

第六章 约束满足问题

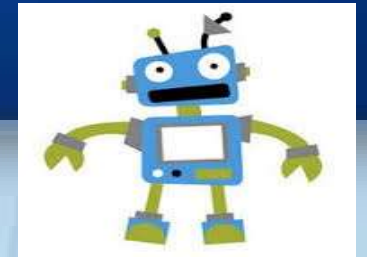
Constraint Satisfaction Problems (CSP)

主讲人 赵曼

中国地质大学 计算机学院计算机科学系

提纲

- 约束满足问题定义
- 约束满足问题放入标准搜索
- 回溯搜索算法求解CSP
- 局部搜索算法求解CSP
- 小结



约束满足问题

- 标准搜索问题：
 - 状态是一个黑盒，通过后续函数，启发式函数和目标测试来访问。
- CSP：
 - 状态由多个变量 X_i 定义，变量 X_i 的值域为 D_i
 - 目标测试是由约束集来确定，约束指定变量子集允许的赋值组合。

约束满足问题表示

- **CSP形式化:**

- 有限变量集: $\{X_1, X_2, \dots, X_n\}$
- 有限约束集: $\{C_1, C_2, \dots, C_m\}$
- 每个变量有非空可能值域: $D_{x1}, D_{x2}, \dots, D_{xn}$
- 每个约束 C_i 的值可以用变量来表示: 例如 $X_1 \neq X_2$

- 一个状态被定义为所有变量的赋值

一个相容的赋值: 所有赋值均满足所有约束

例子: 地图着色问题

- 变量: WA, NT, Q, NSW, V, SA, T
- 域: $D_i = \{\text{red}, \text{green}, \text{blue}\}$
- 约束: 相邻区域不能着相同的颜色



例子: 地图着色问题



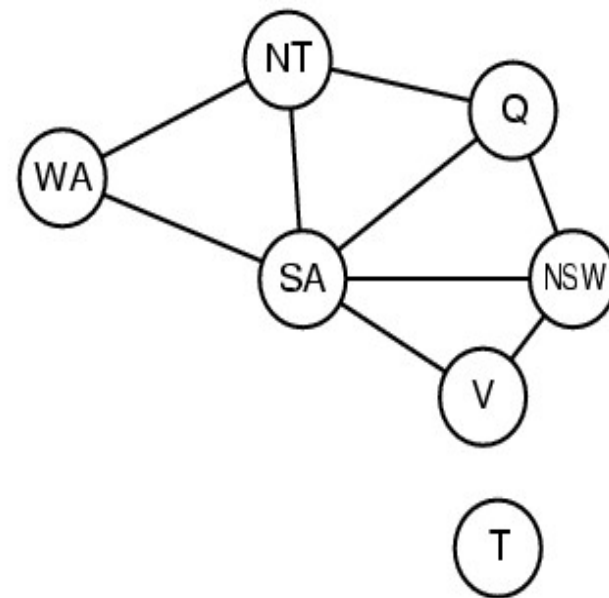
- 解是完整而且符合约束的赋值,

如: WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

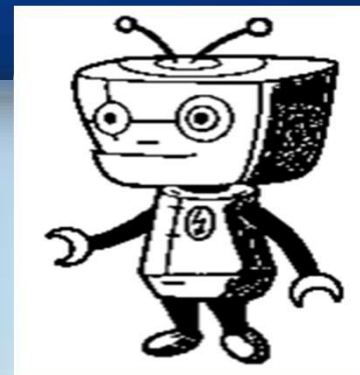
- 一些CSP问题要求一个目标函数的最佳解。

约束图：

- 二元 **CSP**：每个约束都关联两个变量
- 约束图：结点表示变量，边表示约束
- **CSP**的特点（优势）
 - 呈现出标准的模式
 - 通用的目标和后继函数
 - 通用的启发函数（无域的特别技术）



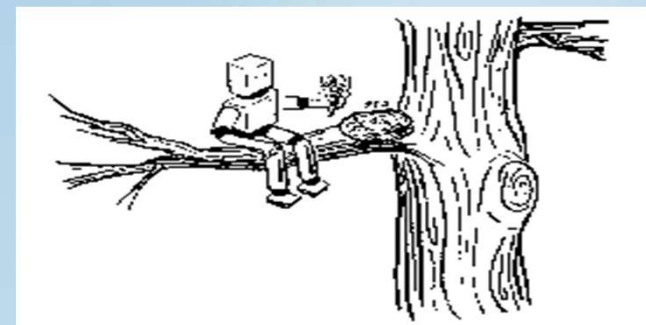
CSP的变量类型



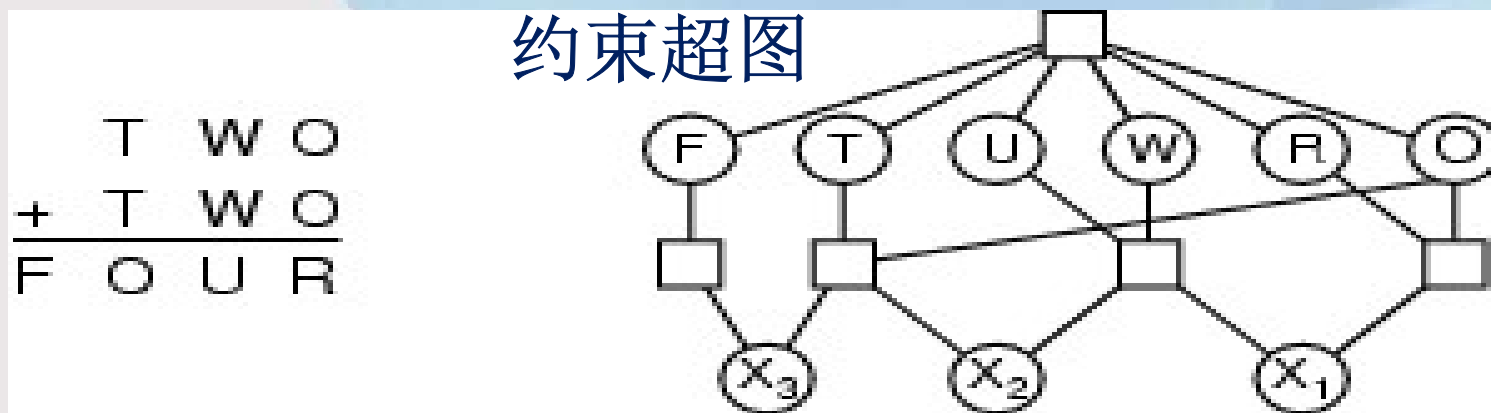
- 离散变量
 - 有限域:
 - n 个变量, 域大小为 $d \rightarrow O(d^n)$
 - 无限域:
 - 作业调度, 变量是工序的起始和结束日期
 - 约束: $StartJob_1 + 5 \leq StartJob_3$
- 连续变量
 - 哈勃太空望远镜观测的起始和结束时间

变量约束类型

- 一元约束只涉及到一个变量
 - e.g., $SA \neq \text{green}$
- 二元约束涉及到两个变量
 - e.g., $SA \neq WA$
- 高阶约束涉及到3个以个的变量
 - e.g., 密码算术约束
- 偏好约束



例子:密码算术



• 变量:

$FTUWR O \mathbf{X_1 X_2 X_3}$

• 域:

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

• 约束: $Alldiff(F,T,U,W,R,O)$

$$O + O = R + 10 \cdot \mathbf{X_1}$$

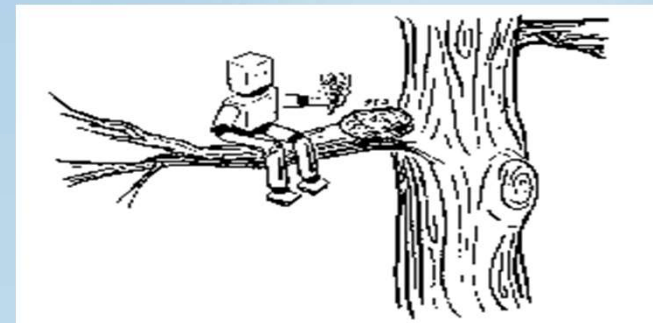
$$\mathbf{X_1} + W + W = U + 10 \cdot \mathbf{X_2}$$

$$\mathbf{X_2} + T + T = O + 10 \cdot \mathbf{X_3}, T \neq 0$$

$$\mathbf{X_3} = F, F \neq 0$$

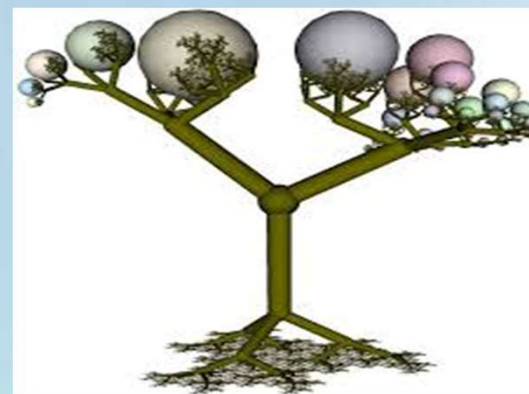
变量约束类型

- 一元约束只涉及到一个变量
 - e.g., $SA \neq \text{green}$
- 二元约束涉及到两个变量
 - e.g., $SA \neq WA$
- 高阶约束涉及到3个以个的变量
 - e.g., 密码算术约束
- 偏好约束（软约束）
 - e.g., 用一个代价值代表红色比绿色好



真实世界 CSPs

- 分配问题
- 排课问题
- 运输调度
- 生产调度



标准搜索形式化CSP(增量形式化)

CSP的特点（优势）

- 呈现出标准的模式
- 通用的目标和后继函数
- 通用的启发函数（无域的特别技术）

状态用已经给变量赋的值来表示



- 初始状态：空 $\{\}$
- 后续函数：给一个没有赋值的变量赋值，此值与已经赋的值不能与约束冲突。
- 目标测试：当前赋值是完整赋值，即所有变量都有值。

标准搜索形式化CSP(增量形式化)

- 有 n 个变量的问题，解都在深度为 n 的层
→ 可以使用深度优先搜索
- 在深度为 ℓ 时，分支数 $b = (n - \ell)d$ ，所以搜索树有 $n! \cdot d^n$ 个叶子结点

