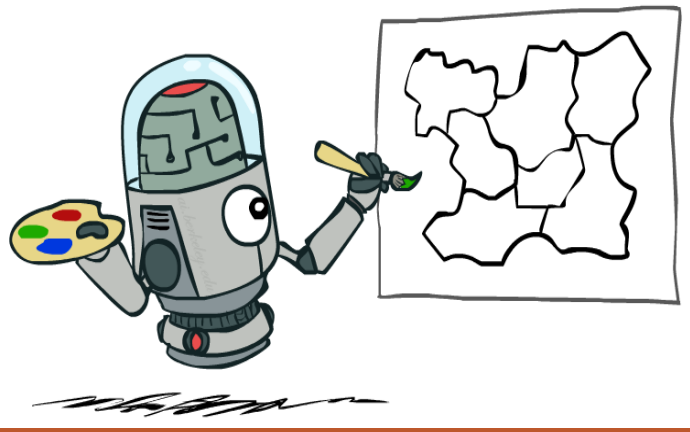
约束: 临近区域必须有不同的颜色

隐式: $WA \neq NT$

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

解 是 一组对所有变量的配值，其满足了所有的约束，例如：

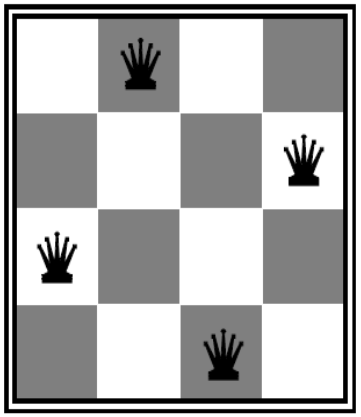
{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

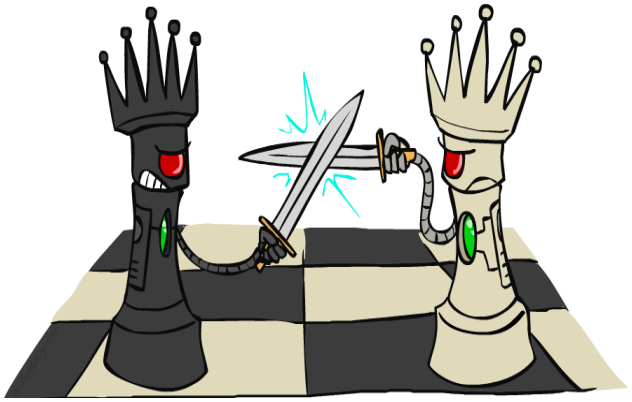


举例: N-皇后

问题描述方法 1:

- 变量: X_{ij}
- 值域: $\{0,1\}$
- 约束条件:





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

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

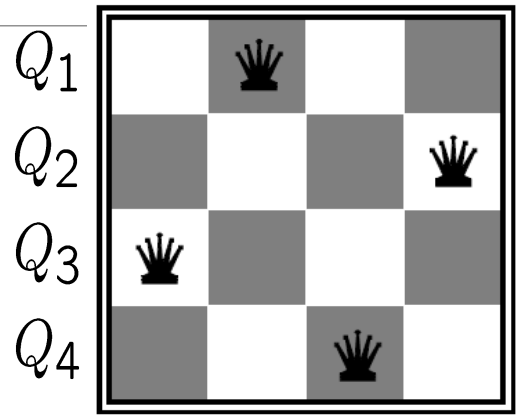
赋值选择空间：

2^{N^2}

举例: N-皇后

描述方法 2:

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



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

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

赋值探索空间

N^N

$\ldots$

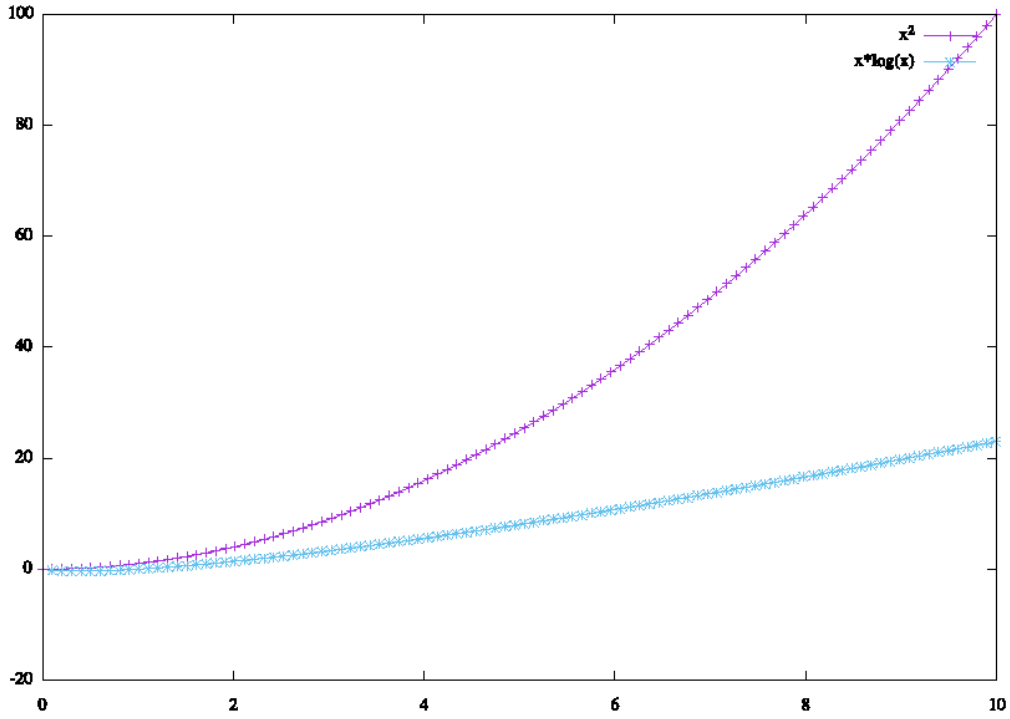
哪一个更大?

2^{N^2} vs N^N

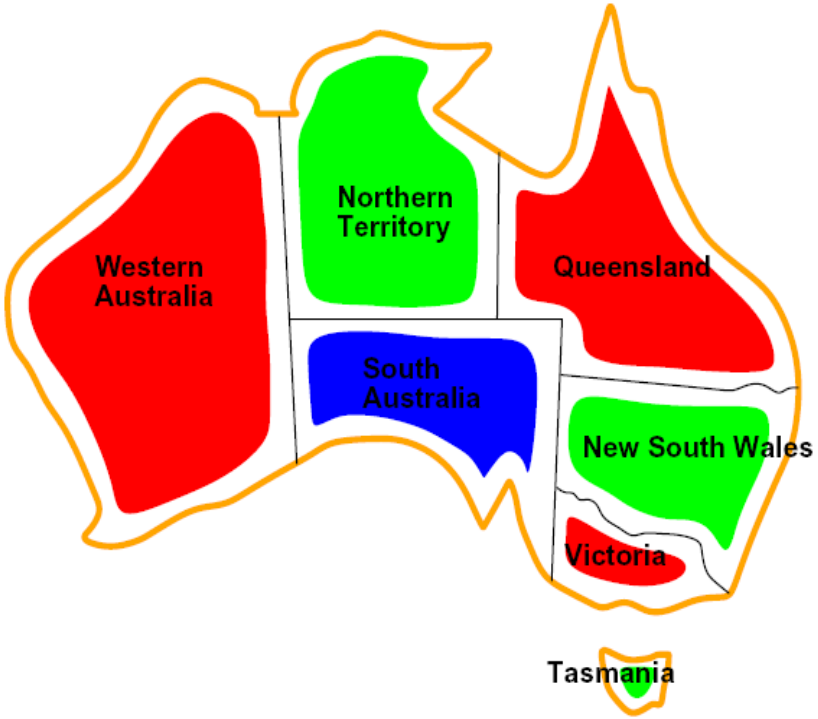
$\log_2 2^{N^2}$ vs $\log_2 N^N$

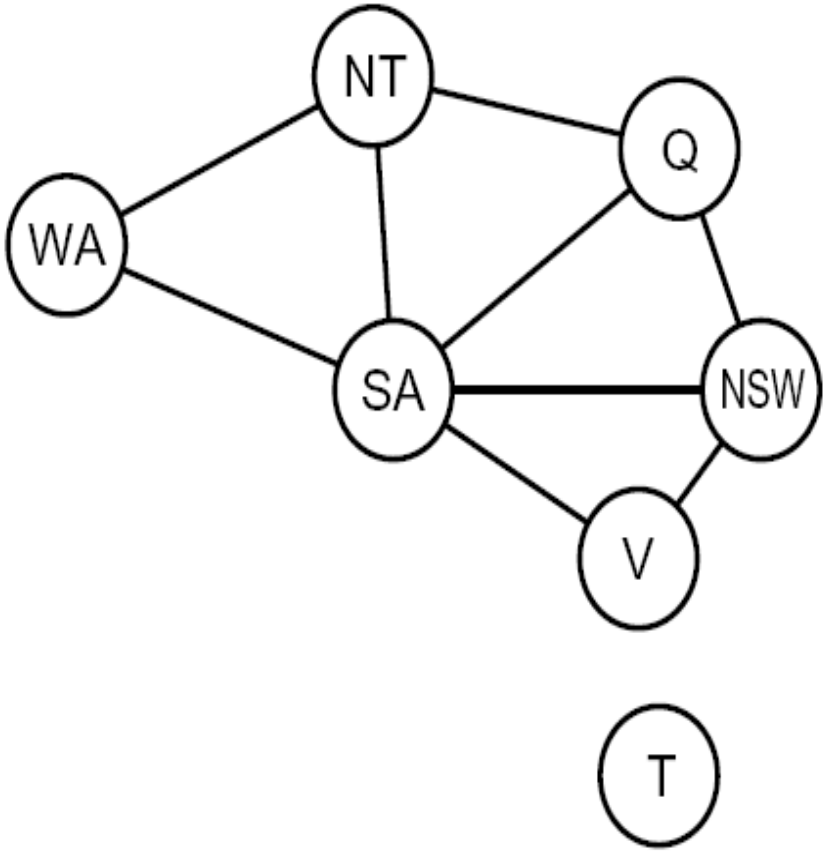
N^2 vs $N \log_2 N$

N=10: 10^{30} vs 10^{10}



约束图

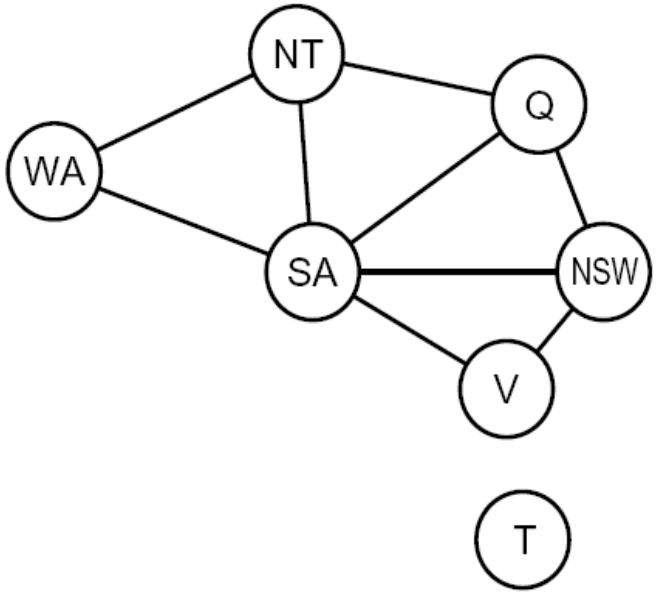




节点：变量
连接：存在约束关系

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:

