

PROBLEMAS DE SATISFAÇÃO DE RESTRIÇÕES

ADAPTADO DE [HTTP://AIMA.EECS.BERKELEY.EDU/SLIDES-TEX/](http://aima.eecs.berkeley.edu/slides-tex/)

Resumo

- ◇ Exemplos de CSP (Constraint Satisfaction Problems)
- ◇ Procura com retrocesso (backtracking) para CSPs
- ◇ Estrutura e decomposição de problemas
- ◇ Procura local para CSPs

Problemas de Satisfação de Restrições (CSPs)

Problema usual de procura:

o **estado** é uma “caixa negra” — estrutura de dados arbitrária suportando métodos para teste objectivo, função de avaliações e sucessor

CSP:

estado é definido com *variáveis* X_i que tomam *valores* num *domínio* D_i

teste objectivo é um conjunto de *restrições* especificando as combinações permitidas para subconjuntos de variáveis

Exemplo simples de uma *linguagem de representação formal*

Permite recorrer a algoritmos *genéricos* melhores do que os algoritmos de procura estudados anteriormente

Exemplo: Coloração de Mapas



Exemplo: Coloração de Mapas



Variáveis WA, NT, Q, NSW, V, SA, T

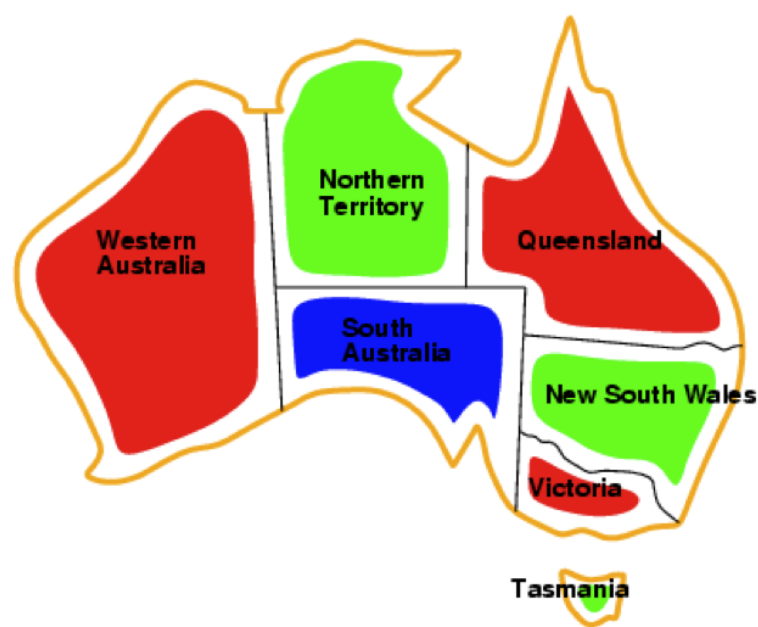
Domínios $D_i = \{red, green, blue\}$

Restrições: regiões adjacentes devem ter cores diferentes

e.g., $WA \neq NT$ (se a linguagem o permitir), ou

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

Exemplo: Coloração de Mapas (cont.)



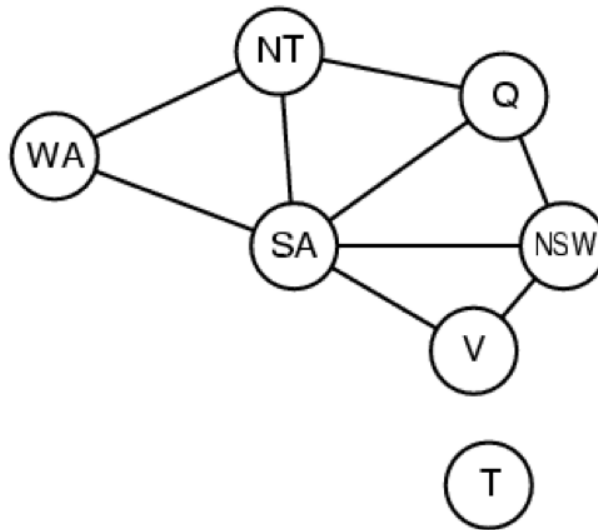
Soluções são atribuições de valores a variáveis que satisfazem todas as restrições, e.g.,

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

Grafo de restrições

CSP binário: cada restrição relaciona no máximo duas variáveis

Grafo de restrições: nós são variáveis e arcos correspondem a restrições



Algoritmos genéricos para CSPs fazem uso da estrutura do grafo para tornar a procura mais eficiente. E.g., Tasmânia é um subproblema independente!

Tipos de CSPs

Variáveis discretas

domínios finitos; cardinalidade $d \Rightarrow O(d^n)$ atribuições completas

◇ e.g., CSPs Booleanos, incl. satisfatibilidade Booleana (NP-completo)

domínios infinitos (inteiros, cadeias de caracteres, etc.)

◇ escalonamento: variáveis representam início/fim das tarefas

◇ utilizam **linguagem de restrições**: $StartJob_1 + 5 \leq StartJob_3$

◇ restrições **lineares** são solúveis, **não-lineares** indecidíveis

Variáveis contínuas

◇ e.g., tempos de início/fim das observações do Telescópio Hubble

◇ restrições lineares resolúveis em tempo polinomial por métodos de programação linear

Tipos de restrições

Unárias restrições envolvendo apenas uma variável,

e.g., $SA \neq green$

Binárias restrições envolvendo pares de variáveis,

e.g., $SA \neq WA$

Ordem superior restrições envolvendo 3 ou mais variáveis,

e.g., restrições das colunas em problemas cripto-aritméticos

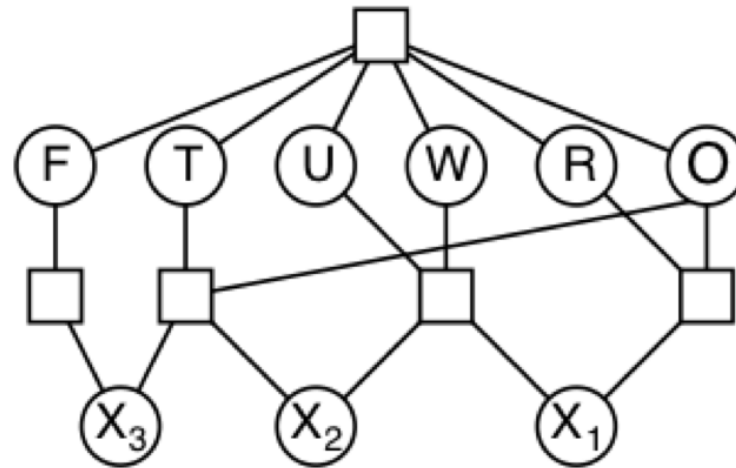
Preferências (restrições suaves), e.g., *red* é melhor do que *green*

habitualmente representado atribuindo um custo a cada atribuição de variáveis

→ problemas de otimização com restrições

Exemplo: Cripto-aritmético

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Variáveis: $F T U W R O X_1 X_2 X_3$

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

Restrições

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

$O + O = R + 10 \cdot X_1$, etc.