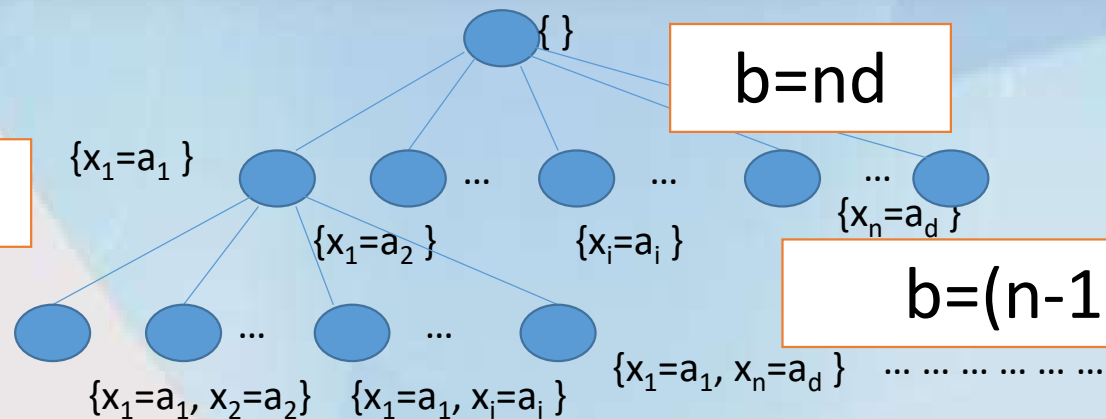


标准搜索形式化(增量形式化)

- 状态表示: x_1, x_2, \dots, x_n
- 最大 $|D_i|=d$

$$nd$$

$$n(n-1)d^2$$



$$b=nd$$

$$b=(n-1)d$$

$$n(n-1)\dots 2d^{n-1}$$

$$\{x_1=a_1, x_2=a_1, \dots, x_i=a_i, \dots, x_{n-1}=a_1\}$$

$$n!d^n$$

$$b=d$$

$$\{x_1=a_1, x_2=a_1, \dots, x_i=a_i, \dots, x_{n-1}=a_1, x_n=a_d\}$$



回溯搜索

- 变量赋值是可交换的, i.e.,
[WA = red then NT = green] 等同于 [NT = green then WA = red]
- 每个结点只需要考虑给一个变量赋值
→ $b = d$ 所以 d^n 片叶子
- 采用单变量赋值的深度优先搜索称为回溯搜索

回溯搜索算法



```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to Constraints[csp] then
      add { var = value } to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove { var = value } from assignment
  return failure
```

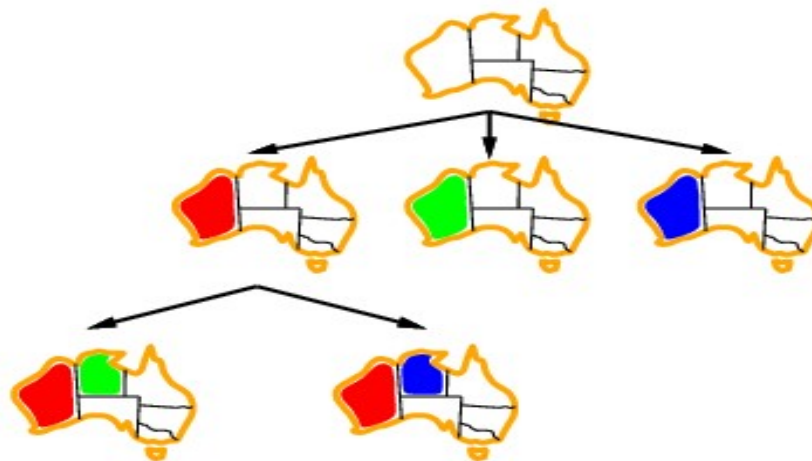
回溯搜索示例



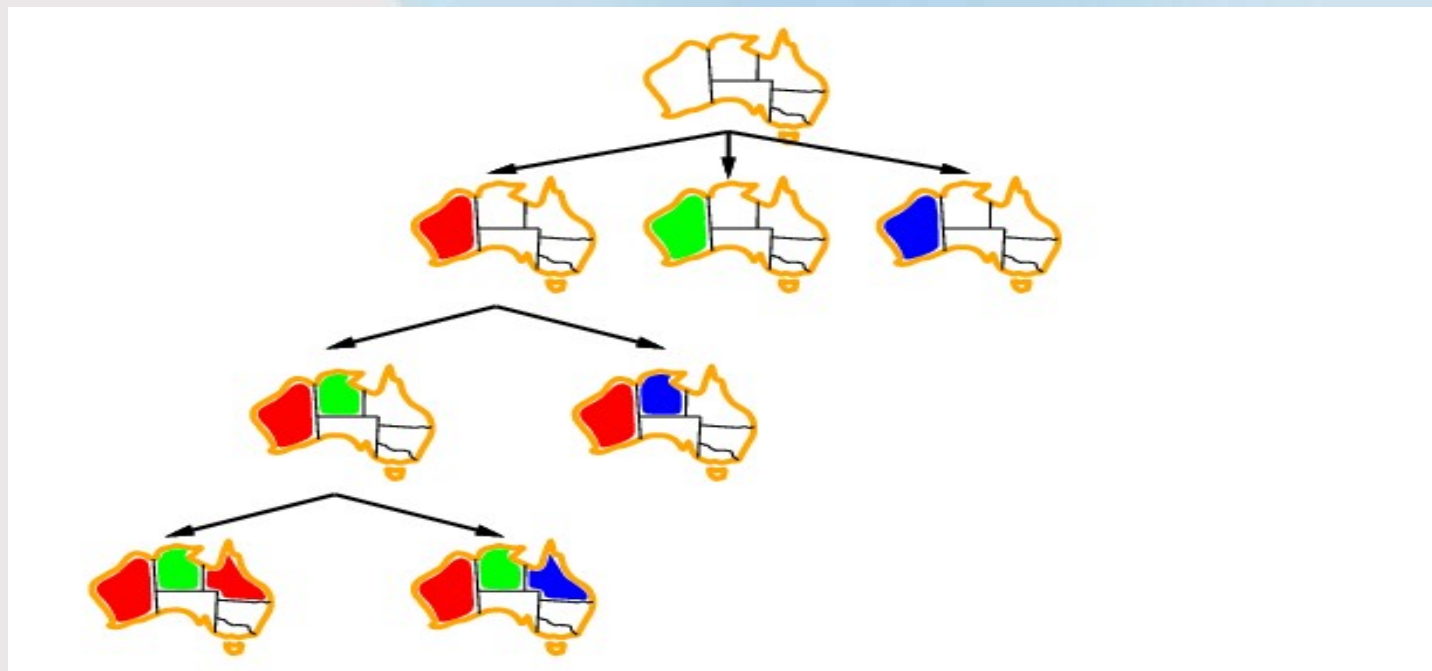
回溯搜索示例



回溯搜索示例



回溯搜索示例

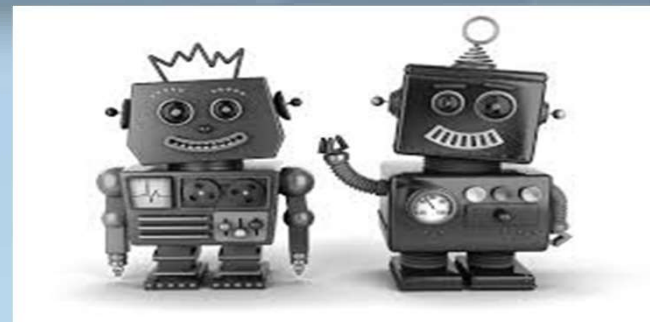


提高回溯搜索的效率

- 通用方法就能巨大提高效率：
 - 下一步应该给哪个变量赋值？
 - 应该以一种什么顺序来试着给变量赋值？
 - 能不能早些检测到不可避免的失败？

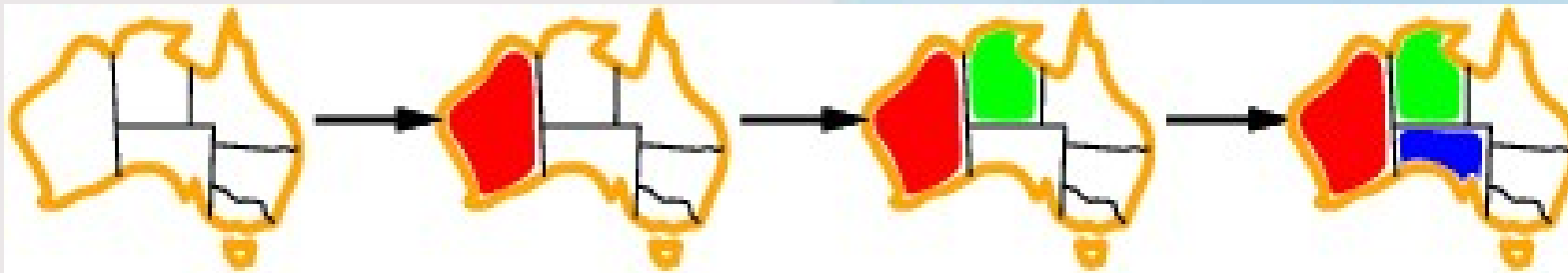
如何较早探测到不可避免的失败，并且避免它

```
if assignment is complete then return assignment  
var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)  
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do  
    if value is consistent with assignment according to Constraints[csp] then  
        add { var = value } to assignment
```



回溯改进——最大受限变量

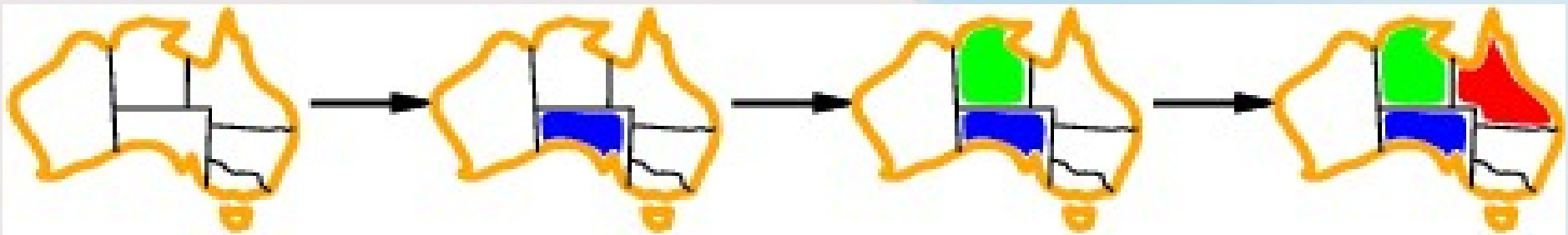
- 最大受限变量（most constrained variables）：
具有最少合法赋值的变量



- 也称为最少剩余值**启发式**
(minimum remaining values , MRV)