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

```
  T W O
+ T W O
-------
F O U R
```

约束条件:

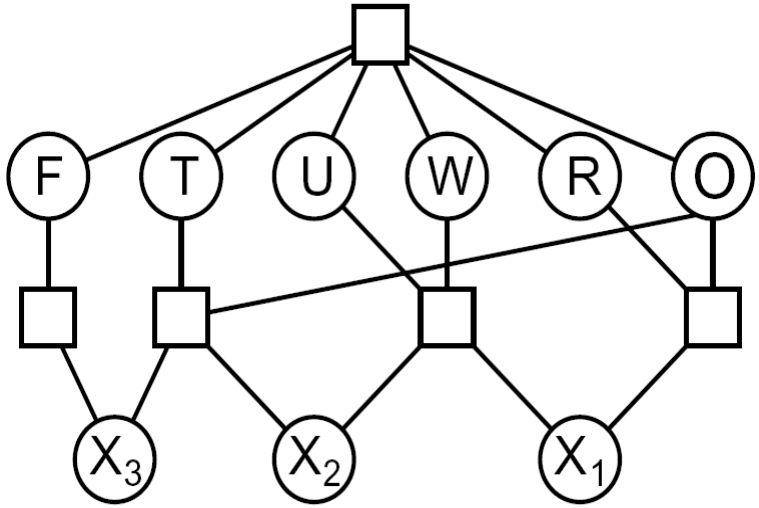
alldiff(*F,T,U,W,R,O*)

$O + O = R + 10 * X_1$

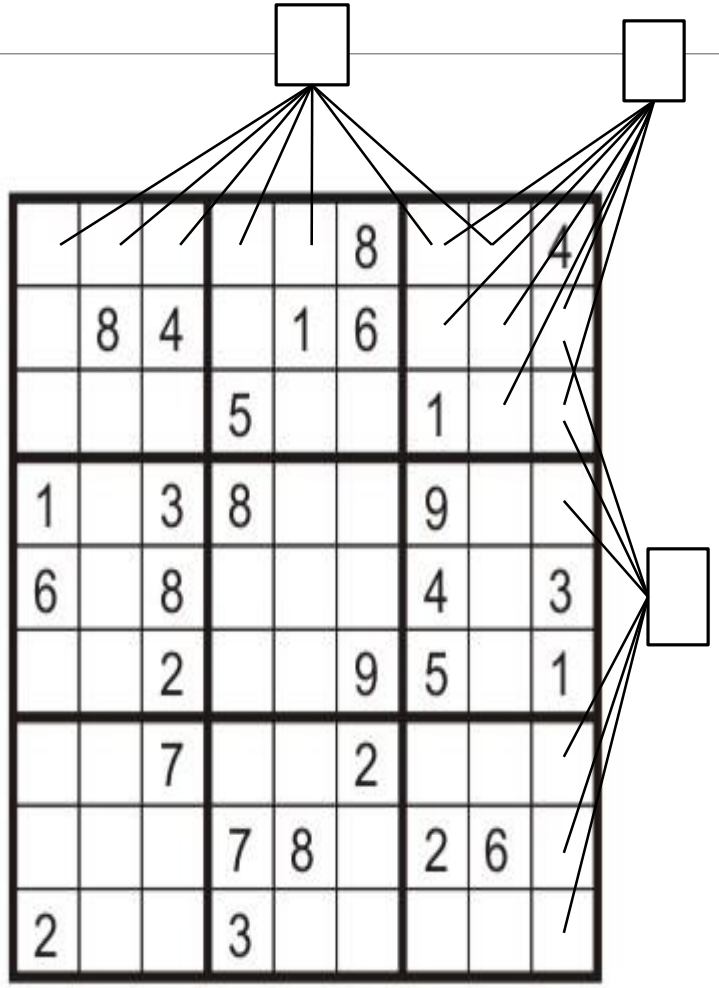
$X_1 + W + W = U + 10 * X_2$

$X_2 + T + T = O + 10 * X_3$

$X_3 = F$



举例: 数独(Sudoku)



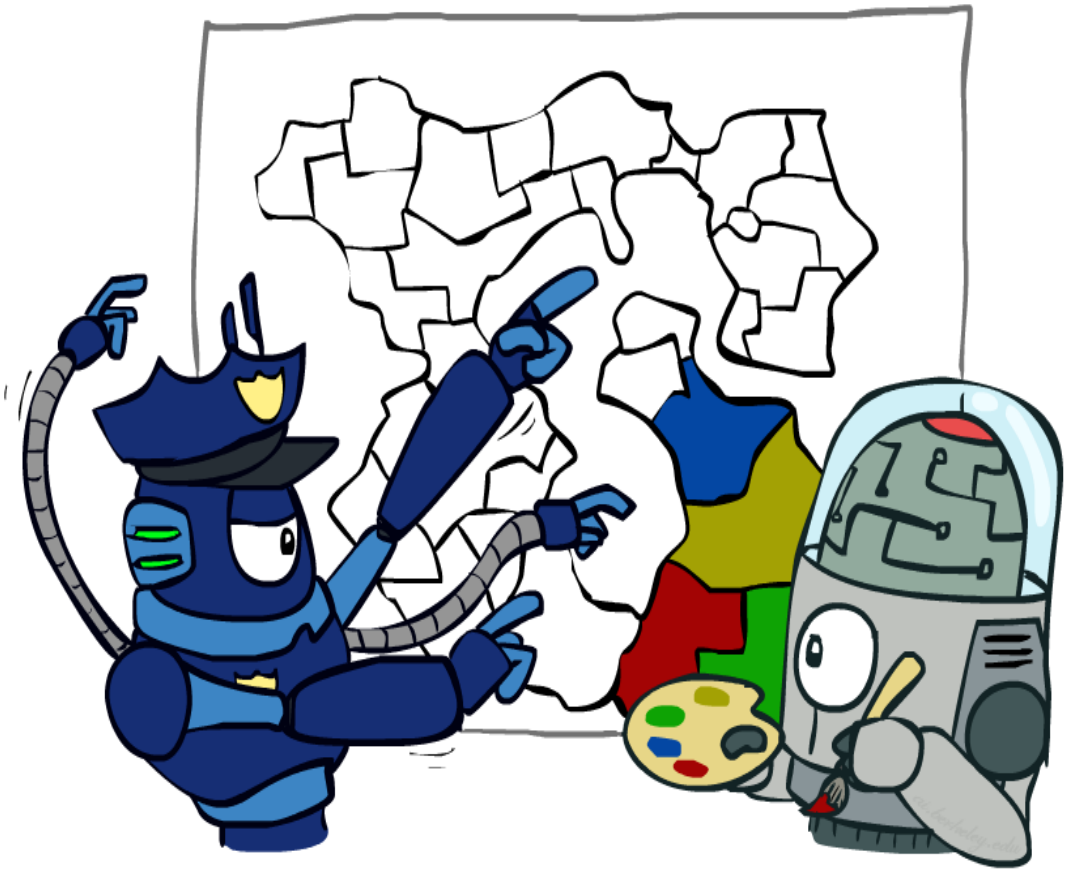
- 变量:
  - 每一个 空白方格
- 值域:
  - {1,2,…,9}
- 约束条件:

  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

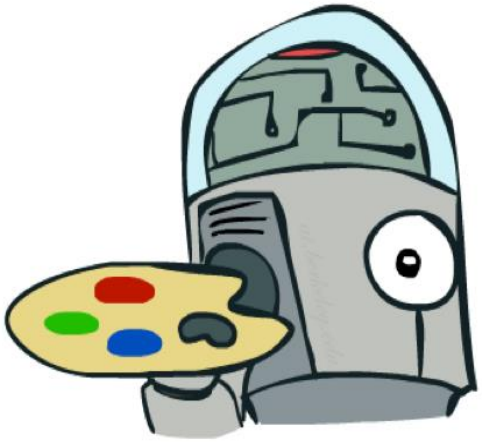
约束满足问题和约束条件的多样性



约束满足问题的多样性

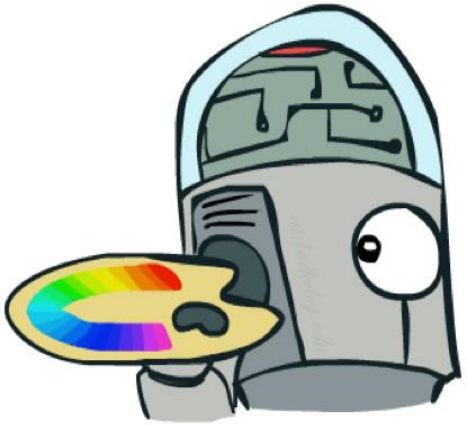
离散变量

- 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*** (SAT 是 NP-complete问题)
- 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
  - 线性约束是可解的, 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