CSPs reais

Problemas de alocação

e.g., quem é o professor de determinada disciplina ?

Problemas de horários

e.g., quando é que uma disciplina é oferecida e onde?

Configuração de Hardware

Folhas de cálculo

Escalonamento de Transportes

Planeamento/Escalonamento de Produção

Planeamento Espacial

Notar que muitos problemas reais requerem variáveis contínuas

Formulação como Problema de Procura

Os estados são definidos pelos valores atribuídos até ao momento às variáveis

- ◇ **Estado inicial:** a atribuição vazia, $\{ \}$
 - ◇ **Função sucessor:** atribuir um valor a uma variável livre que não entre em conflito com a atribuição actual.
⇒ falha se não existirem atribuições possíveis (irreparável!)
 - ◇ **Teste objectivo:** a atribuição não tem variáveis livres
- 1) É o mesmo para todos os CSPs!
 - 2) Toda a solução ocorre à profundidade n com n variáveis
⇒ utilização de procura em profundidade primeiro
 - 3) Caminho é irrelevante, pode-se usar formulação de estado completo
 - 4) $b = (n - \ell)d$ à profundidade ℓ , logo $n!d^n$ folhas!!!!

Procura com Retrocesso

As atribuições de variáveis são **comutativas**, i.e.,

$[WA = red \text{ depois } NT = green]$ mesmo que $[NT = green \text{ depois } WA = red]$

Só é preciso considerar atribuições a uma única variável em cada nó

$\Rightarrow b = d$ e existem d^n folhas

Procura em profundidade primeiro para CSPs com atribuições a uma única variável é designada por procura com **retrocesso**

Procura com retrocesso é o algoritmo básico cego para CSPs

Consegue resolver n -rainhas para $n \approx 25$

Procura com retrocesso

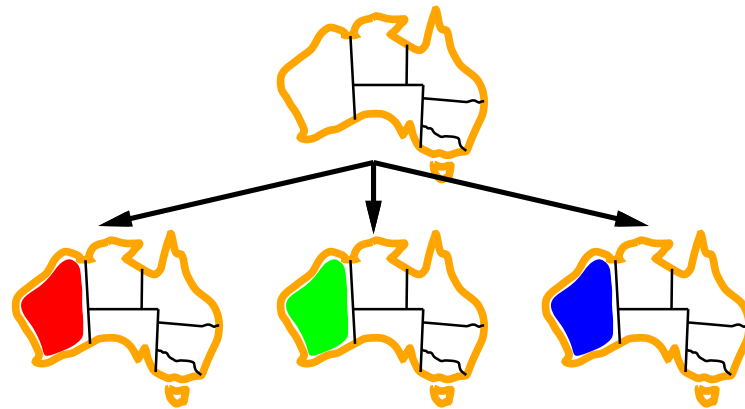
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING([], csp)