回溯改进——最大约束变量

- 当多个变量的MRV值相同时，采用最大约束准则（Most constraining variable）从中选择
- 选择约束其他未赋值变量最多的变量



回溯改进——最少约束值

- 选择好了变量后，选择一个最少约束值来尝试赋值
- 即选择一个对其他未赋值变量的合法赋值影响最少的值



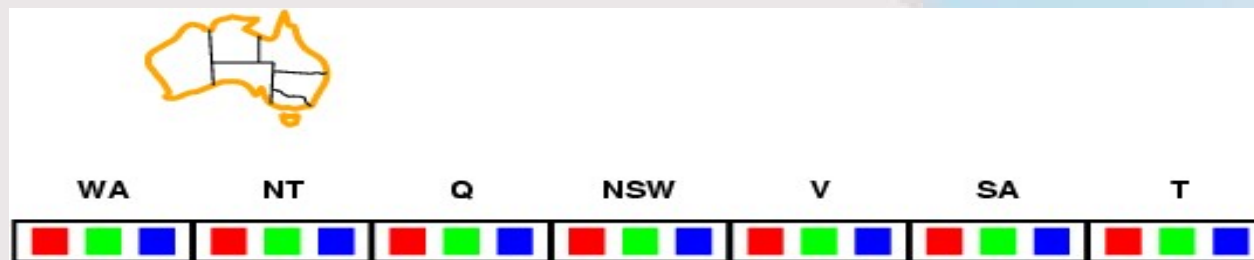
前向检查

- 我们如何较早探测到不可避免的失败？

——并且在后面避免它。

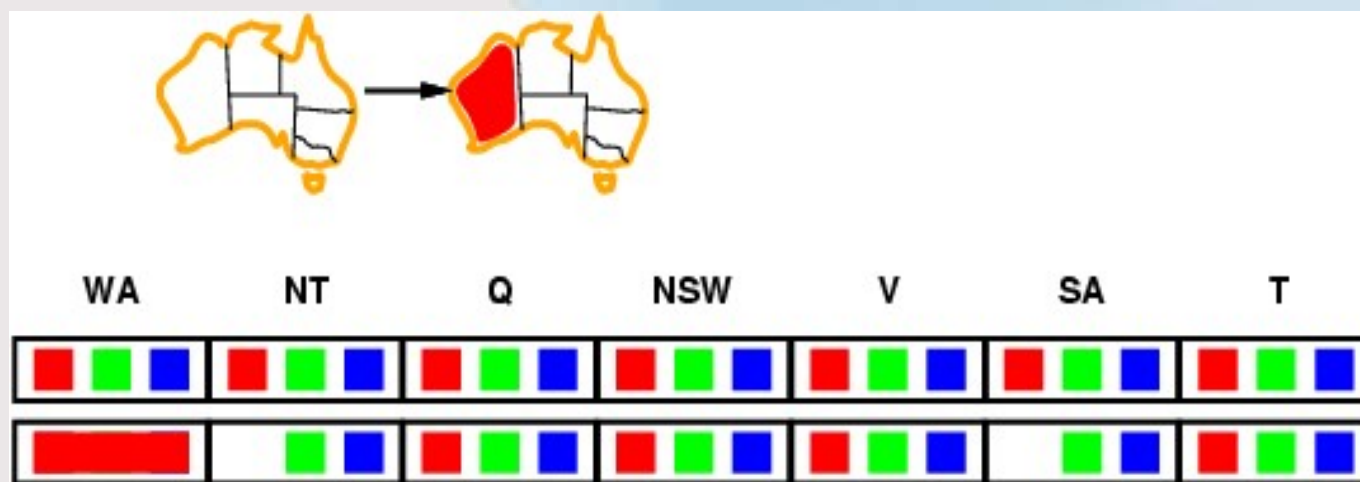
- 向前检查的思想：只保留未赋值的变量的合法值。
- 当任何变量出现没有合法可赋值的情况终止搜索。