$$SA \neq green$$

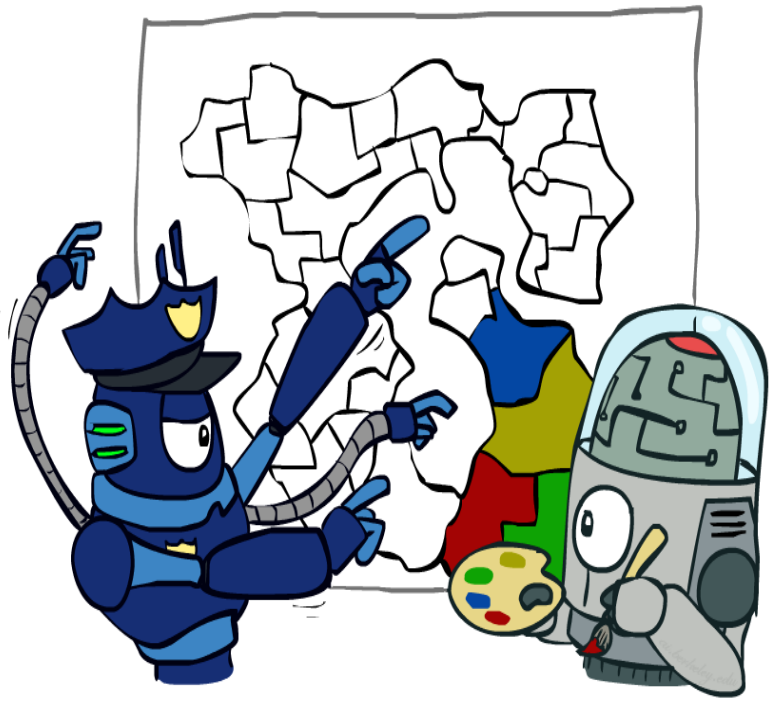
- 二元约束， 涉及成对的变量, 例如:

$$SA \neq WA$$

- 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

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

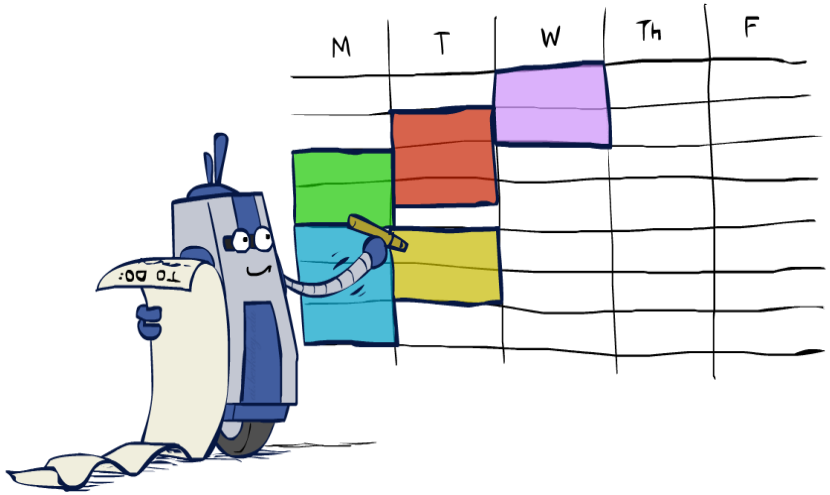
公交时间调度

工厂时间调度

电路设计规划

故障诊断

... 还有许多!



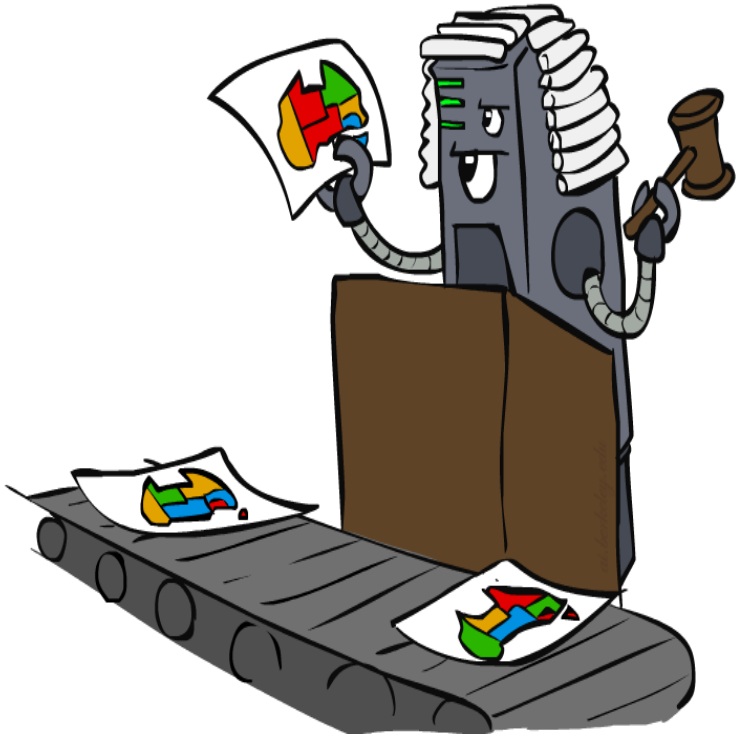
许多真实世界里的问题都涉及实数值变量...

求解约束满足问题

按标准的搜索问题来建立

状态反映了变量赋值的当前情况（部分赋值）

- 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)（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

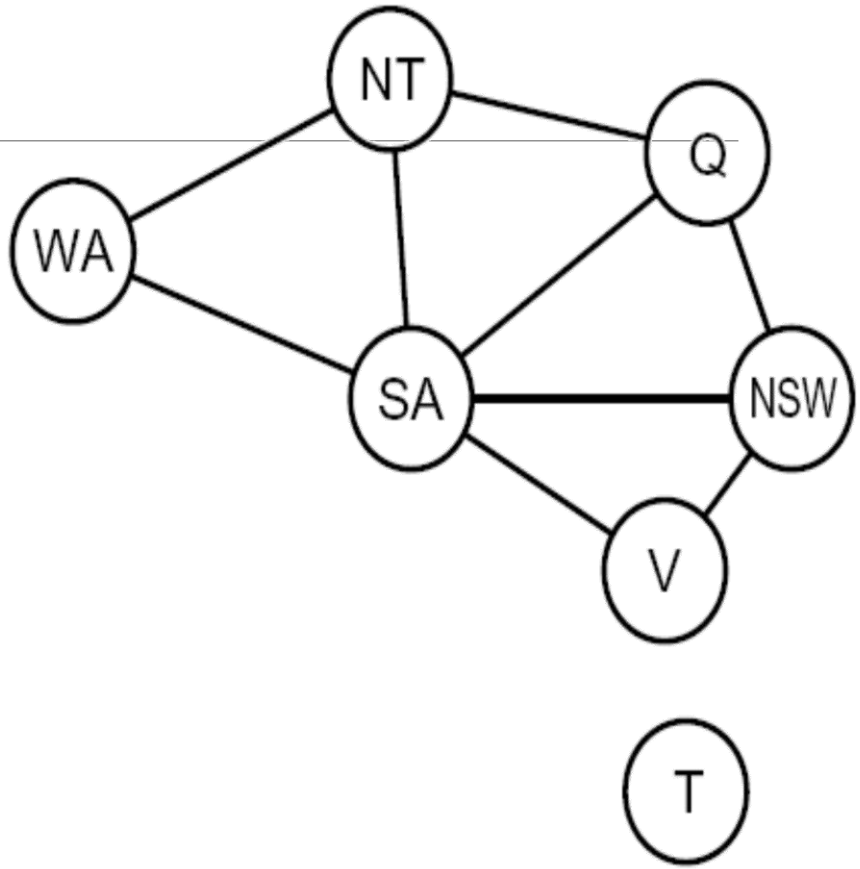


我们开始将用最直接的方法，然后逐步改进

一般搜索方法

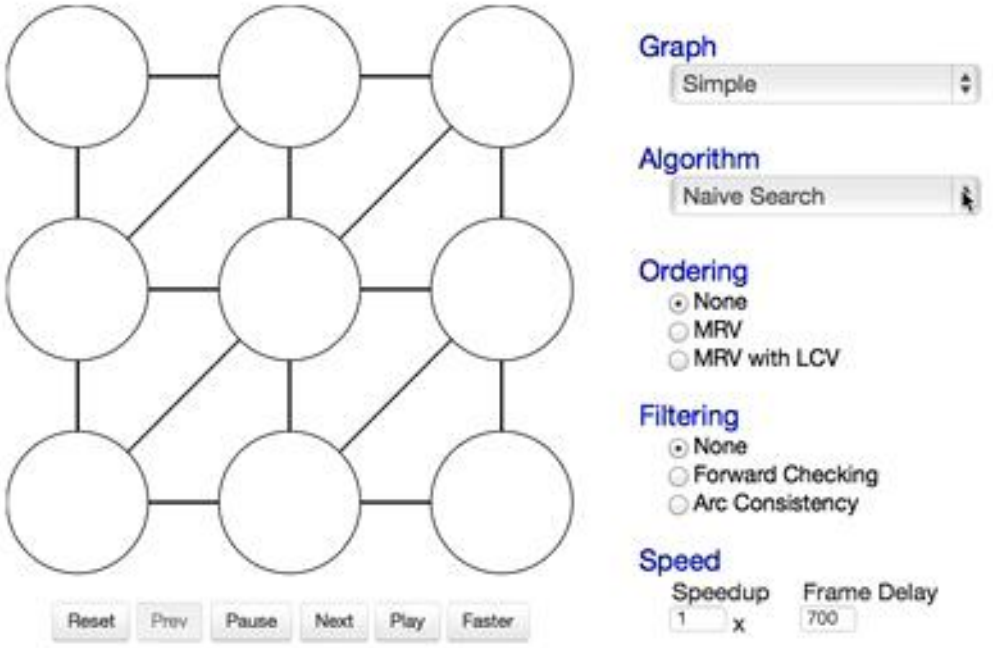
广度优先(BFS)会怎么样?

深度优先(DFS)会怎么样?