function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
  if assigned is complete then return assigned
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
    if value is consistent with assigned according to CONSTRAINTS[csp] then
      result ← RECURSIVE-BACKTRACKING([var = value | assigned], csp)
      if result ≠ failure then return result
  end
  return failure
```

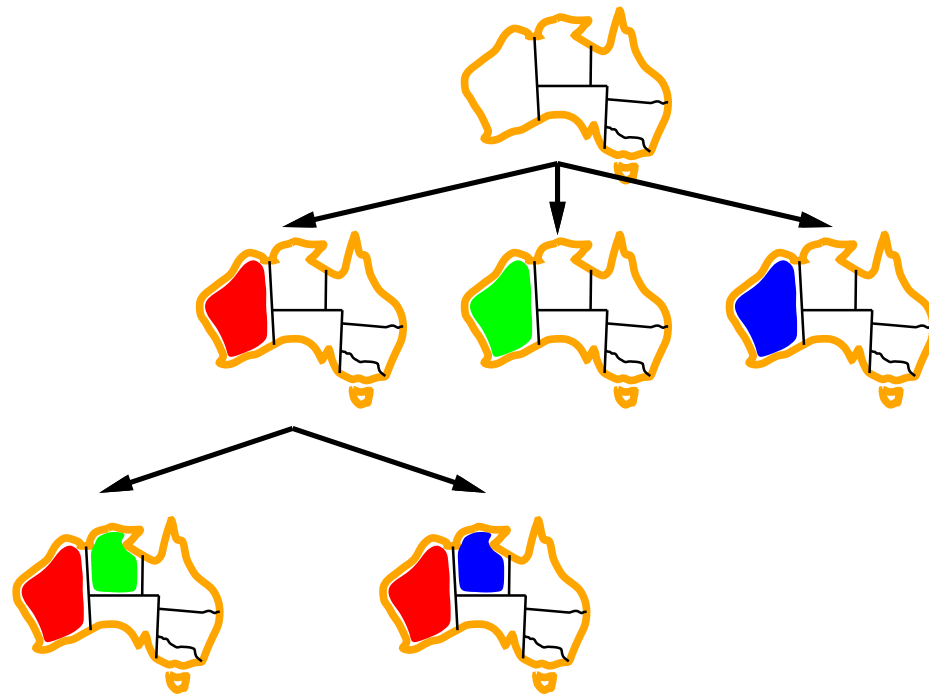
Exemplo de procura com retrocesso



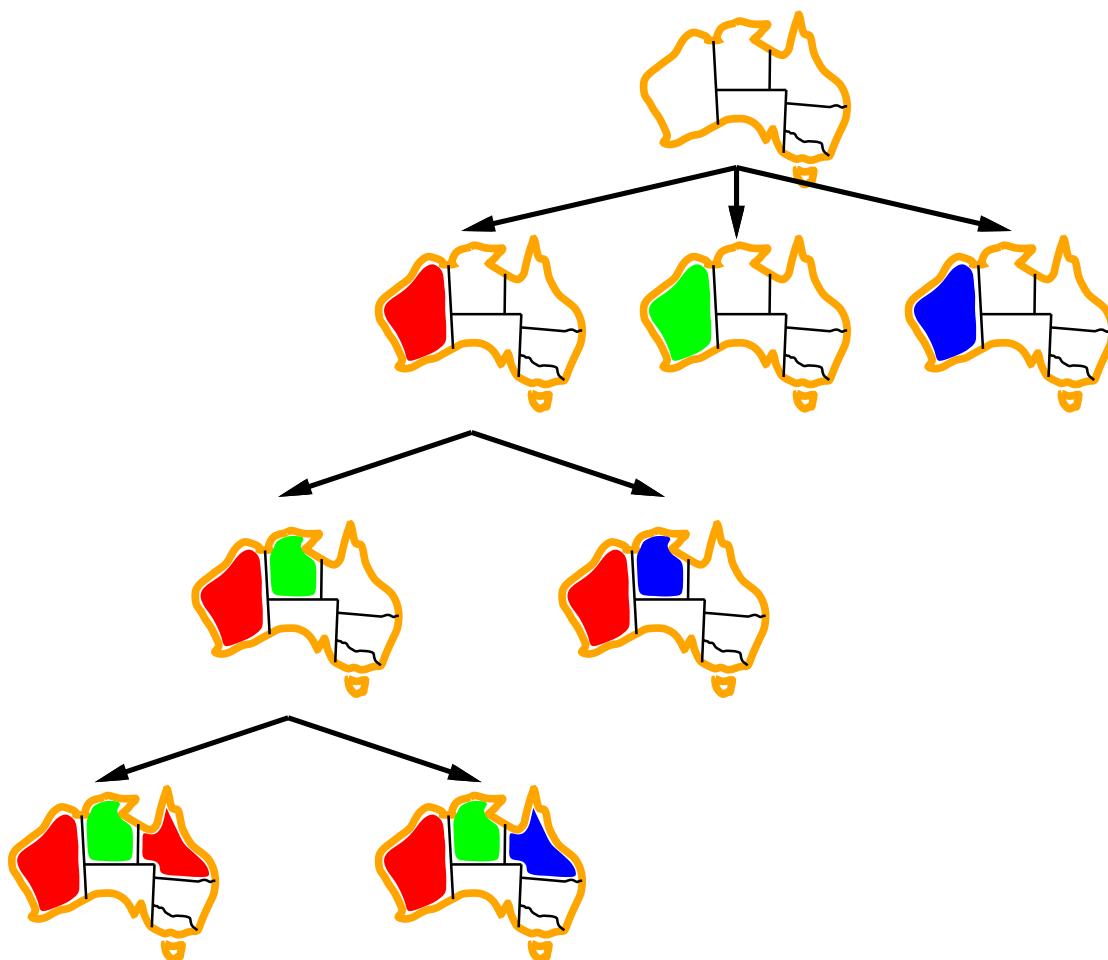
Exemplo de procura com retrocesso



Exemplo de procura com retrocesso



Exemplo de procura com retrocesso



Melhorando eficiência do Retrocesso

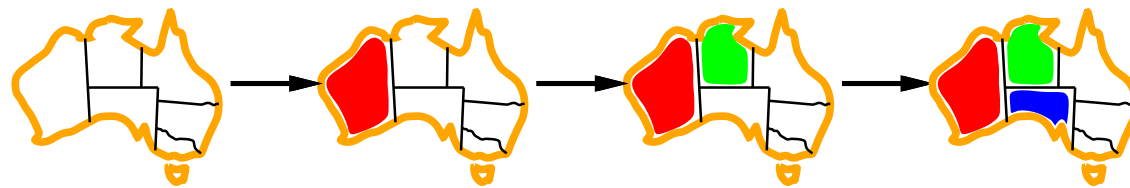
Métodos *genéricos* podem resultar em ganhos substanciais de eficiência:

1. Qual a variável a atribuir?
2. Por que ordem devem ser tentados os seus valores?
3. Podem-se detectar falhas inevitáveis mais cedo?
4. Pode-se utilizar vantajosamente a estrutura do problema?

Variável mais constrangida

Variável mais constrangida:

seleccionar a variável com menos valores possíveis

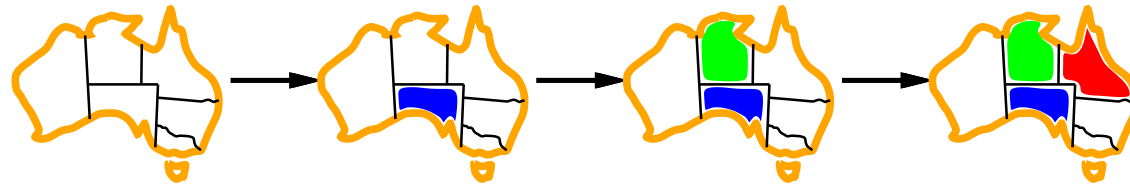


Variável mais constrangida

Desempate entre variáveis mais constrangidas

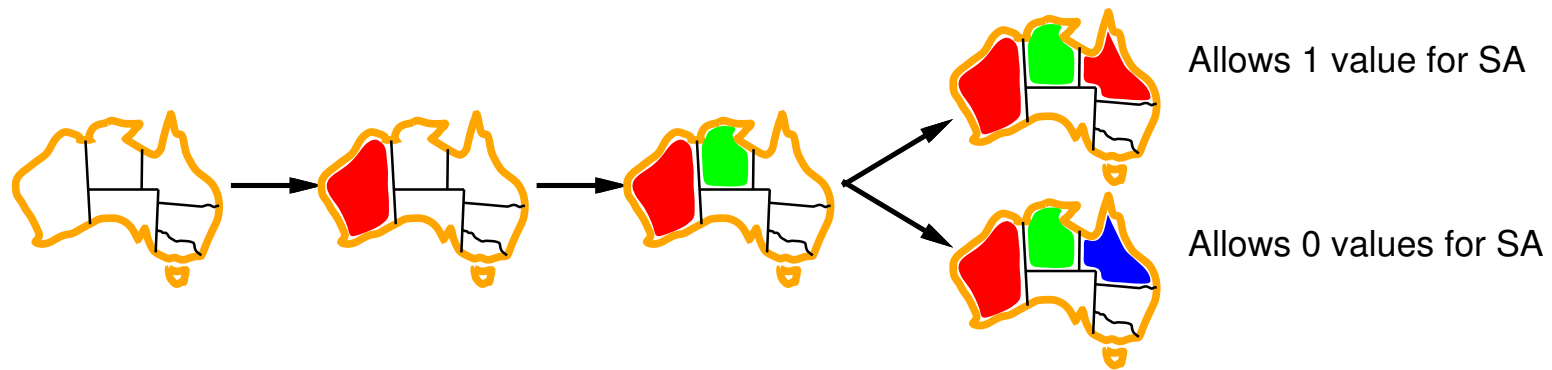
De entre as variáveis mais constrangidas:

escolher a variável com maior número de restrições nas restantes variáveis



Valor menos restritivo

Dado uma variável, escolher o valor menos restritivo:
aquele que eliminar menos valores nas restantes variáveis

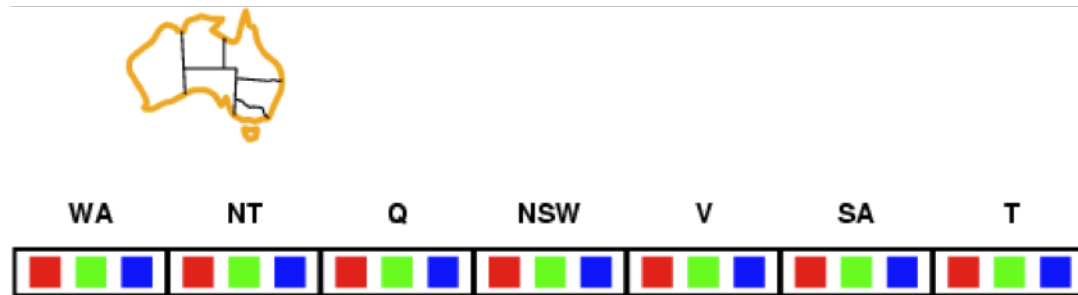


Importante quando só estamos interessados **numa** solução. Irrelevante para quando pretendemos obter todas as soluções, ou quando o problema não tem soluções.

A combinação destas heurísticas permite a resolução de problemas com 1000 rainhas.

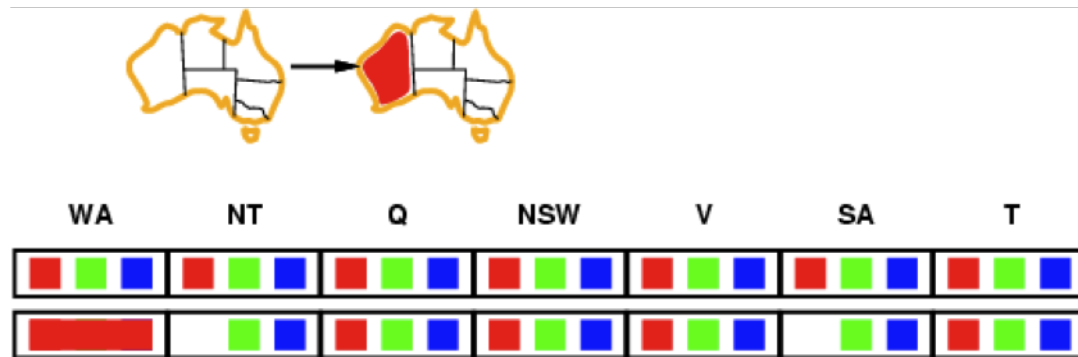
Verificação para a frente

Ideia: Vigiar os valores possíveis das variáveis por atribuir
Termina a procura quando uma variável não possui valores possíveis



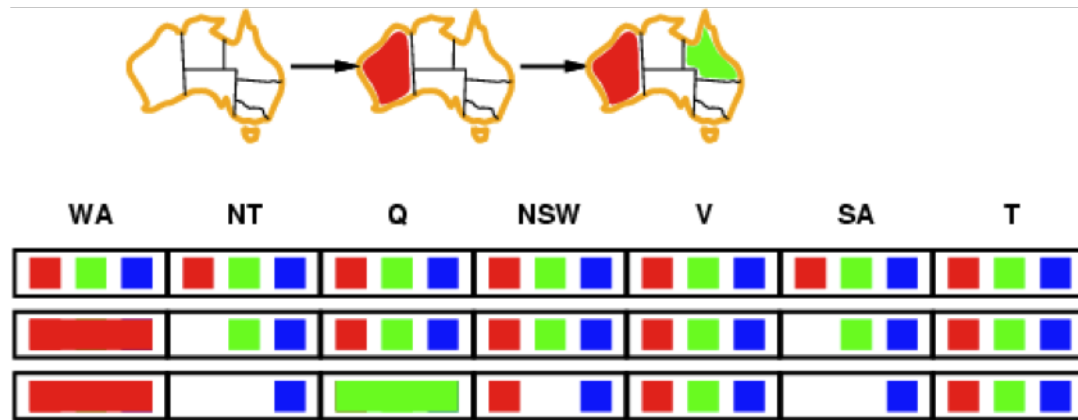
Verificação para a frente

Ideia: Vigiar os valores possíveis das variáveis por atribuir
Termina a procura quando uma variável não possui valores possíveis



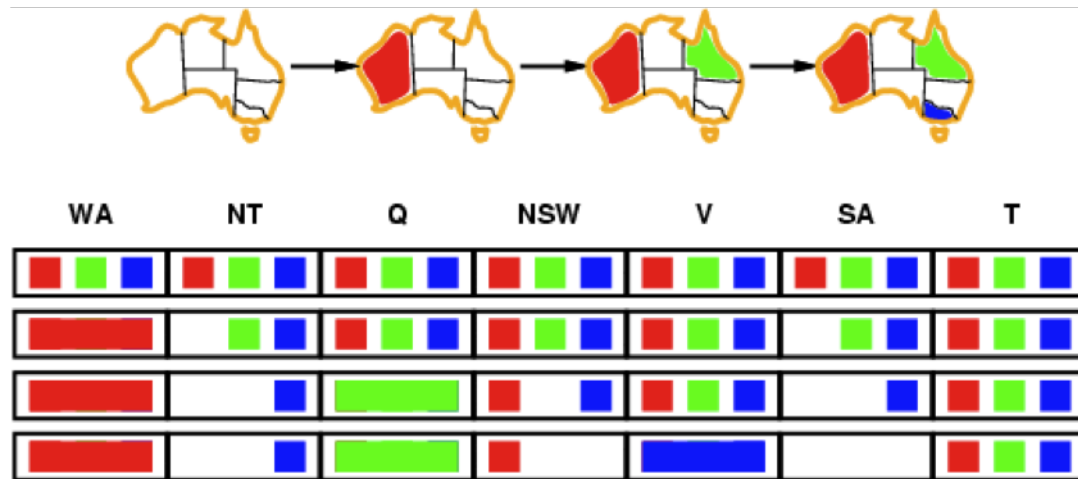
Verificação para a frente

Ideia: Vigiar os valores possíveis das variáveis por atribuir
Termina a procura quando uma variável não possui valores possíveis