跟踪未赋值变量的合法赋值，当发现有变量没有合法赋值时就停止搜索



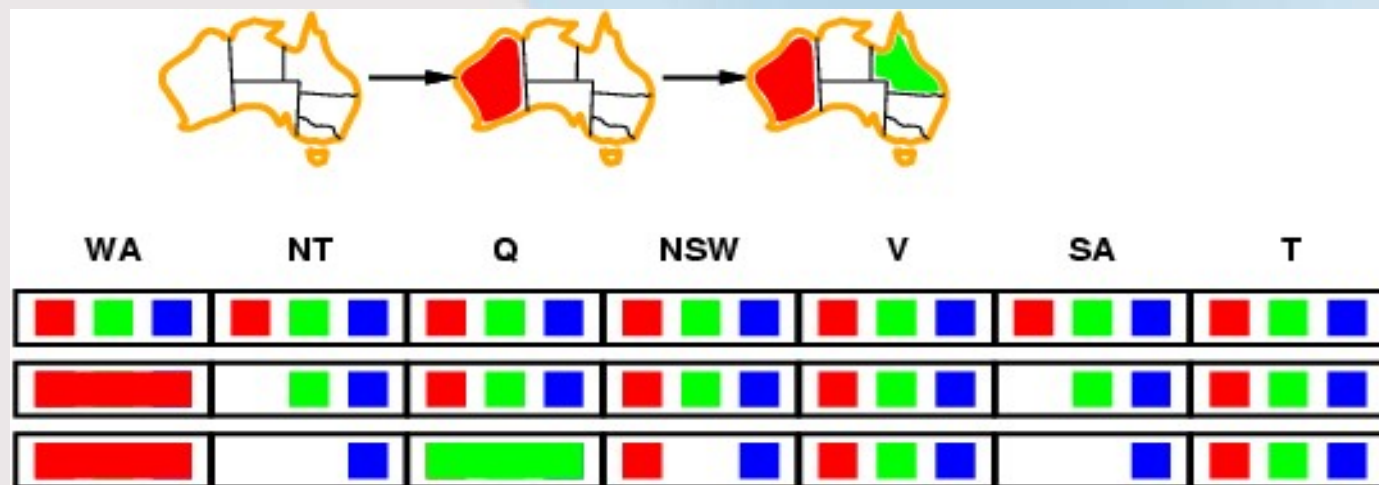
前向检查

- 跟踪未赋值变量的合法赋值，当发现有变量没有合法赋值时就停止搜索



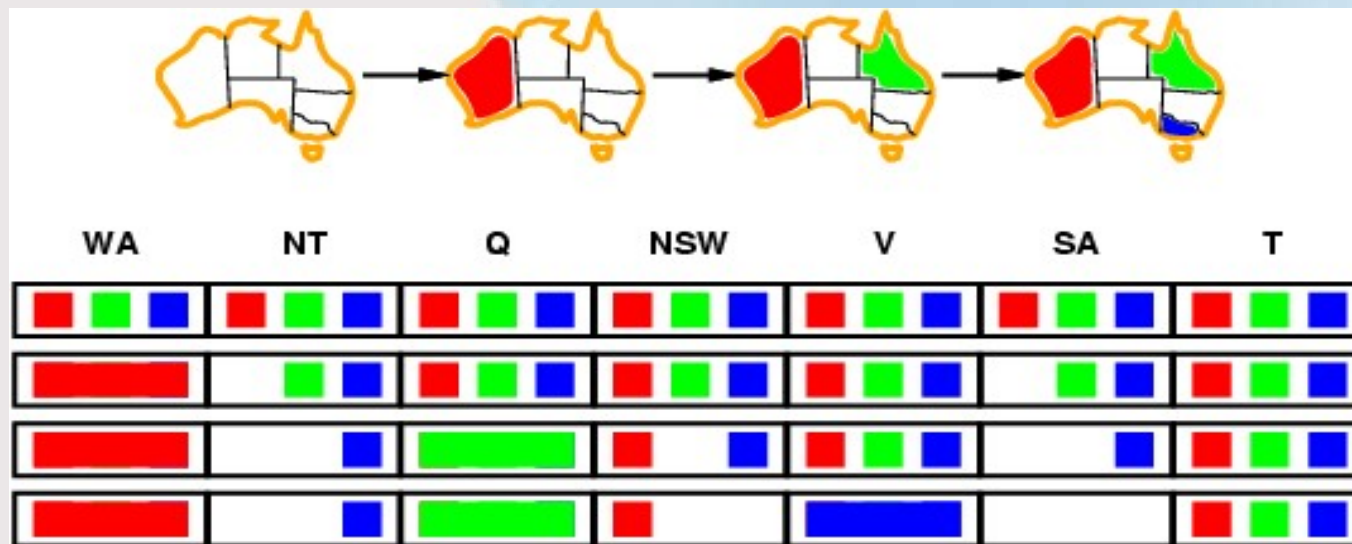
前向检查

- 跟踪未赋值变量的合法赋值，当发现有变量没有合法赋值时就停止搜索



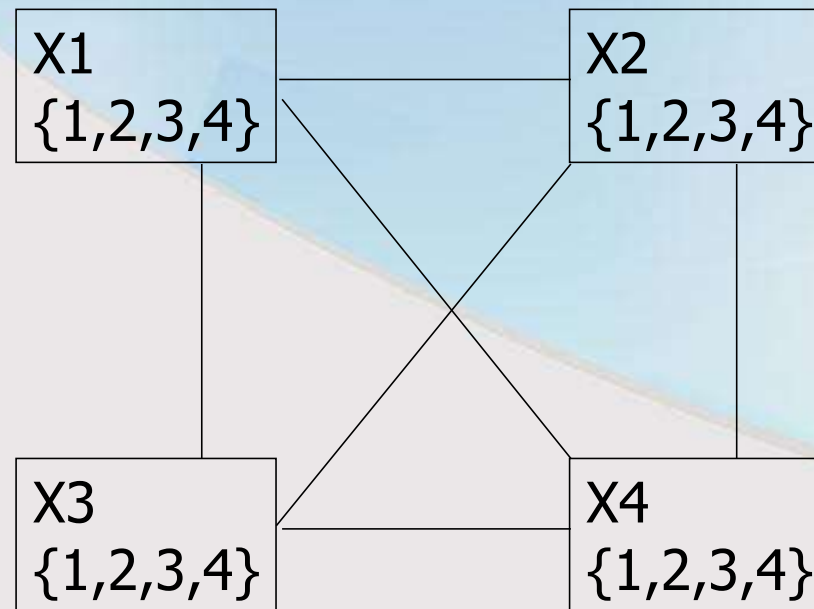
前向检查

- 跟踪未赋值变量的合法赋值，当发现有变量没有合法赋值时就停止搜索










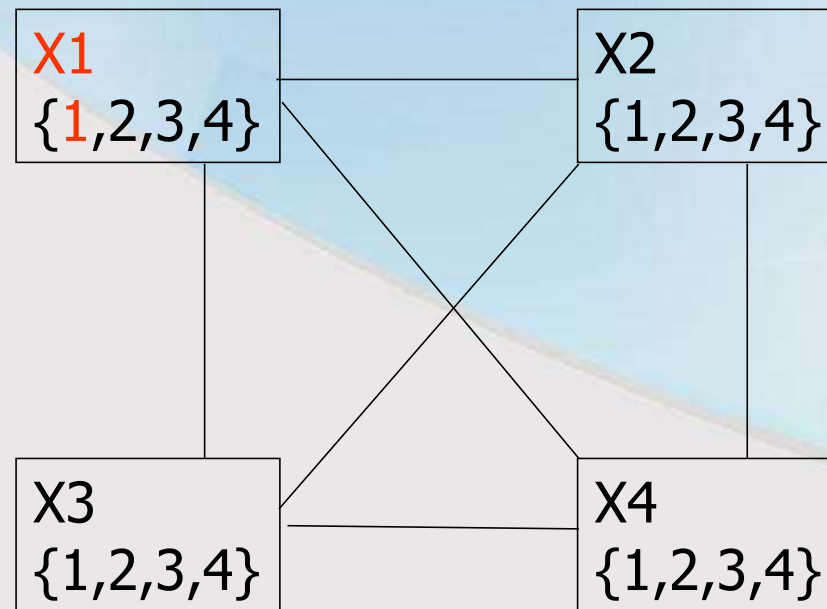
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				










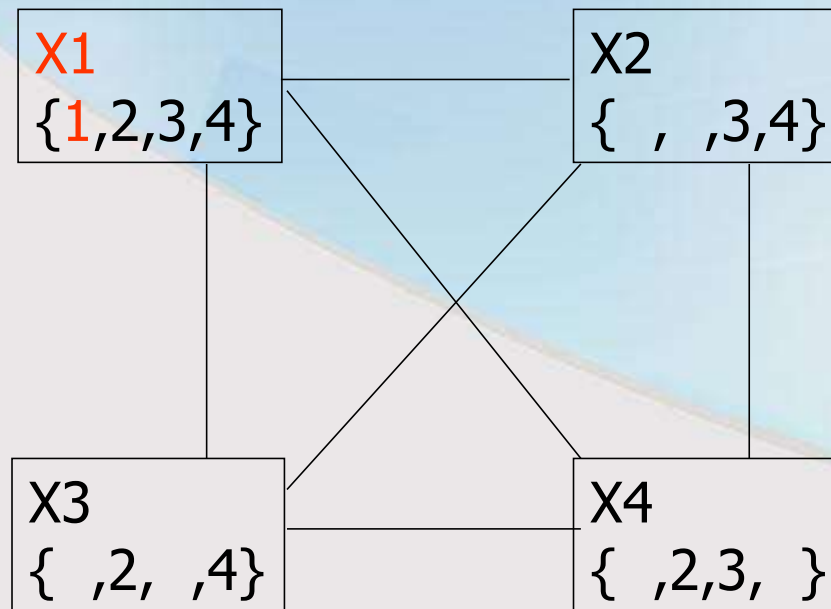
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				


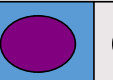
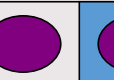


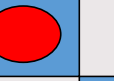







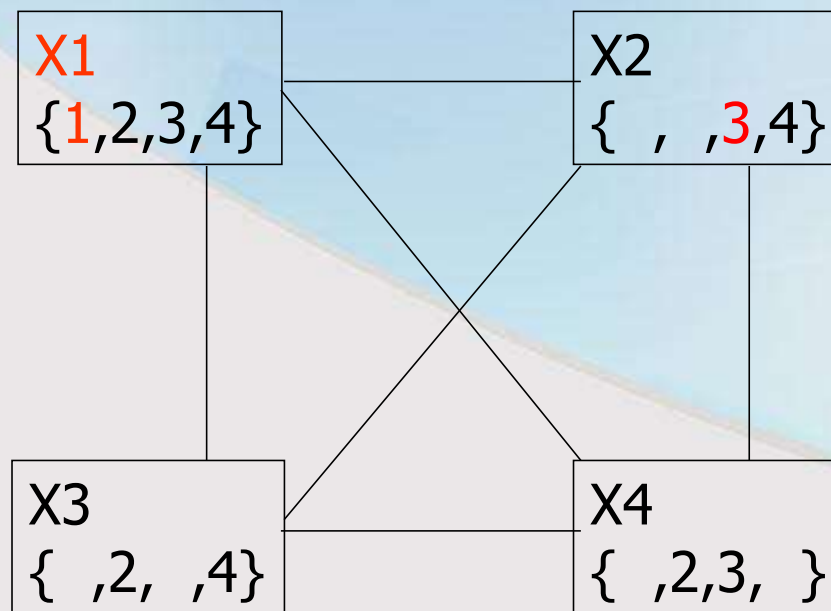
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				



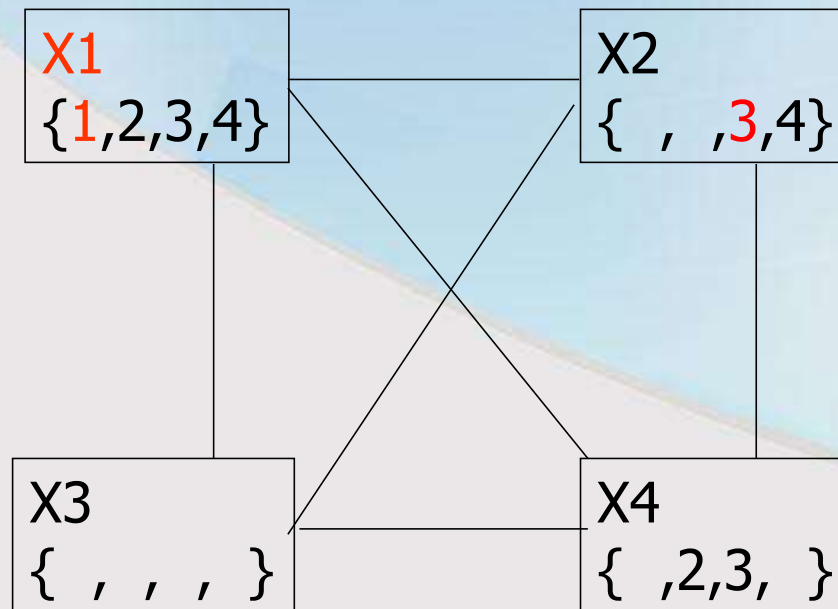
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				



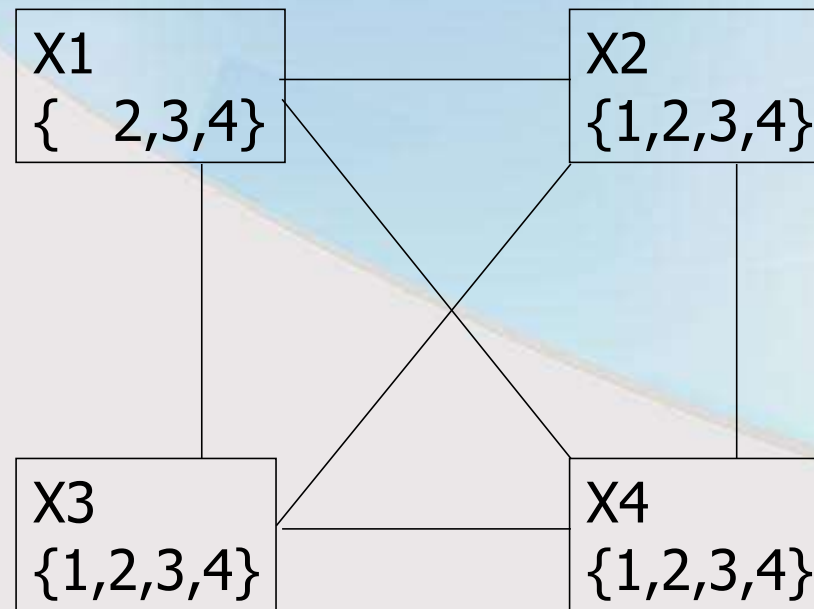
Example: 4-Queens Problem

	1	2	3	4
1	★	●	●	●
2		●	●	
3		★	●	●
4			●	●



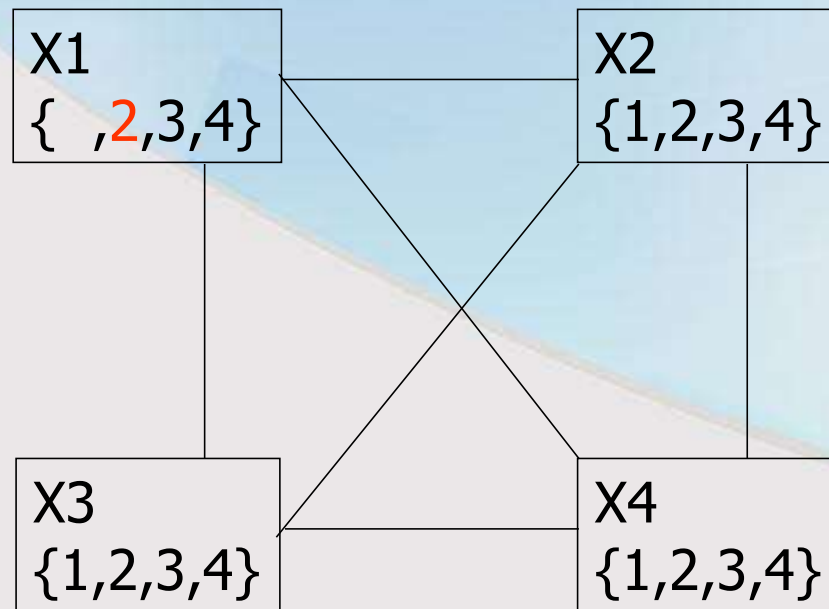
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				