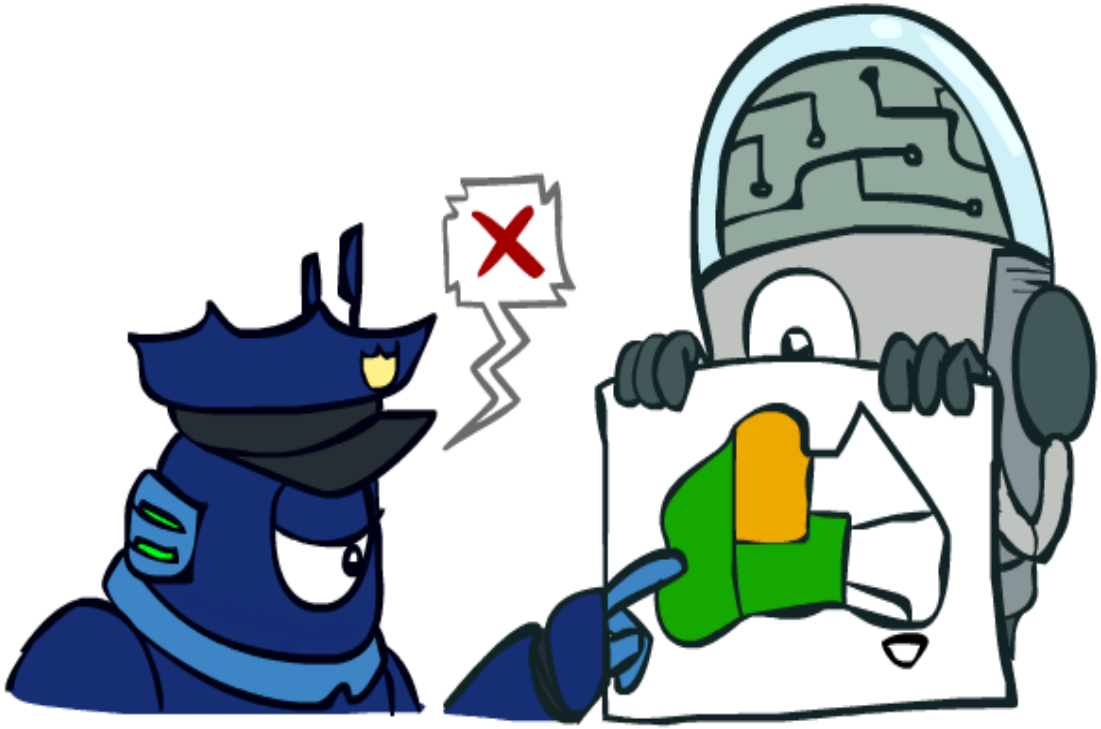


这些最直接的搜索方法在解这个问题时有什么问题?

视频演示：应用简单的DFS，图着色问题



回溯搜索（Backtracking Search）



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 = red then NT = green] 和 [NT = green then WA = 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

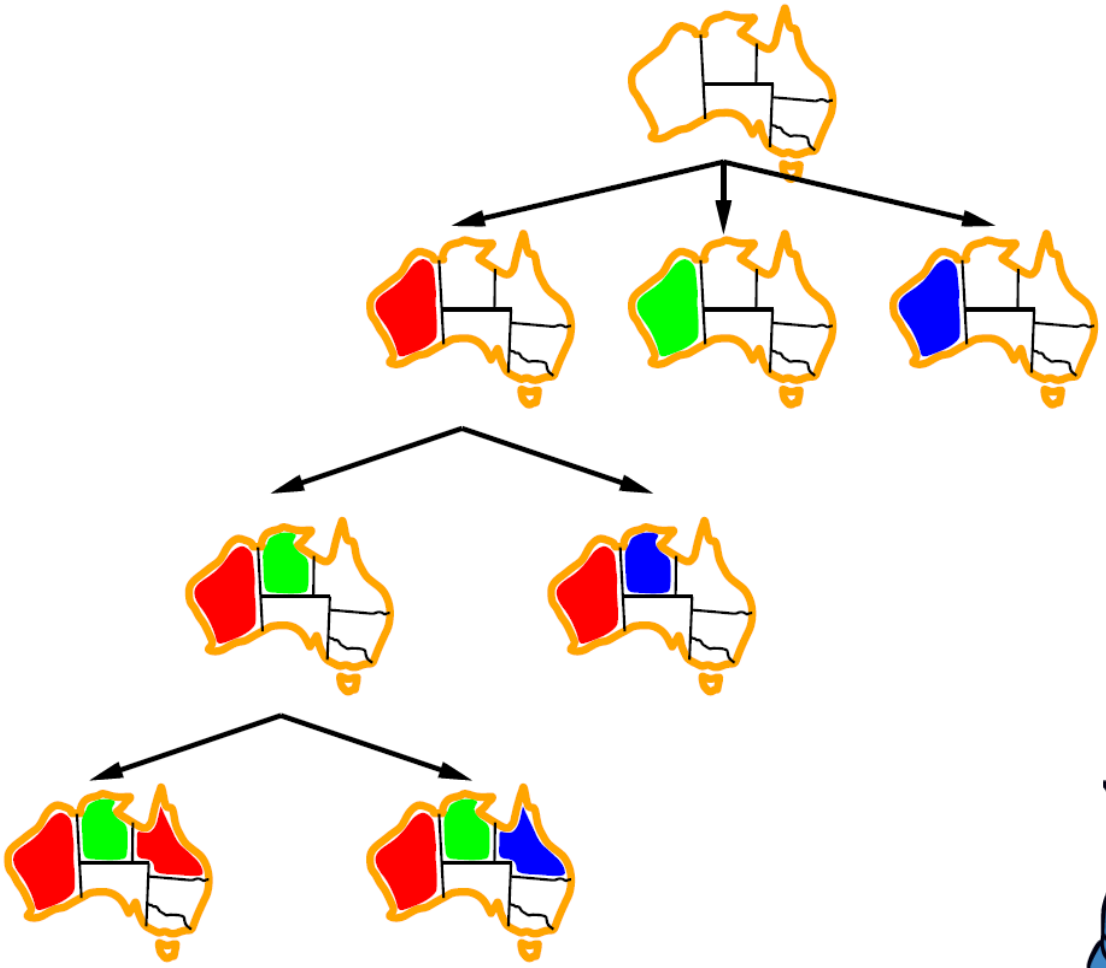
想法 2: 一边探索一边检查约束条件

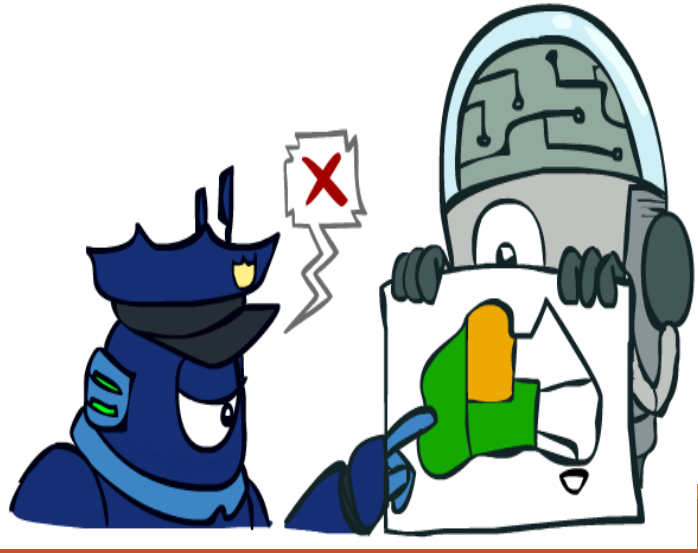
- 探索过程中检查当前的变量赋值是否满足约束条件，和之前已赋值的不冲突
- 也许需要花费一些计算来检查约束条件是否满足
- 相当于 “逐步增加的目标测试”

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 ≈ 25

回溯搜索举例

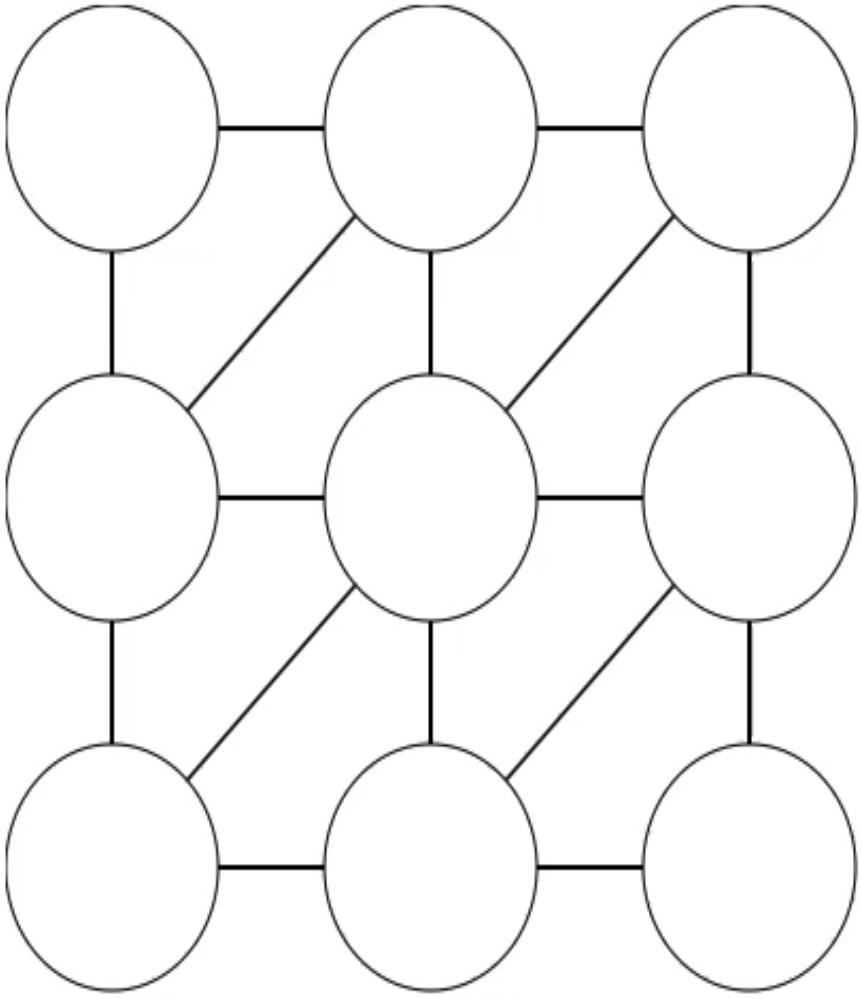




回溯搜索

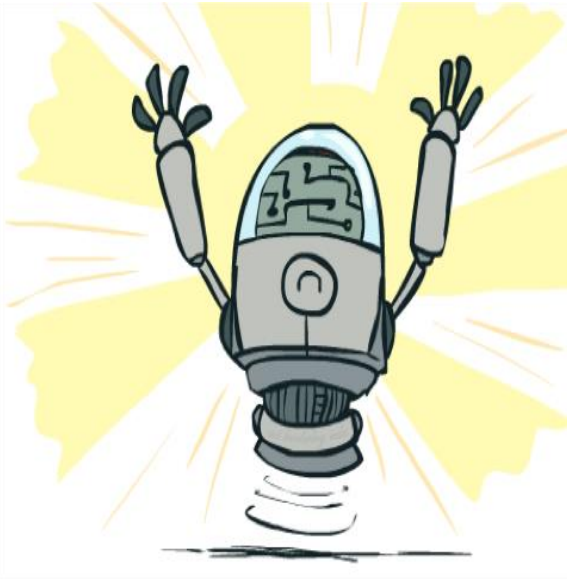
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)
function BACKTRACK(assignment,csp) returns a solution, or failure
    if assignment 是完全的 then return assignment
    var ← 选择-未赋值的-变量(csp,assignment)
    for each value in 排序-值域中的值(var,assignment,csp) do
        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