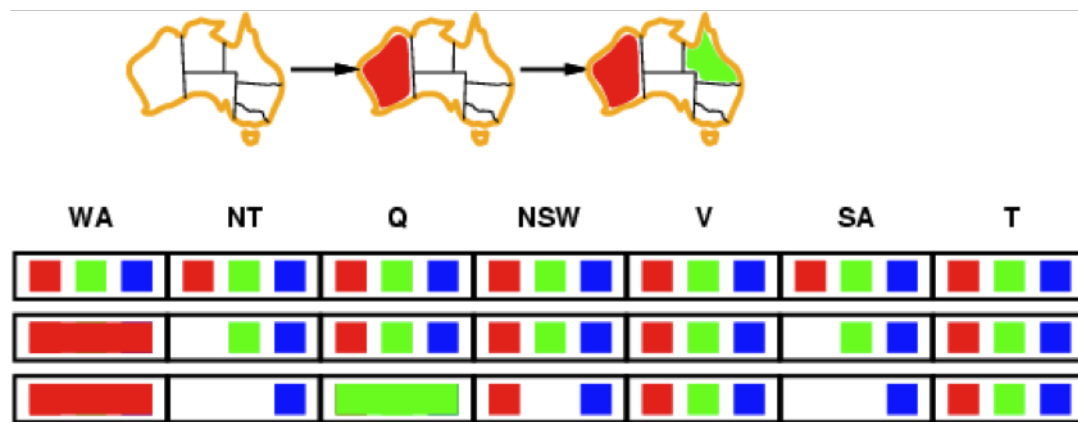
Verificação para a frente

Ideia: Vigiar os valores possíveis das variáveis por atribuir
Termina a procura quando uma variável não possui valores possíveis



Propagação de Restrições

A verificação para a frente propaga informação de variáveis atribuídas para variáveis não atribuídas, mas não permite a detecção prematura de todas as falhas:



NT e *SA* não podem ser ambos azuis!

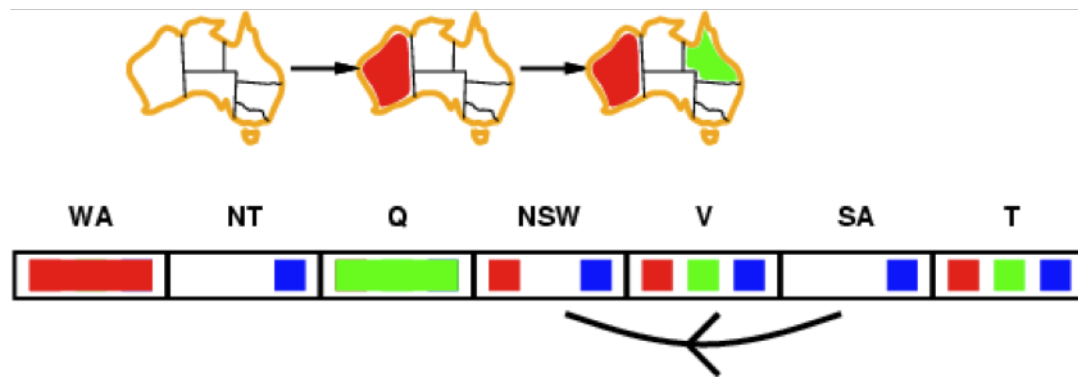
Propagação de restrições obriga repetidamente à satisfação local de restrições

Consistências de arcos

A forma mais simples de propagação torna cada arco **consistente**

$X \rightarrow Y$ é consistente sse

para *todo* o valor x de X então *existe* um valor permitido para y

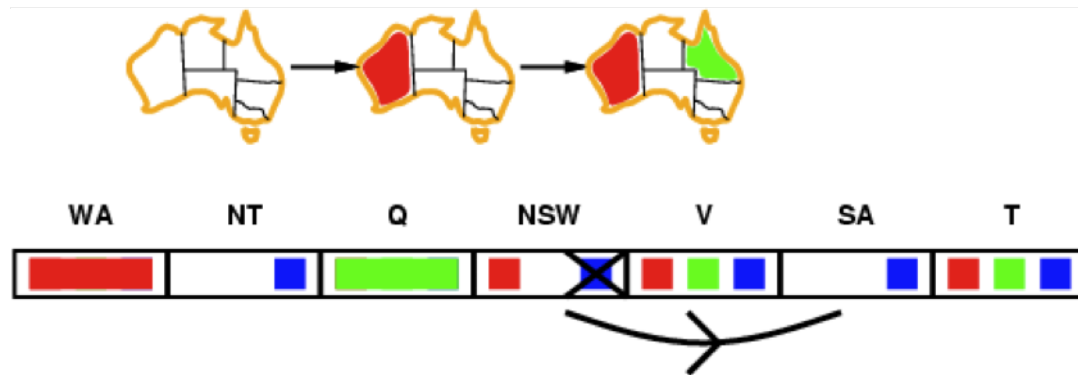


Consistências de arcos

A forma mais simples de propagação torna cada arco **consistente**

$X \rightarrow Y$ é consistente sse

para *todo* o valor x de X então *existe* um valor permitido para y

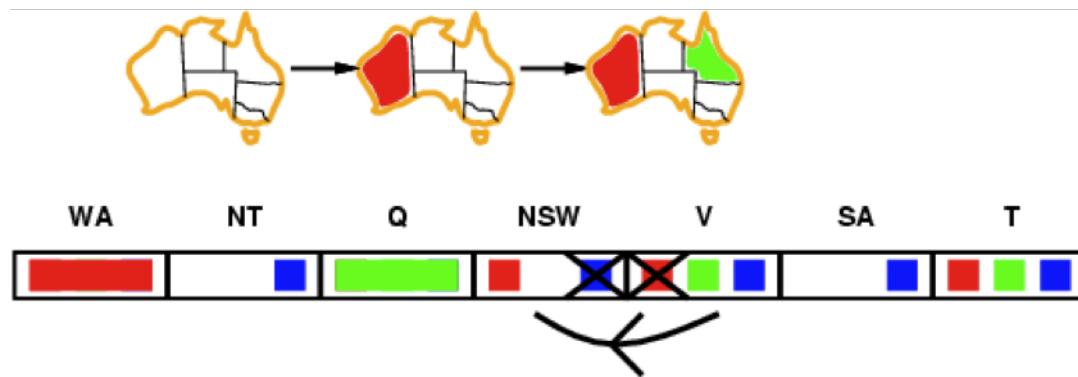


Consistências de arcos

A forma mais simples de propagação torna cada arco **consistente**

$X \rightarrow Y$ é consistente sse

para *todo* o valor x de X então *existe* um valor permitido para y



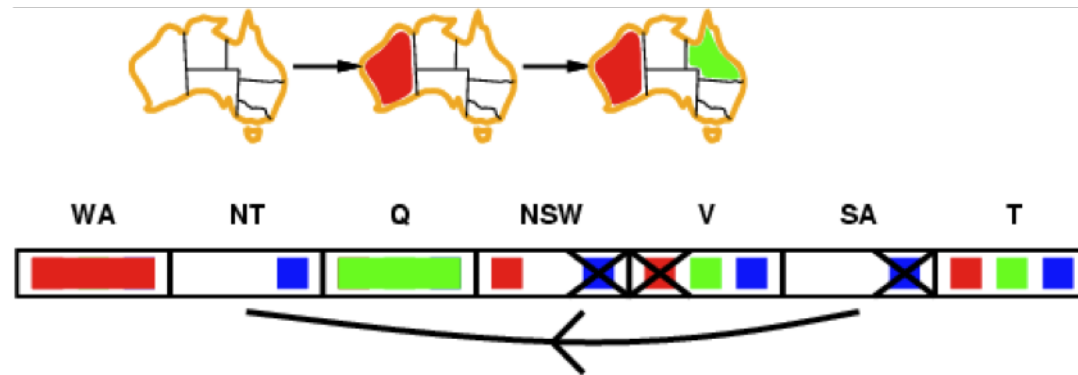
Se X perde um valor, vizinhos de X precisam de ser reverificados

Consistências de arcos

A forma mais simples de propagação torna cada arco **consistente**

$X \rightarrow Y$ é consistente sse

para *todo* o valor x de X então *existe* um valor permitido para y



Se X perde um valor, vizinhos de X precisam de ser reverificados

A consistência de arcos detecta falhas mais cedo do que a verificação para a frente

Pode ser utilizado como preprocessamento ou após cada atribuição

Algoritmo de Consistência de Arcos

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 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 $X_i \leftrightarrow X_j$

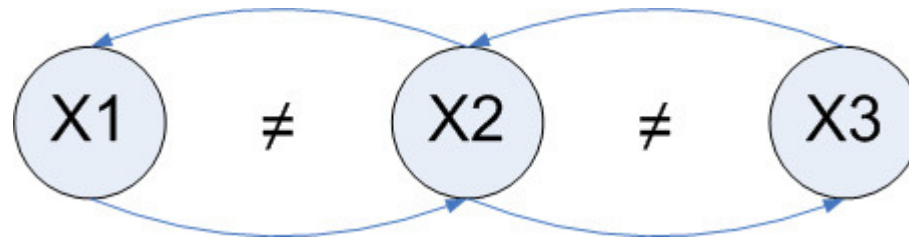
then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ true

return *removed*

$O(n^2d^3)$, pode ser reduzido a $O(n^2d^2)$

mas não consegue detectar todas as falhas em tempo polinomial!

Exemplo de utilização do AC-3



X_1	X_2	X_3	Queue
{1}	{1, 2}	{1, 2}	$X_2 \rightarrow X_3, X_3 \rightarrow X_2, X_1 \rightarrow X_2, X_2 \rightarrow X_1$
{1}	{1, 2}	{1, 2}	$X_3 \rightarrow X_2, X_1 \rightarrow X_2, X_2 \rightarrow X_1$