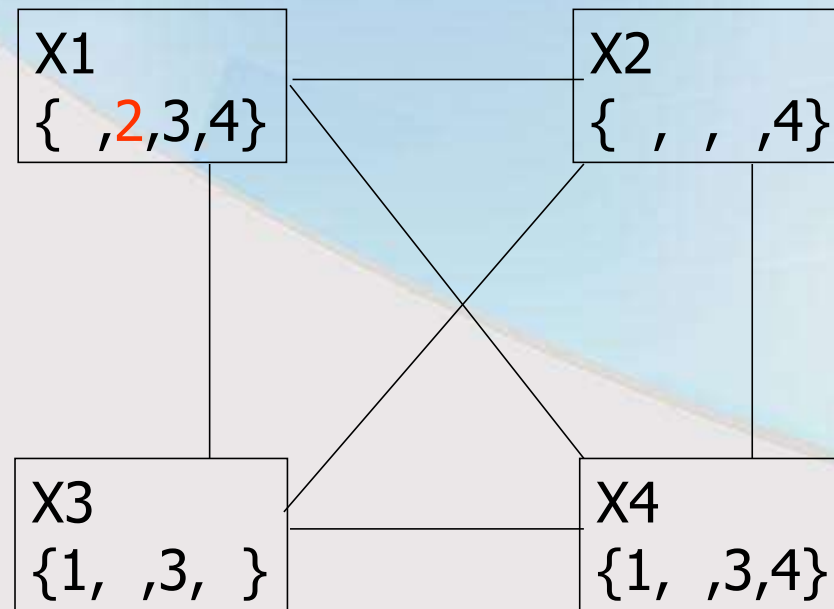
Example: 4-Queens Problem

	1	2	3	4
1		●		
2	★	●	●	●
3		●		
4			●	



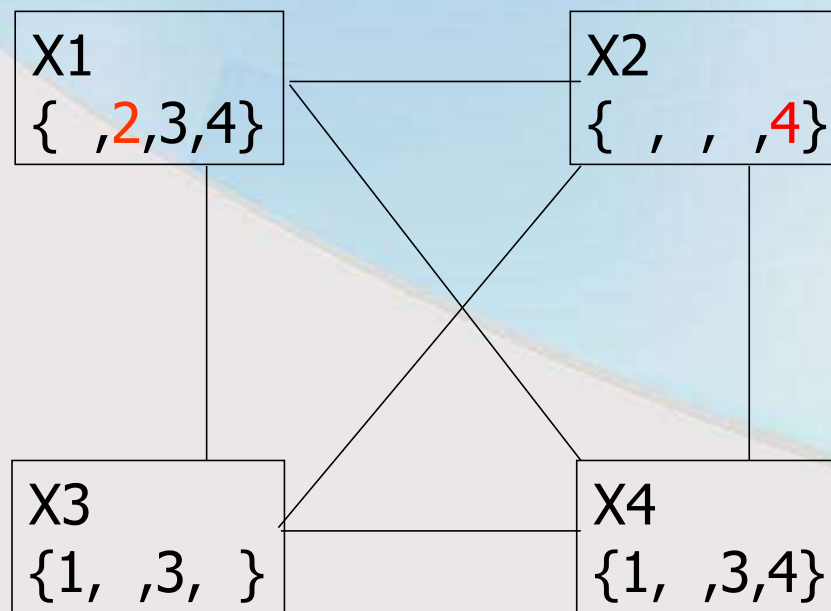
Example: 4-Queens Problem

	1	2	3	4
1		●		
2	★	●	●	●
3		●		
4			●	













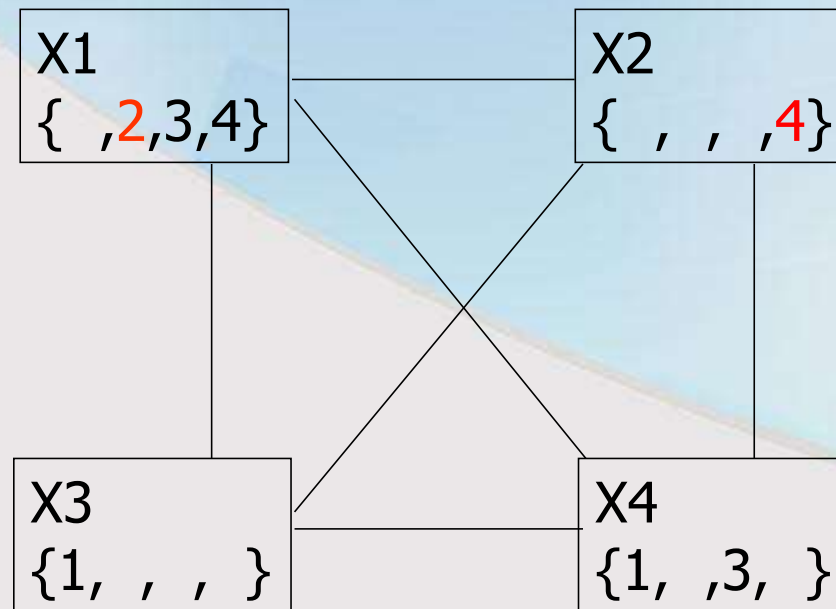
Example: 4-Queens Problem

	1	2	3	4
1		●		
2	★	●	●	●
3		●	●	
4		★	●	●



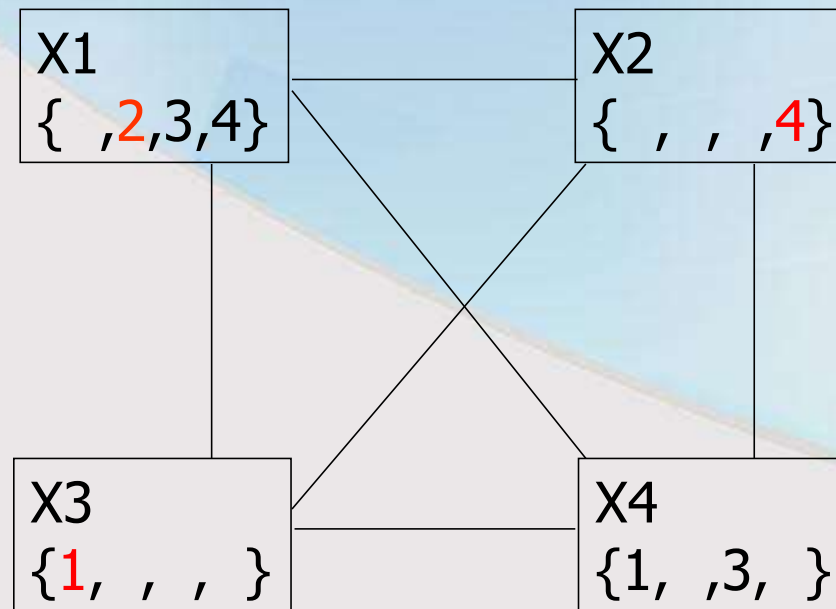
Example: 4-Queens Problem

	1	2	3	4
1				
2				
3				
4				



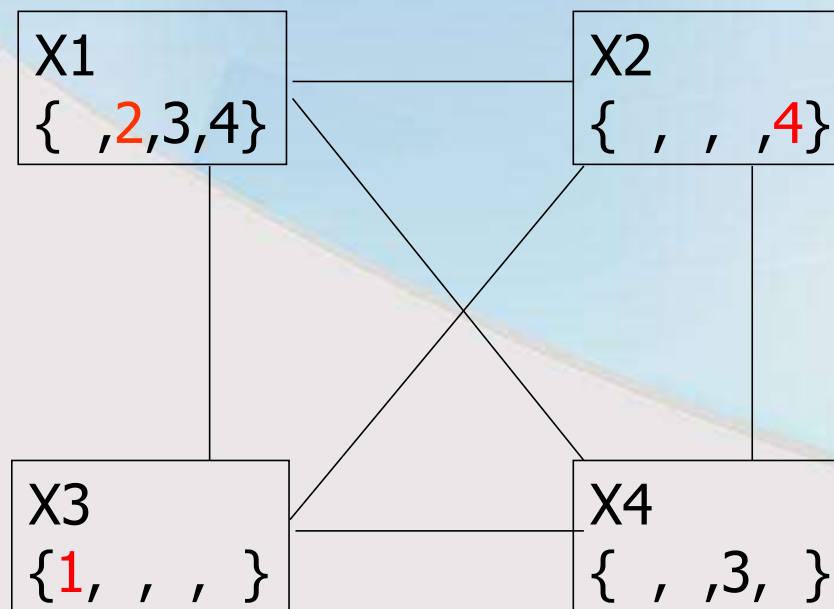
Example: 4-Queens Problem

	1	2	3	4
1		●	★	●
2	★	●	●	●
3		●	●	
4		★	●	●



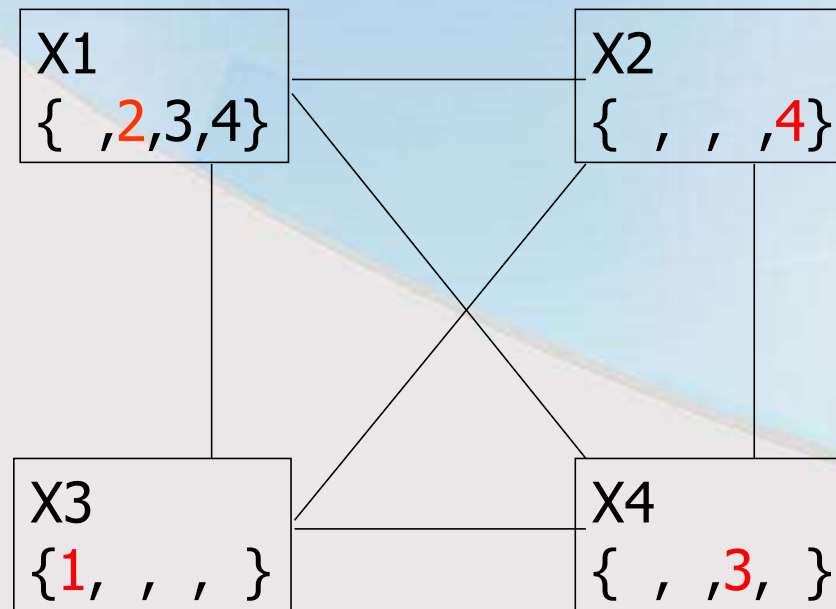
Example: 4-Queens Problem

	1	2	3	4
1		●	★	●
2	★	●	●	●
3		●	●	
4		★	●	●



Example: 4-Queens Problem

	1	2	3	4
1		●	★	●
2	★	●	●	●
3		●	●	★
4		★	●	●



约束传播

- 前向检查把已经赋值变量的信息传播给未赋值变量，但不能提早检测到所有失败。