Graph
Simple
Algorithm
Naive Search
Ordering
None
MRV
MRV with LCV
Filtering
None
Forward Checking
Arc Consistency
Speed
Speedup
Frame Delay
1
x
700
Reset
Prev
Pause
Next
Play
Faster

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



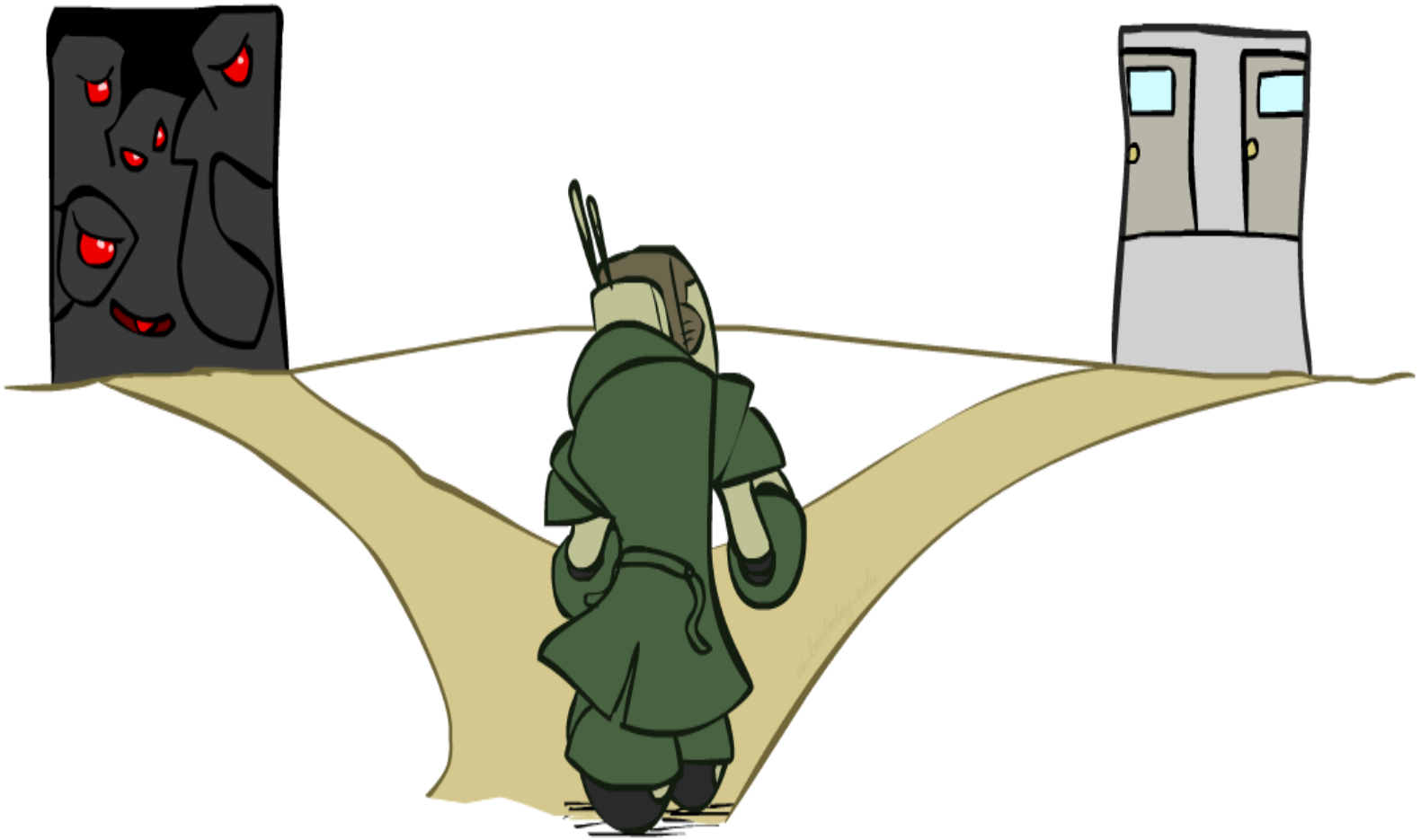
排序(ordering):

- 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: 我们能否利用问题的结构?

排斥

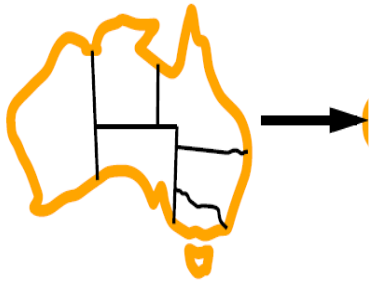


变量排序

var ← 选择-未赋值的-变量(csp,assignment)

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

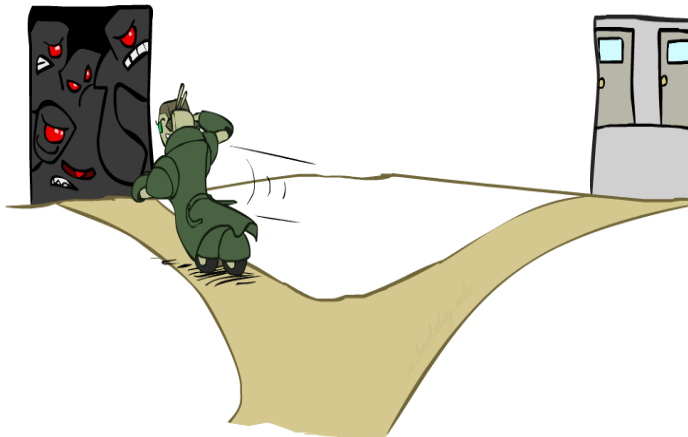


为什么是最少而不是最大?

“快速失败” 排序

使用连接度数启发信息 打破一样的情况

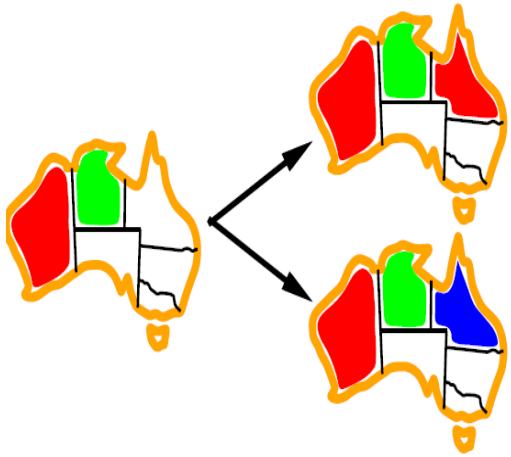
- 选择和其他变量连接数最多的变量（受约束最多的）



图着色演示: backtracking+MRV

对值的排序选择

for each value in 排序-值域里的值(var,assignment,csp) do

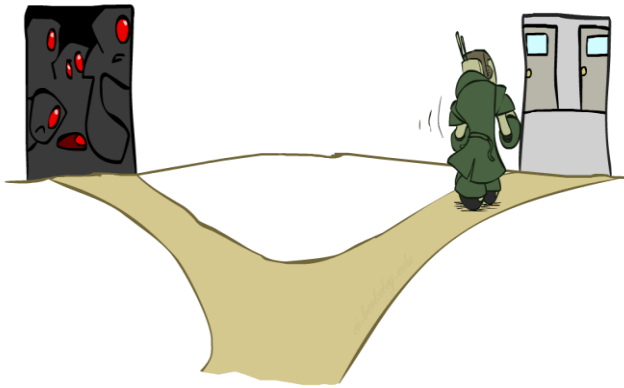


选择最小制约的值 (Least Constraining Value -- LCV)

- 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



过滤（Filtering）

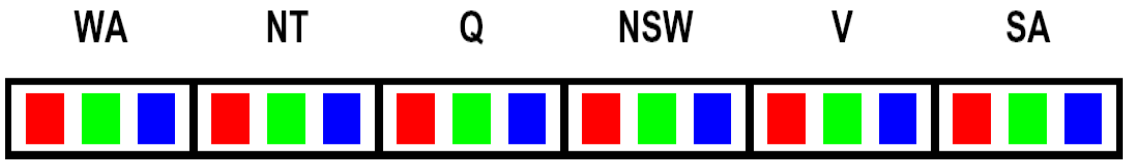
推理 ←**INFERENCE**(csp,var,assignment)
if 推理≠ 失败 **then**
　　添加 推理 到 assignment
　　…

过滤: 前向检查法(Forward Checking)