{1}	{1, 2}	{1, 2}	$X_1 \rightarrow X_2, X_2 \rightarrow X_1$
{1}	{1, 2}	{1, 2}	$X_2 \rightarrow X_1$
{1}	{2}	{1, 2}	$X_1 \rightarrow X_2, X_3 \rightarrow X_2$
{1}	{2}	{1, 2}	$X_3 \rightarrow X_2$
{1}	{2}	{1}	$X_2 \rightarrow X_3$
{1}	{2}	{1}	

Outras formas de consistência

- ◇ **Consistência de nó (ou consistência-1)**: todas as restrições unárias são verificadas.
- ◇ **Consistência de arco (ou consistência-2)**: visto anteriormente.
- ◇ **Consistência de caminho (ou consistência-3)**: um conjunto de duas variáveis $\{X_i, X_j\}$ path-consistent com uma terceira variável X_k sse para qualquer atribuição consistente de valores a X_i e X_j existe um valor possível para X_k .
- ◇ **Consistência-k**: um CSP k -consistente se para qualquer conjunto de $k - 1$ variáveis e qualquer atribuição consistente a essas variáveis, existe um valor possível para atribuir consistentemente a qualquer outra variável.

Habitualmente opta-se pela consistência de arco ou consistência de caminho.

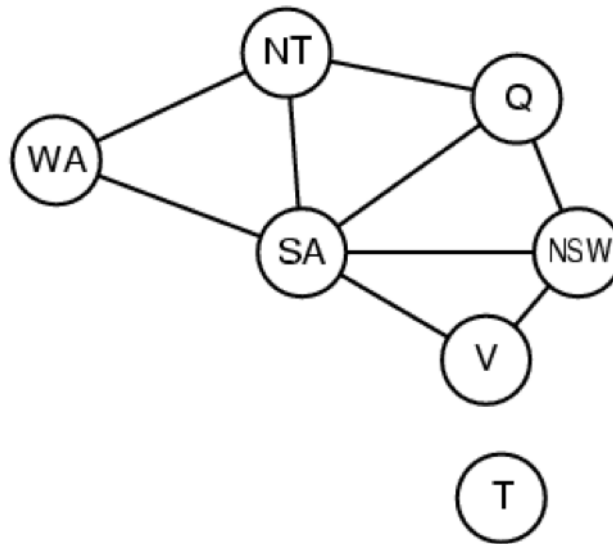
Melhorias ao algoritmo de retrocesso

Vimos o algoritmo com **retrocesso cronológico** (em caso de falha retorna para a última variável atribuída).

◇ O **retrocesso inteligente** consiste em voltar para a variável anterior responsável pela violação da restrição.

◇ Tecnicamente isto é conseguido mantendo para cada variável o conjunto de variáveis conflitantes, e actualizando-o incrementalmente (e.g. quando *RemoveInconsistent* retira valores ao domínio).

Estrutura dos Problemas



A Tasmânia e o continente são **subproblemas independentes**

Identificáveis a partir das **componentes ligadas** do grafo de restrições

Estrutura dos Problemas (cont)

Suponha-se que cada subproblema tem c variáveis de um total de n

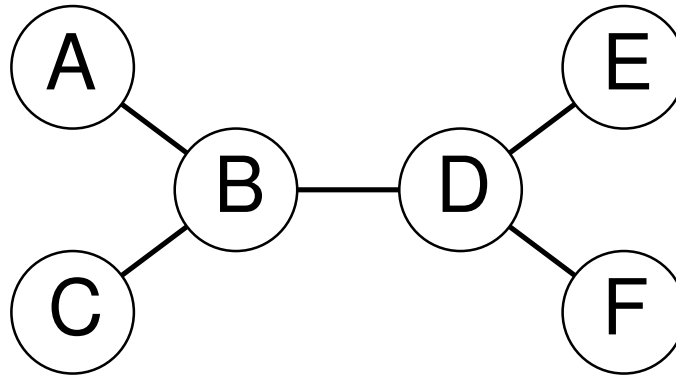
No pior caso, uma solução demora $n/c \cdot d^c$, *linear* em n

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ mil milhões de anos a 10 milhões de nós/seg

$4 \cdot 2^{20} = 0.4$ segundos a 10 milhões de nós/seg

CSPs com estrutura arbórea



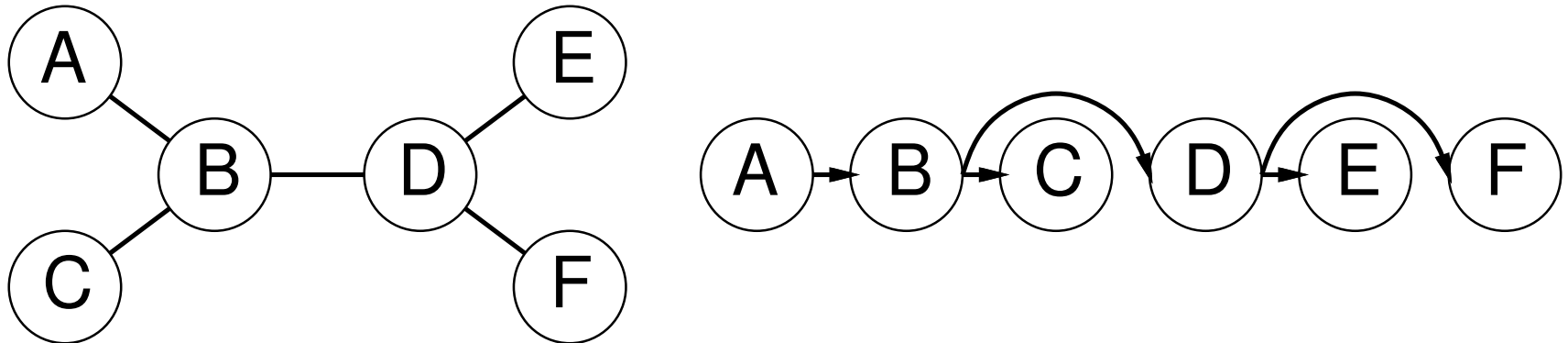
Teorema: se o grafo de restrições não tem ciclos, então o CSP pode ser resolvido em tempo $O(n d^2)$

Comparar com CSPs genéricos, cujo pior caso temporal é $O(d^n)$

Esta propriedade também se aplica ao raciocínio lógico e probabilístico: um exemplo importante da relação entre restrições sintáticas e a complexidade do raciocínio.