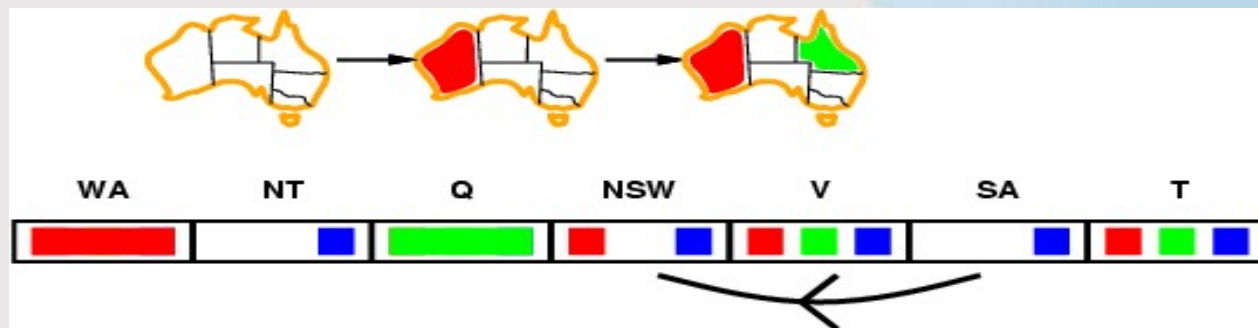
- NT 和 SA 不能同时为蓝色

向前检验不能检测出该类矛盾，因为它看得不够远。

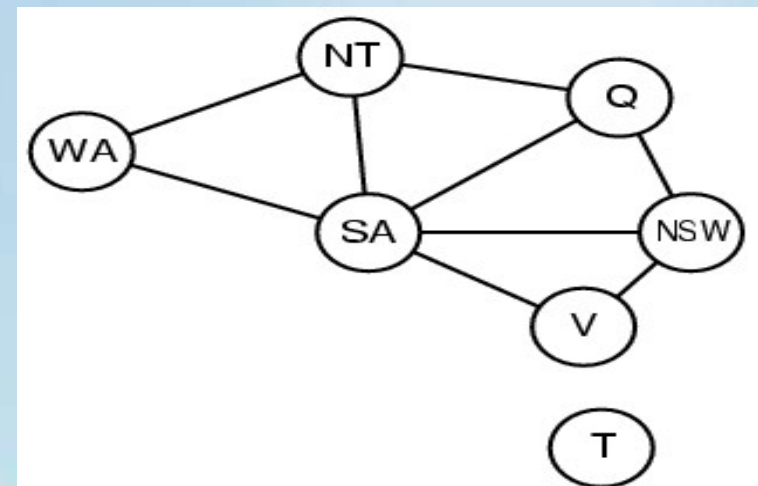
约束传播是将一个变量的约束内容传播到其它变量上的方法。

弧相容(Arc consistency)

- 弧 $x \rightarrow y$ 是相容的当且仅当对于变量 x 的每一种赋值，变量 y 都存在一个合法赋值。

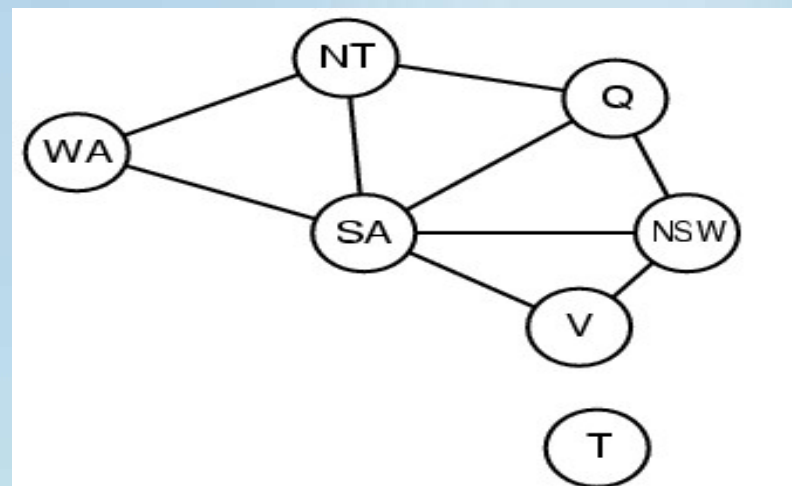
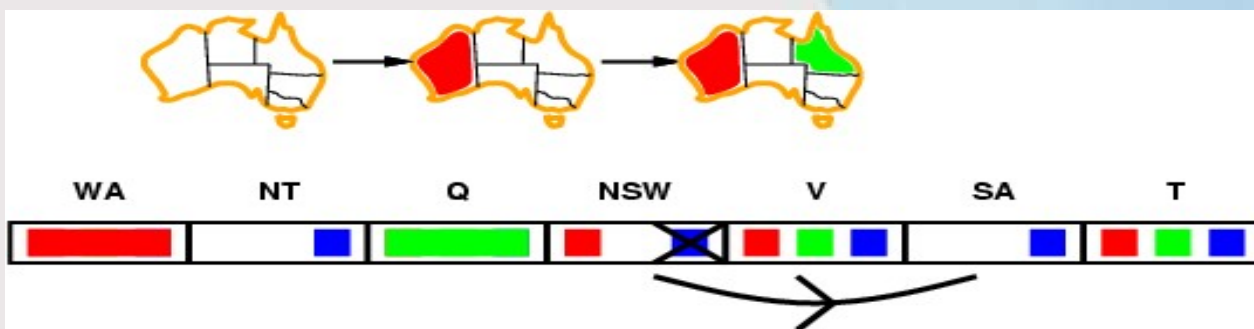


- $SA \rightarrow NSW$ 是相容的当且仅当 $SA=blue$ and $NSW=red$



弧相容(Arc consistency)

- 弧 $X \rightarrow Y$ 是相容的当且仅当对于变量 X 的每一种赋值，变量 Y 都存在一个合法赋值。

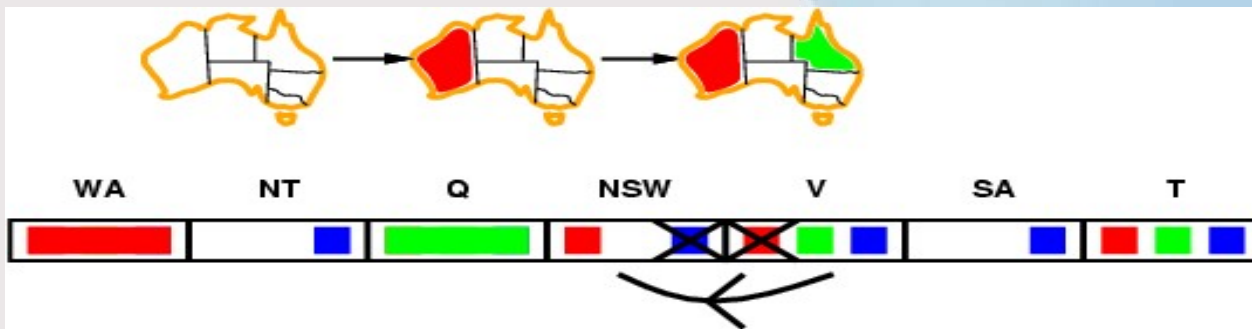


- $NSW \rightarrow SA$ 是相容的当且仅当
 $NSW = red$ and $SA = blue$
 $NSW = blue$ and $SA = ???$

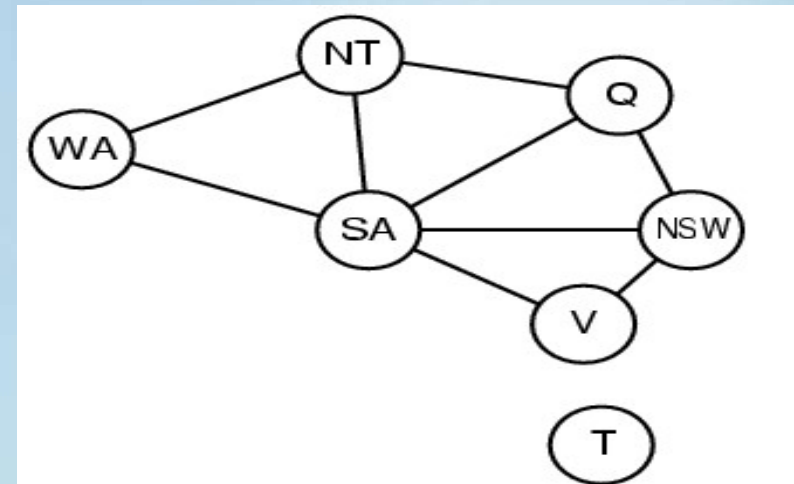
没有对应的值，因此，从 **NSW** 中移出蓝色，使弧相容。

弧相容(Arc consistency)

- 如果变量 x 删除了一个合法赋值，则它的邻居变量需要重新检查

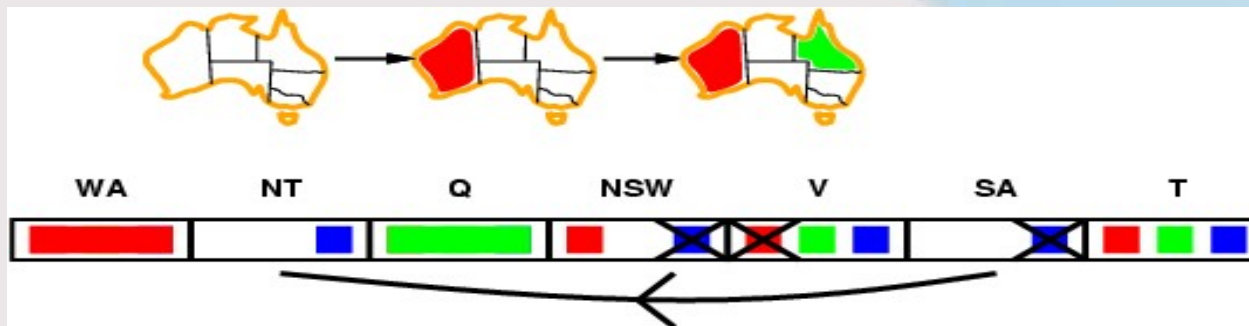
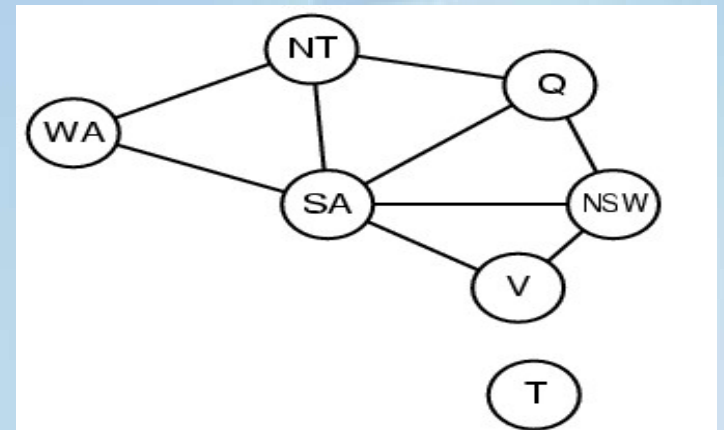