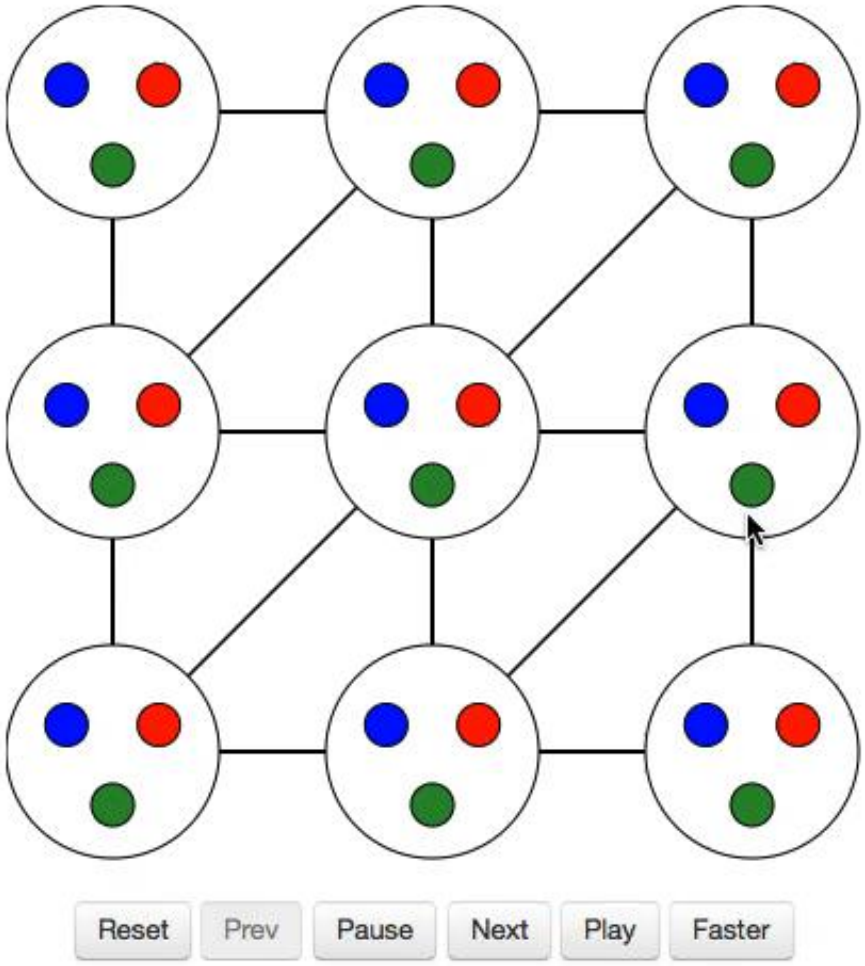
过滤：搜索中, 持续检测未赋值变量的值域，去掉违反约束条件的值

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



Graph

Simple

Algorithm

Backtracking

Ordering

- None
- MRV
- MRV with LCV

Filtering

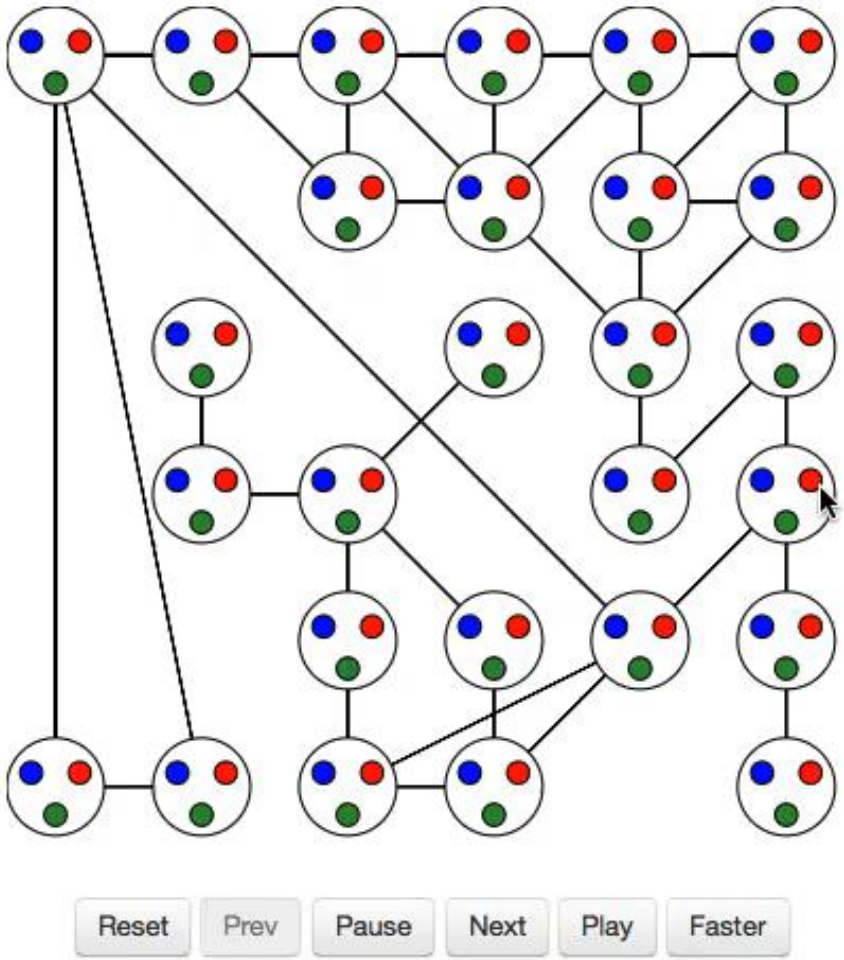
- None
- Forward Checking
- Arc Consistency

Speed

Speedup: 1 x

Frame Delay: 700

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
Frame Delay
1
x
700

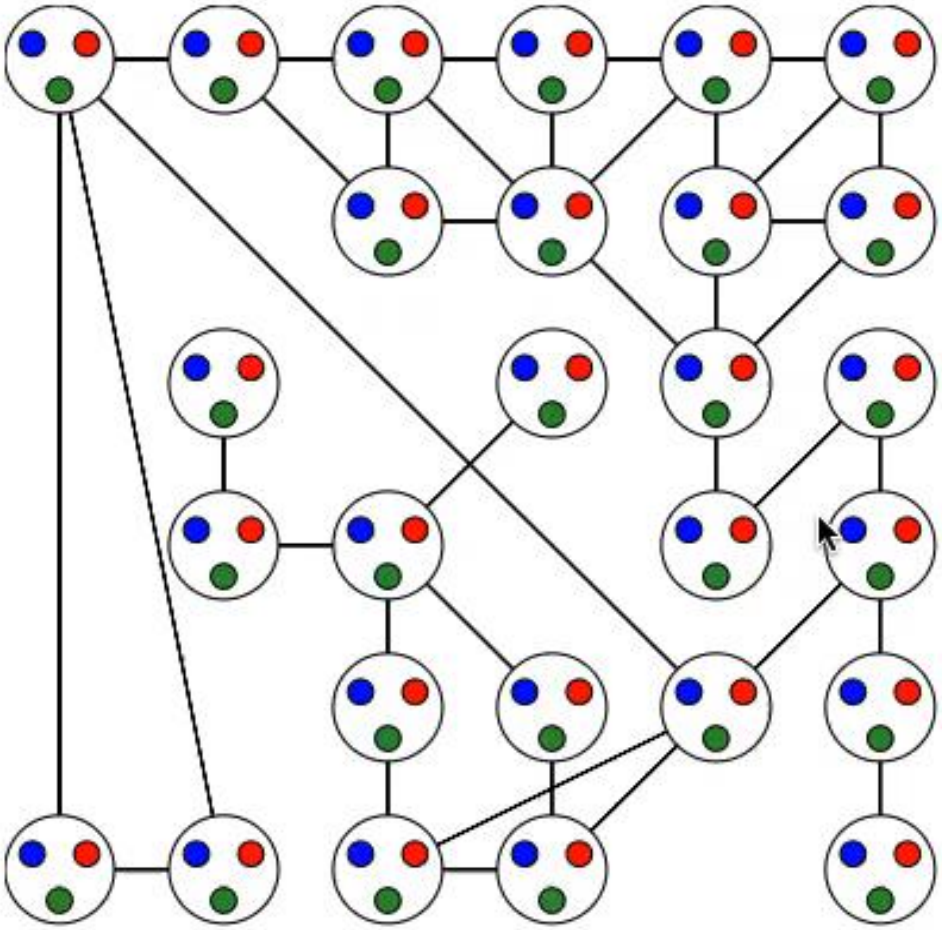
更复杂的推理

维护弧的一致性 -- MAC (maintaining ***arc consistency***) 算法可以比前向检查法更早地检测出许多失败的情况

请见书的第 6.2 章节

我们会在稍后简单介绍相关的定义和应用。

图着色问题演示：回溯算法 + 前向检查 + 变量排序



Reset Prev Pause Next Play Faster

Graph

Complex

Algorithm

Backtracking

Ordering

- None
- MRV
- MRV with LCV

Filtering

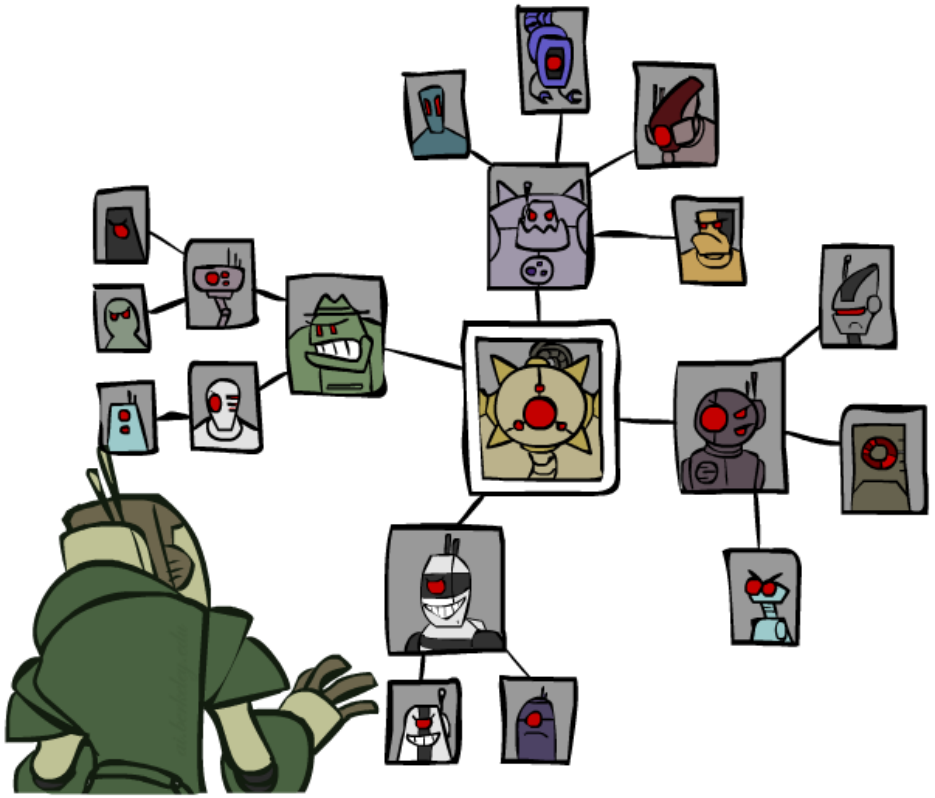
- None
- Forward Checking
- Arc Consistency