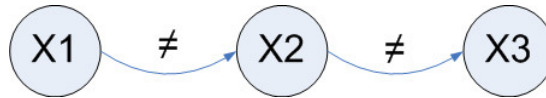
Algoritmo para CSPs com estrutura arbórea

1. Escolher uma variável como raiz, ordenar variáveis da raiz para as folhas tal que o pai de um nó aparece primeiro do que todos os seus filhos na ordenação

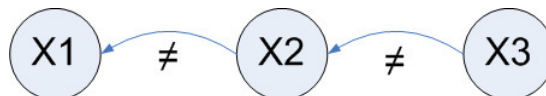


2. Para j de n até 2, aplicar $\text{REMOVEINCONSISTENT}(Parent(X_j), X_j)$
3. Para j de 1 até n , atribuir X_j consistentemente com $Parent(X_j)$

Aplicação ao exemplo



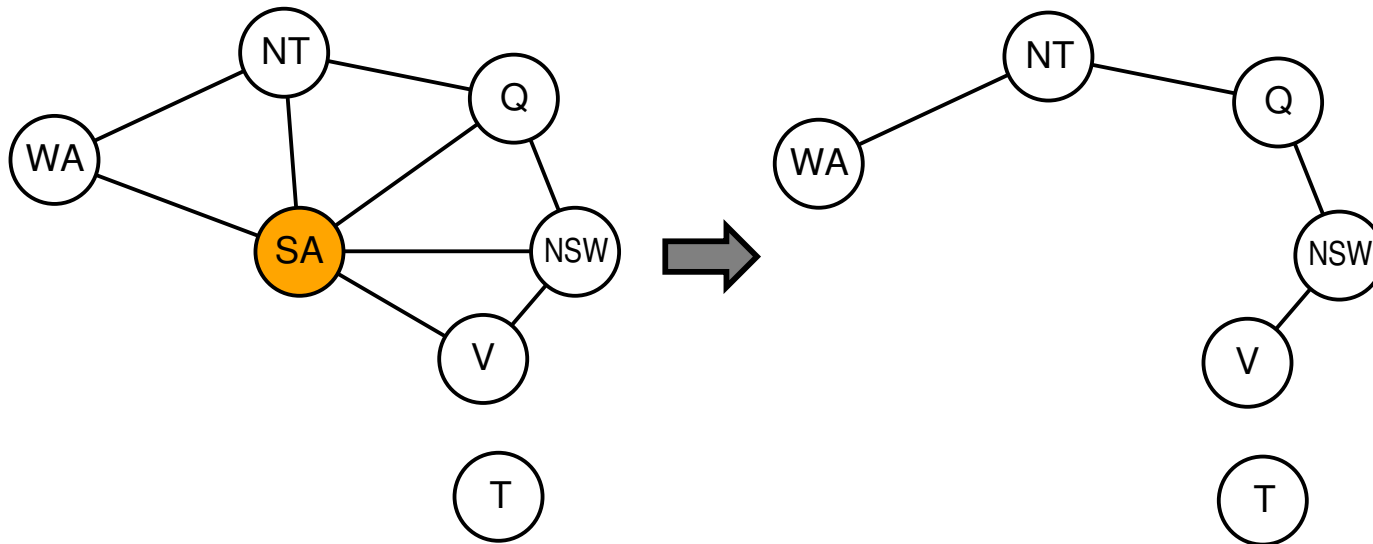
X_1	X_2	X_3	Acção
{1}	{1, 2}	{1, 2}	<i>RemoveInconsistent</i> (X_2, X_3)
{1}	{1, 2}	{1, 2}	<i>RemoveInconsistent</i> (X_1, X_2)
{1}	{1, 2}	{1, 2}	$X_1 := 1$
{1}	{2}	{1, 2}	$X_2 := 2$
{1}	{2}	{1}	$X_3 := 1$



X_1	X_2	X_3	Acção
{1}	{1, 2}	{1, 2}	<i>RemoveInconsistent</i> (X_2, X_1)
{1}	{2}	{1, 2}	<i>RemoveInconsistent</i> (X_3, X_2)
{1}	{2}	{1}	

CSPs de estrutura quasi-arbórea

Condicionamento: instanciar uma variável, reduzir os domínios dos nós vizinhos



Condicionamento por conjunto de corte: instanciar (de todas as maneiras) um conjunto de variáveis tal que o grafo resultante seja árvore

Conjunto de corte de tamanho $c \Rightarrow$ tempo de execução $O(d^c \cdot (n - c)d^2)$, muito rápido para c pequeno

Algoritmos locais para CSPs

Trepa-colinas, recristalização simulada funcionam habitualmente com estados “completos”, i.e., todas as variáveis atribuídas

Aplicação a CSPs:

permitir estados com restrições por satisfazer
os operadores *reatribuem* valores a variáveis

Seleccção de Variáveis: escolher aleatoriamente qualquer variável conflituante

Seleccção de valores por intermédio da heurística *min-conflitos*:

escolher valor que viola o menor número de restrições

i.e., trepa-colinas com $h(n)$ = número total de restrições violadas

Algoritmo min-conflitos

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max-steps, the number of steps allowed before giving up
  local variables: current, a complete assignment
                    var, a variable
                    value, a value for a variable

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max-steps do
    var  $\leftarrow$  a randomly chosen, conflicted variable from VARIABLES[csp]
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var=value in current
    if current is a solution for csp then return current
  end
  return failure
```