- 由于弧相容的问题从NSW中移除了蓝色
- 检查它所有的邻居!!
 - 把红色从V中移出。



弧相容(Arc consistency)

- 弧相容原则能比前向检查更早检测到失败
- 可以在变量赋值前或后进行弧相容检查
 - ✓ 重复执行直到没有不相容存在



弧相容算法AC-3

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**

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

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

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

 add (X_k, X_i) to *queue*

function RM-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff remove a value

$\textit{removed} \leftarrow \textit{false}$

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

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

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

return *removed*

K-相容算法AC-3

- 弧相容依然不能探测出所有的不相容
- 一种K-相容的概念可以定义更强的约束传播方式
 - 如果对于任何k-1个变量的相容赋值，第k个变量总能被赋予一个和前k-1个变量相容的值，那么这个csp就是一个k相容的。
例如：1-相容是指每个单独的变量，自己是不矛盾的，也称为节点相容。
2-相容就是弧相容。
3-相容是指任何一对相邻的变量总可以扩展到第三个邻居变量，也称为路径相容。
K-相容的只有当它是K-相容，也是(K-1)-相容，(K-2)-相容，...，直到1-相容

- 弧相容算法**AC-3**时间复杂度: $O(n^2d^3)$

- 对这样的K-相容是理想化的（完美的）。

因为，一个解可以在 $O(nd)$ 内找到，而不是 $O(n^2d^3)$ 。

- 但是世界上没有免费的午餐：在最坏的情况下建立任何一个n-相容算法需要花费n的指数级时间。

进一步的改进

- 检查特别的约束（以下2种代表）

Alldif(...)约束：在要求涉及的全部变元都必须取不同的值时

- 如果约束涉及 m 个变元，它们共有 n 个不同的取值，如果有 $m > n$ ，那么这个约束不可满足，要及早停止。

算法思想：

首先，删除约束中只有一个取值的变元；

然后，将这个值也从值域中删除；

循环执行上的步骤，直到得到一个空值域或者剩下的变量数比可取值的个数多，则产生矛盾。

Atmost(...)至多约束：对于每个变量 X 和它的取值的上、下界，每个变量 Y 都存在某个取值满足 X 和 Y 之间的约束，我们称该CSP为边界相容的。

这种边界传播广泛地应用于实际的约束问题中。

局部搜索算法解决CSPs

- 局部搜索应用到约束满足问题CSPs:
 - 采用完全形式化表示（即允许状态不符合约束）
 - 动作定义为：给变量重新赋值（对不满足约束的变量）
- 变量选择：随机选择一个违反约束的变量重新赋值
- 赋值方案采用最少冲突启发式原则，
即选择违反最少约束的值进行赋值操作

局部搜索算法解决CSPs

function MIN-CONFLICTS(*csp*, *max_steps*) **return** solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current \leftarrow an initial complete assignment for *csp*

for *i* = 1 to *max_steps* **do**

if *current* is a solution for *csp* then **return** *current*

var \leftarrow a randomly chosen, conflicted variable from VARIABLES[*csp*]

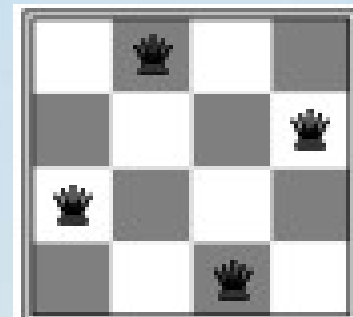
value \leftarrow the value *v* for *var* that minimize CONFLICTS(*var*, *v*, *current*, *csp*)

set *var* = *value* in *current*

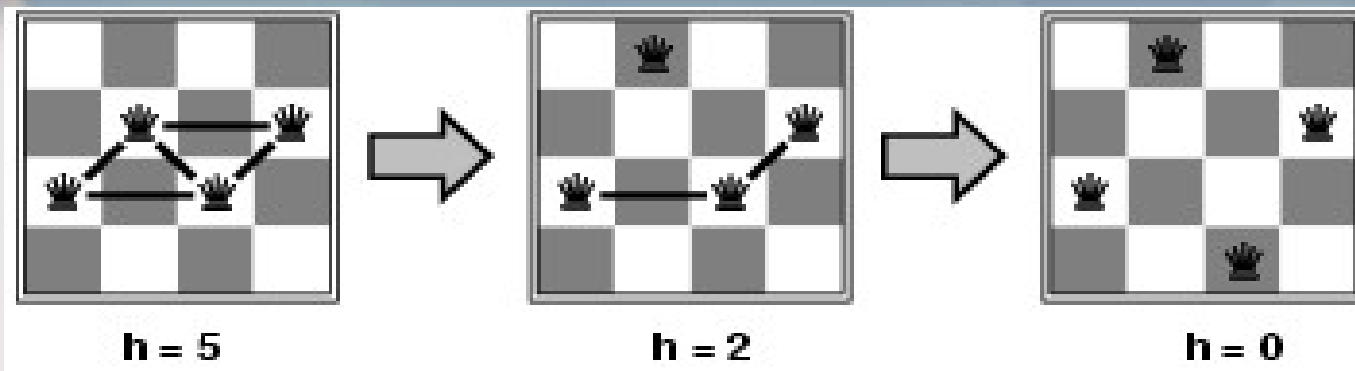
return *failure*

例子: 4皇后问题

- 状态: 每列摆放一个皇后 ($4^4 = 256$ 个不同状态)
- 动作: 将皇后在同一列内移动
- 目标测试: 互相攻击不到
- 评估函数: $h(n) =$ 攻击到的皇后对数



例子: 4皇后问题



- 任意给定一个初始状态，局部搜索算法能以很高的概率在常数时间内解决大规模 n 皇后问题 (e.g., $n = 10,000,000$)

效果：最小冲突法几乎是独立于问题的规模的。

可以在大约50步以内解决百万皇后的问题。

局部搜索可以应用于联机问题的求解，因为回溯需要更多的时间。

总结

- CSP是一类特别的问题：状态是通过对变量集的赋值，目标测试是通过变量值的约束定义。
- 回溯法=采用深度优先每次给一个节点赋值。
- 用通用的启发式进行变量的选择和值的选择是非常有效。
- 最小剩余值MRV：选择赋值变量是优先选取取值数量较少的变量；
- 取值数相同时，采用度启发：受限制多（约束较多）的变量优先；
- 选择为那个变元赋值后，选择对变量约束最少的值进行赋值，以便于给余下的变量有更大的赋值空间。
- 向前检测可以阻止导致失败。
- 附加的约束传播工作可以提前检测出不相容值。
- 循环的最小冲突法在实际中是非常有效的。

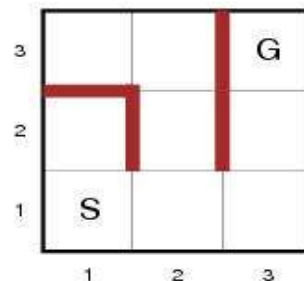


扩展：联机（Online）搜索问题

- 到如今为止，我们探讨的都是非实时的，或称为脱机问题。
 - 脱机问题（offline）= 它们在执行前就可以计算出解决方案
 - 联机问题（online）= 它们在执行时，需要计算和行动交叉进行
 - 联机搜索对于动态和半动态环境的搜索问题是必须的；
 - 现实，在搜索过程中考虑所有的可能的偶发事情是不可能的。
- 常见的探究问题：
 - 智能体在未知的状态以及行为。
 - e.g. 一个新生儿，
 - 机器人在一个新环境, ...

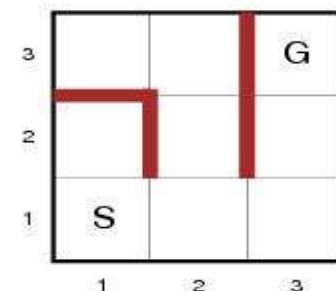
联机搜索

- 假设智能体有以下的知识:
 - **ACTION(s)**: 状态s下可能的行动列表函数
 - **C(s,a,s')**: 单步耗散（直到智能体s知道行动的结果s' 时才能得到！）
 - **GOAL-TEST(s)**: 目标测试
- 事实：智能体能够记住先前的状态，不能访问下个状态。
- 行动是确定的。
- 可以使用可采纳的**启发函数** $h(s)$
e.g. 曼哈顿距离



联机搜索

- 目标：用最小耗散达到目标
- 耗散：整个路径的总耗散
 - ✓ **竞争率**=实际代价同如果已知搜索空间的解路径代价的**比值**。
 - ✓ 当智能体偶然达到死路径可能造成无限循环。
- 一个联机智能体具有维护环境地图能力
 - ✓ 随着输入的感知更新地图
 - ✓ 并应用它确定下一个行动
- 同A*算法的区别
 - ✓ 一个联机版的智能体只能扩展它在当前状态中的实际结点（物理结点）。



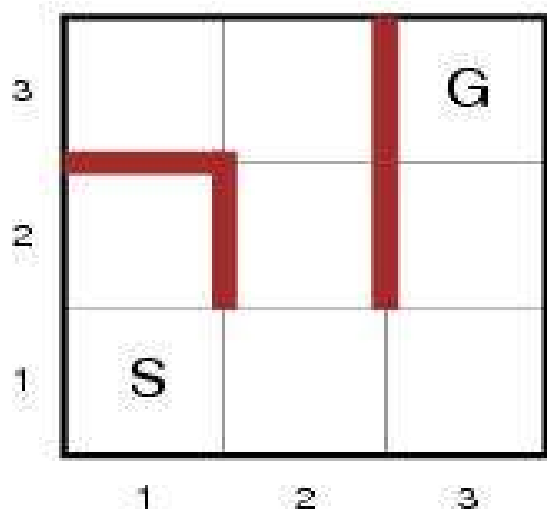
```
function ONLINE_DFS-AGENT( $s'$ ) return an action //联机深度优先搜索

input:  $s'$ , a percept identifying current state
static: result, a table indexed by action and state, initially empty
       unexplored (UX) , a table that lists for each visited state, the action not yet tried
       unbacktracked (UB) , a table that lists for each visited state, the backtrack not yet tried
        $s, a$ , the previous state and action (of  $s'$ ), initially null

if GOAL-TEST( $s'$ ) then return stop
if  $s'$  is a new state then unexplored[ $s'$ ]  $\leftarrow$  ACTIONS( $s'$ )
if  $s$  is not null then do
    result[ $a, s$ ]  $\leftarrow s'$ 
    add  $s$  to the front of unbacktracked[ $s'$ ]
if unexplored[ $s'$ ] is empty then
    if unbacktracked[ $s'$ ] is empty then return stop
    else  $a \leftarrow$  an action  $b$  such that result[ $b, s'$ ] = POP(unbacktracked[ $s'$ ])
else  $a \leftarrow$  POP(unexplored[ $s'$ ])
 $s \leftarrow s'$ 

return  $a$ 
```

联机搜索例子



迷宫问题: 3x3 grid.