Speed

Speedup: 1 x

Frame Delay: 700

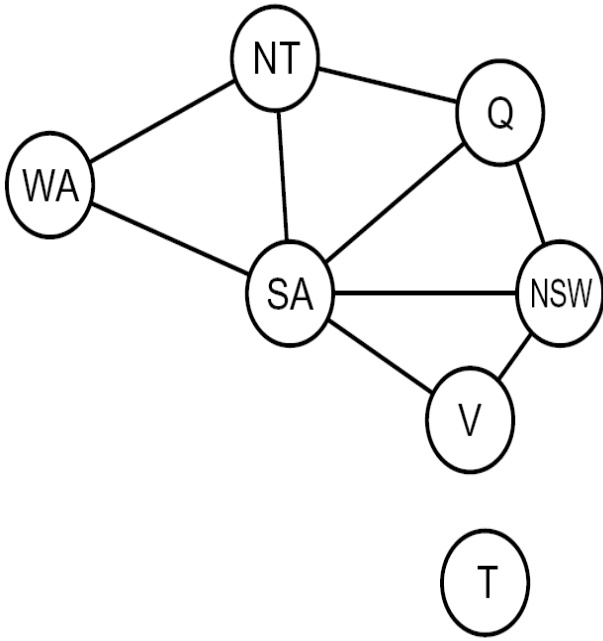
利用问题的结构



问题结构

极端情况: 独立的子问题

- 例如: Tasmania 和大陆不相连



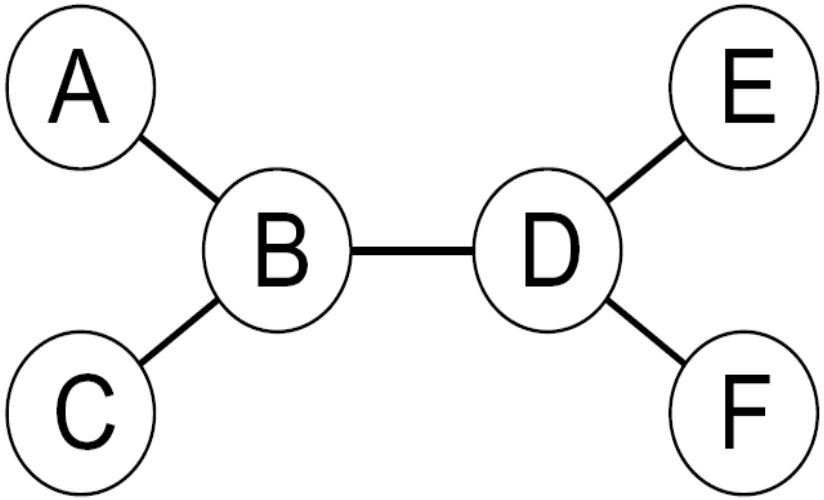
独立的子问题可以被认为是约束图中连接的组件: 可采用:

- ***分治法!***

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: 4×2^{20} = **0.4 秒**

树结构的 CSPs

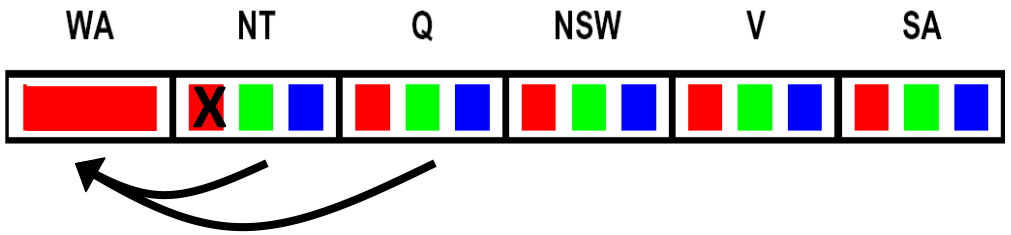


定理: 如果约束图无环(树结构的)，则其对应的约束满足问题的求解时间复杂度是 $O(n\ d^2)$

- 比较一般的 CSPs, 最差时间复杂度是 $O(d^n)$

弧的一致性（Arc consistency）

一个弧 X → 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



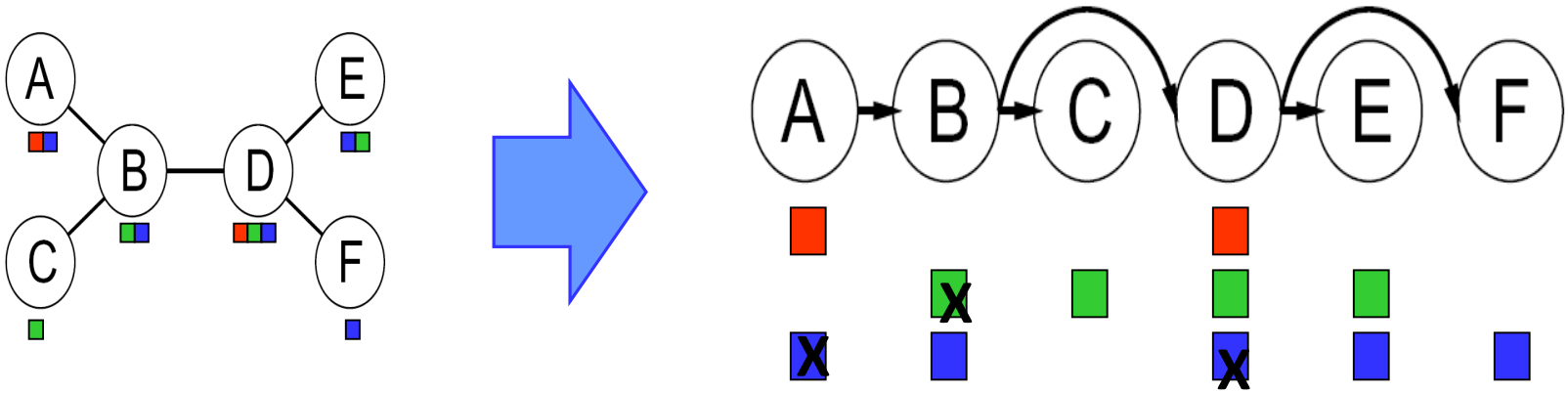
从弧的尾部删除

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

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



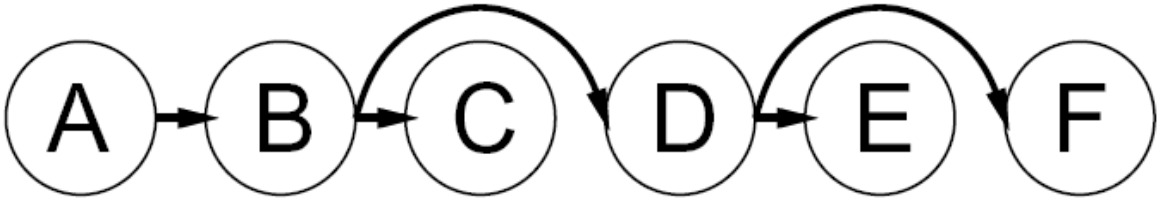
- **从后向前删除**: For i = n : 2, 执行： 删除不一致的值(父节点(X_i),X_i)
- 从前向后赋值: For i = 1 : n, 赋值 X_i 和父节点 Parent(X_i) 相一致的值

运行时间: $O(n\ d^2)$

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->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)



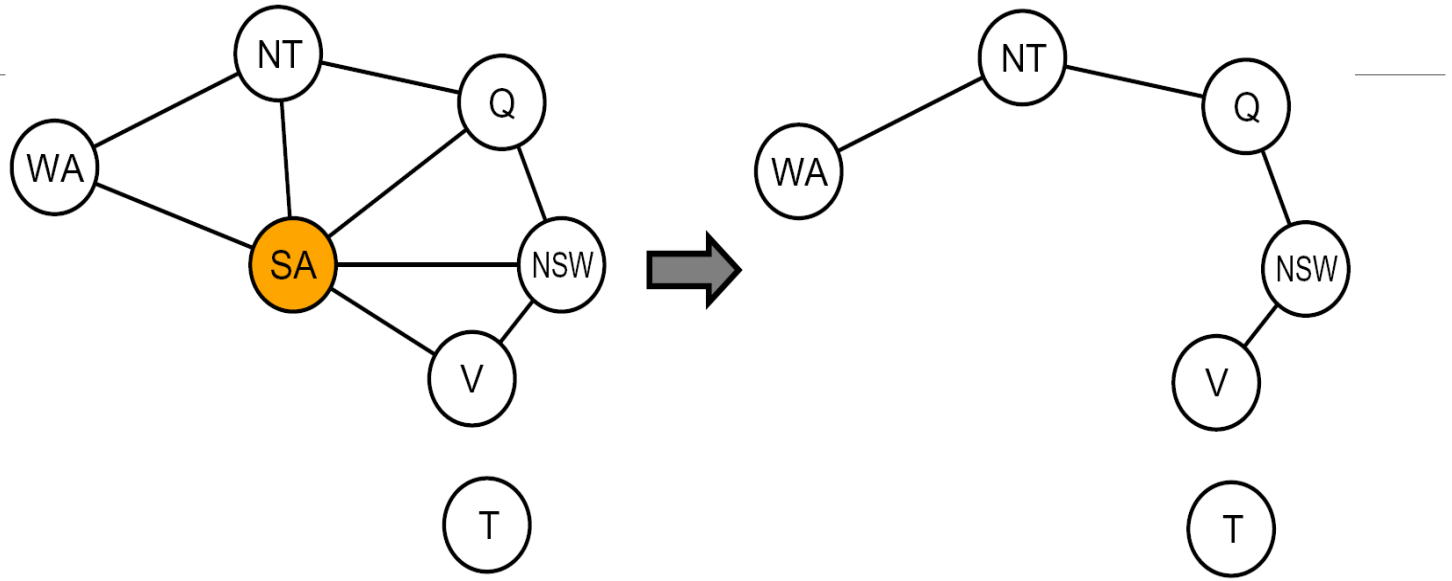
声明 2: **如果从根到叶的弧都是一致性的**, 那么从前向后的赋值过程将不会有回溯