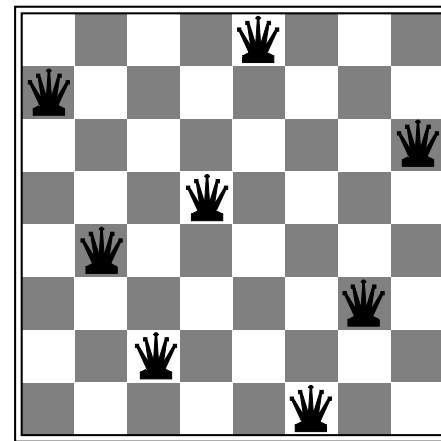
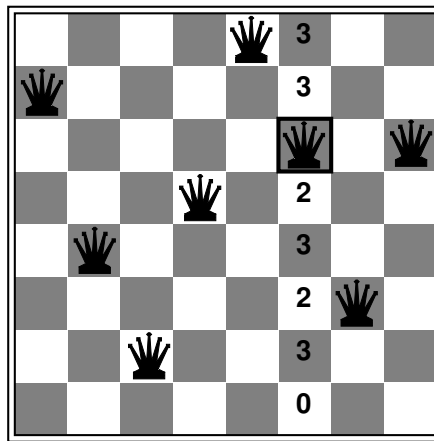
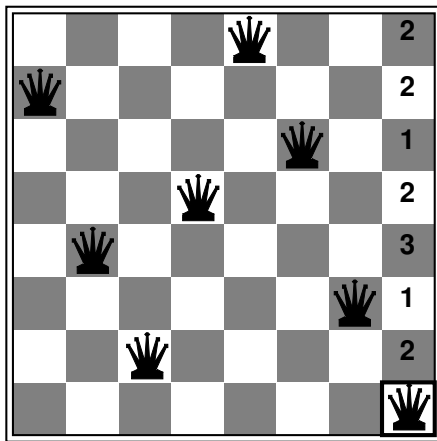
Exemplo: 8-rainhas