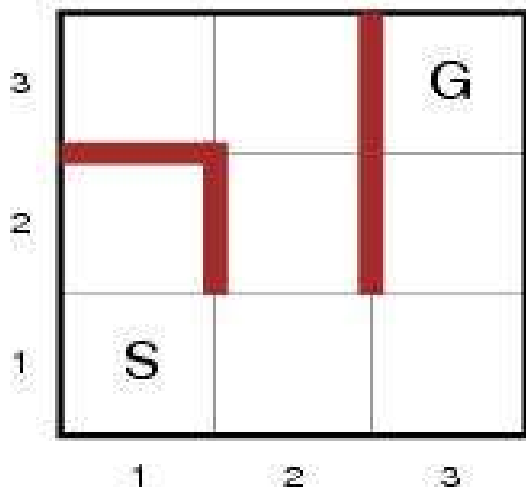
$s' = (1,1)$ is **initial state**
result,

unexplored (UX),
unbacktracked (UB), ...

are empty

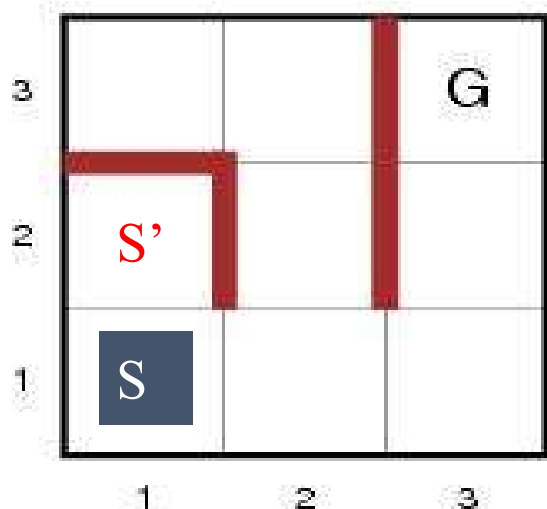
s, a are also **empty**

联机搜索例子



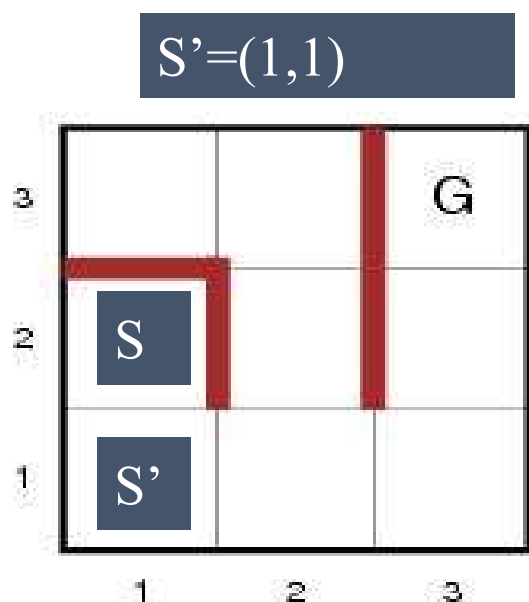
- **GOAL-TEST((1,1))?**
 - $S' \text{ not } = G$ thus false
- **(1,1) a new state?**
 - true
 - **ACTION((1,1)) \rightarrow UX[(1,1)]**
 - **{RIGHT,UP}**
- **s is null?**
 - true (initially)
- **UX[(1,1)] empty?**
 - false
- **POP(UX[(1,1)]) \rightarrow a**
 - **a=UP**
- **s = (1,1)**
- **Return a**

联机搜索例子



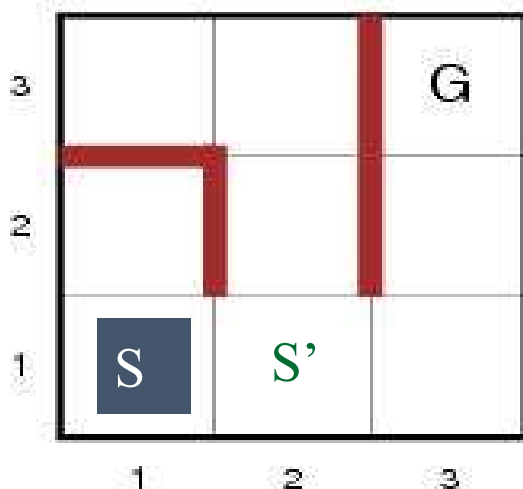
- **GOAL-TEST((2,1))?**
 - $S' \neq G$ thus false
- **(2,1) a new state?**
 - true
 - **ACTION((2,1)) \rightarrow UX[(2,1)]**
 - **{DOWN}**
- **s is null?**
 - false ($s=(1,1)$)
 - **result[UP,(1,1)] \leftarrow (2,1)**
 - **UB[(2,1)] = {(1,1)}**
- **UX[(2,1)] empty?**
 - false
- **a=DOWN, $s=(2,1)$**
- **return a**

联机搜索例子



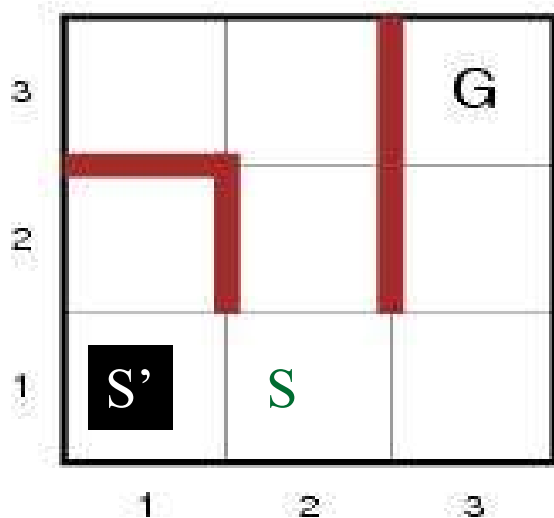
- **GOAL-TEST((1,1))?**
 - $S \text{ not} = G$ thus false
- **(1,1) a new state?**
 - false
- **s is null?**
 - false ($s=(2,1)$)
 - $\text{result}[\text{DOWN},(2,1)] \leftarrow (1,1)$
 - $\text{UB}[(1,1)] = \{(2,1)\}$
- **UX[(1,1)] empty?**
 - false //{RIGHT,UP}
- **a=RIGHT, $s=(1,1)$**
return a

联机搜索例子



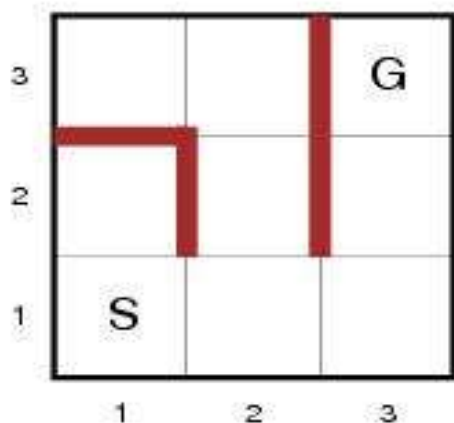
- GOAL-TEST((1,2))?
 - $S' \neq G$ thus false
- (1,2) a new state?
 - True, $UX[(1,2)] = \{\text{RIGHT}, \text{UP}, \text{LEFT}\}$
- s is null?
 - false ($s=(1,1)$)
 - $\text{result}[\text{RIGHT}, (1,1)] \leftarrow (1,2)$
 - $UB[(1,2)] = \{(1,1)\}$
- $UX[(1,2)]$ empty?
 - False
- $a = \text{LEFT}$, $s=(1,2)$ return **a**

联机搜索例子



- GOAL-TEST((1,1))?
 - S not = G thus false
- (1,1) a new state?
 - false
- s is null?
 - false (s=(1,2))
 - result[LEFT,(1,2)] <- (1,1)
 - UB[(1,1)]={ (2,1), (1,2) }
- UX[(1,1)] empty?
 - true
 - UB[(1,1)] empty? False
- a= **b** for b in result[b,(1,1)]=(1,2)
 - **b=RIGHT**
- a=RIGHT, s=(1,1) ...

联机搜索例子

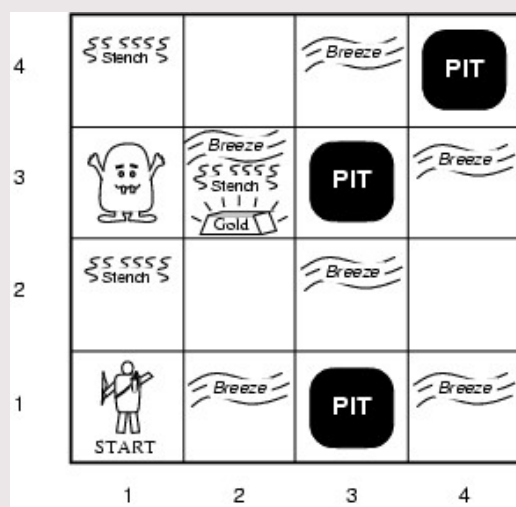


算法分析:

- 最坏的情形是每个节点被访问2次.
- 即使某节点已经很接近目标它还需走很远.
- 一个联机迭代加深的方法可以解决这个问题.
- 联机深度优先只有当行为是可逆的才能进行.

题目（1必做，2、3、4、5选一）

- 1、必做：N皇后问题（分别采用回溯法、遗传算法、模拟退火、爬山法
- 2、五子棋游戏
- 3、推箱子
- 4、扫雷游戏
- 5、Wumpus怪兽世界



Wumpus World PEAS 描述:

性能度量:

gold +1000, death -1000

-1 per step, -10 for using the arrow

环境描述:

Squares adjacent to wumpus are smelly

Squares adjacent to pit are breezy

Glitter iff gold is in the same square

Shooting kills wumpus if you are facing it

Shooting uses up the only arrow

Grabbing picks up gold if in same square

Releasing drops the gold in same square

传感器: Stench, Breeze, Glitter, Bump, Scream

执行器:

Left turn, Right turn, Forward, Grab, Release, Shoot