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

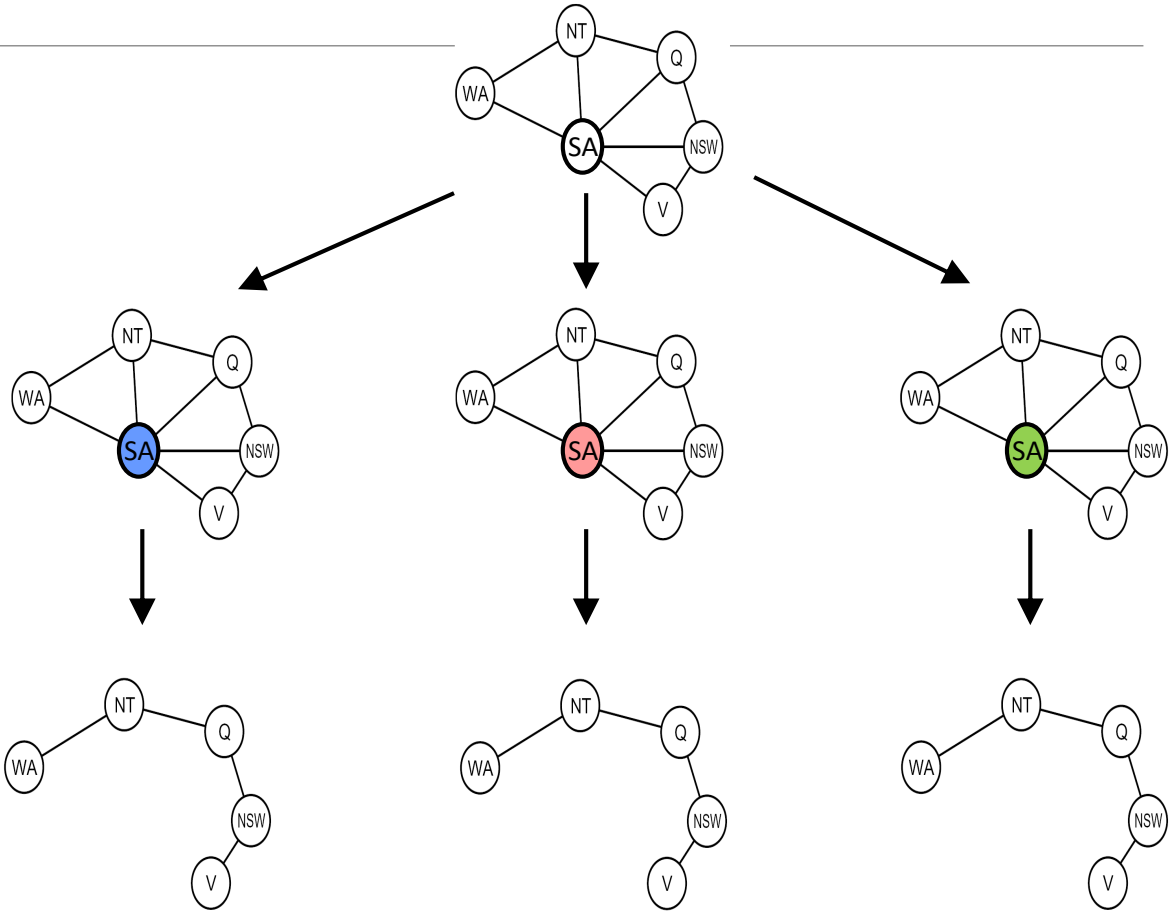
切集条件化制约

选择一个切集

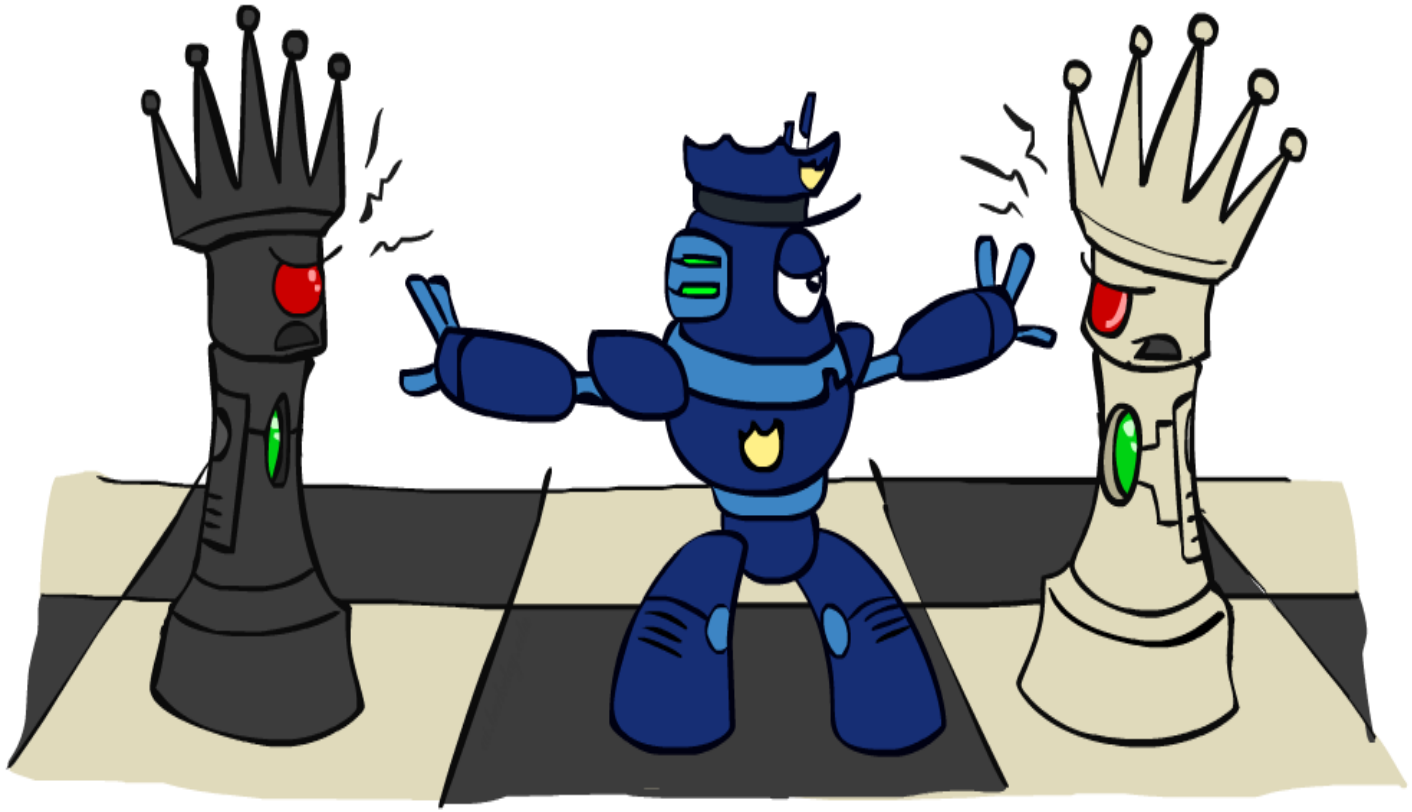
对切集变量赋值(所有可能组合)

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
  - 随机打破平局情况

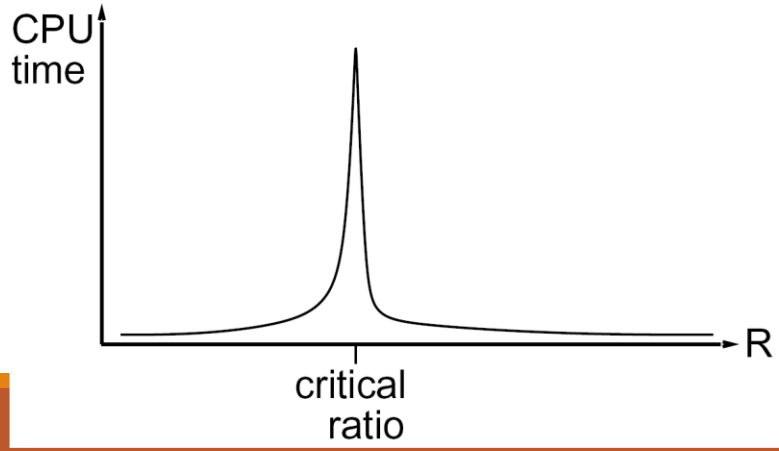
图着色演示: 最小冲突法

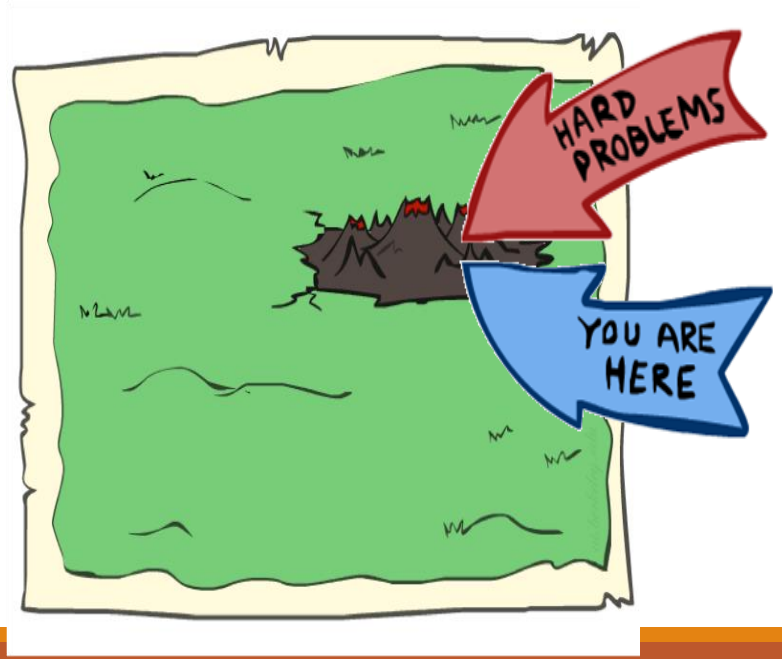
最小冲突算法的表现性能

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

$$R = \frac{\text{约束条件数}}{\text{变量数}}$$



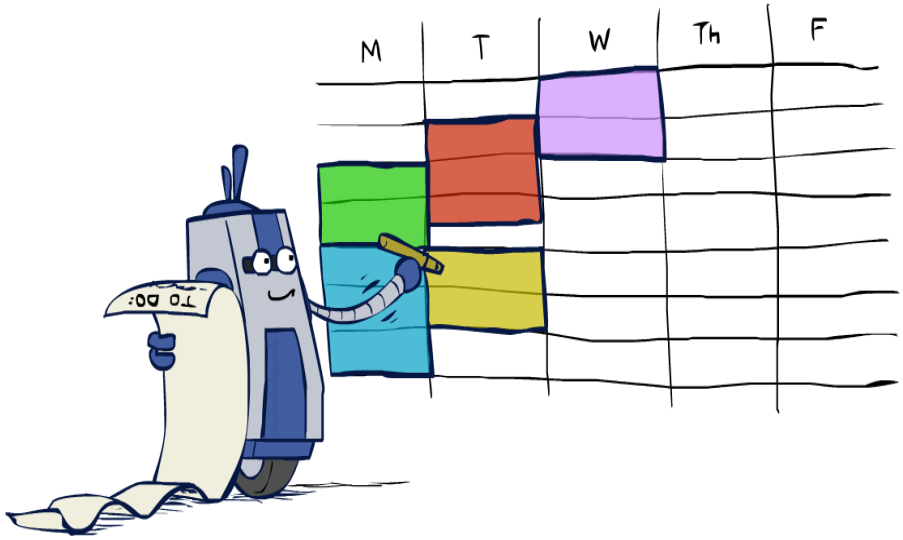


总结：约束满足问题

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