Estado: 8 rainhas em 8 colunas ($8^8 = 16777216$ estados)

Operadores: mover uma rainha na sua coluna

Teste Objectivo: inexistência de ataques

Avaliação: $h(n) = \text{número de ataques}$

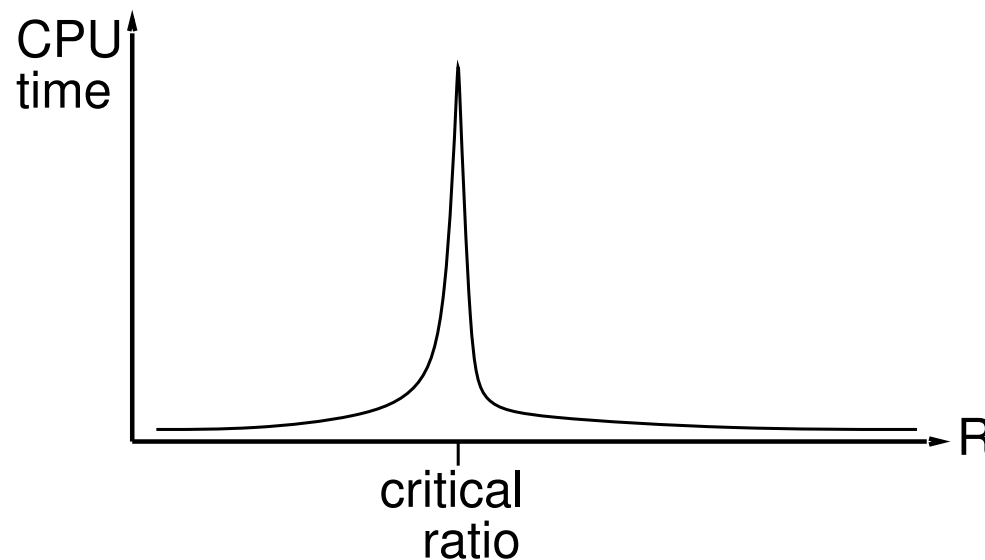


Desempenho de min-conflitos

Dado um estado inicial aleatório, consegue-se resolver n -rainhas quase em tempo constante para n arbitrário com alta probabilidade (e.g., $n = 10,000,000$)

O mesmo se verifica para qualquer CSP gerado aleatoriamente **exceptuando** para uma pequena banda do rácio

$$R = \frac{\text{número de restrições}}{\text{número de variáveis}}$$



Sumário

CSPs são um tipo especial de problema:

estados definidos por valores de um conjunto fixo de variáveis

teste objectivo definido por *restrições* nos valores das variáveis

- ◇ Retrocesso = procura em profundidade primeiro como uma variável atribuída por nó
- ◇ Heurísticas de ordenação de variáveis e selecção valores ajudam muito
- ◇ Verificação para a frente evita atribuições que irão falhar garantidamente
- ◇ Propagação de restrições (e.g., consistência de arcos) efectua trabalho adicional para restringir valores e detectar inconsistências mais cedo
- ◇ A representação de CSPs permite a análise da estrutura do problema
- ◇ CSPs de estrutura arbórea podem ser resolvidos em tempo linear
- ◇ A heurística min-conflitos é habitualmente eficaz na prática