Problemas de Satisfação de Restrições

Capítulo 5

Sumário

- ♢ CSP exemplos
- ♢ Pesquisa Backtracking para CSPs
- ♢ Estrutura dos Problemas e decomposição dos problemas
- ♢ Pesquisa local para CSPs

Constraint satisfaction problems (CSPs)

Problema de pesquisa standart:

estado é uma "caixa preta"—qualquer estrutura de dados
que suporte o teste atingiu objectivo, avaliação, sucessor

CSP:

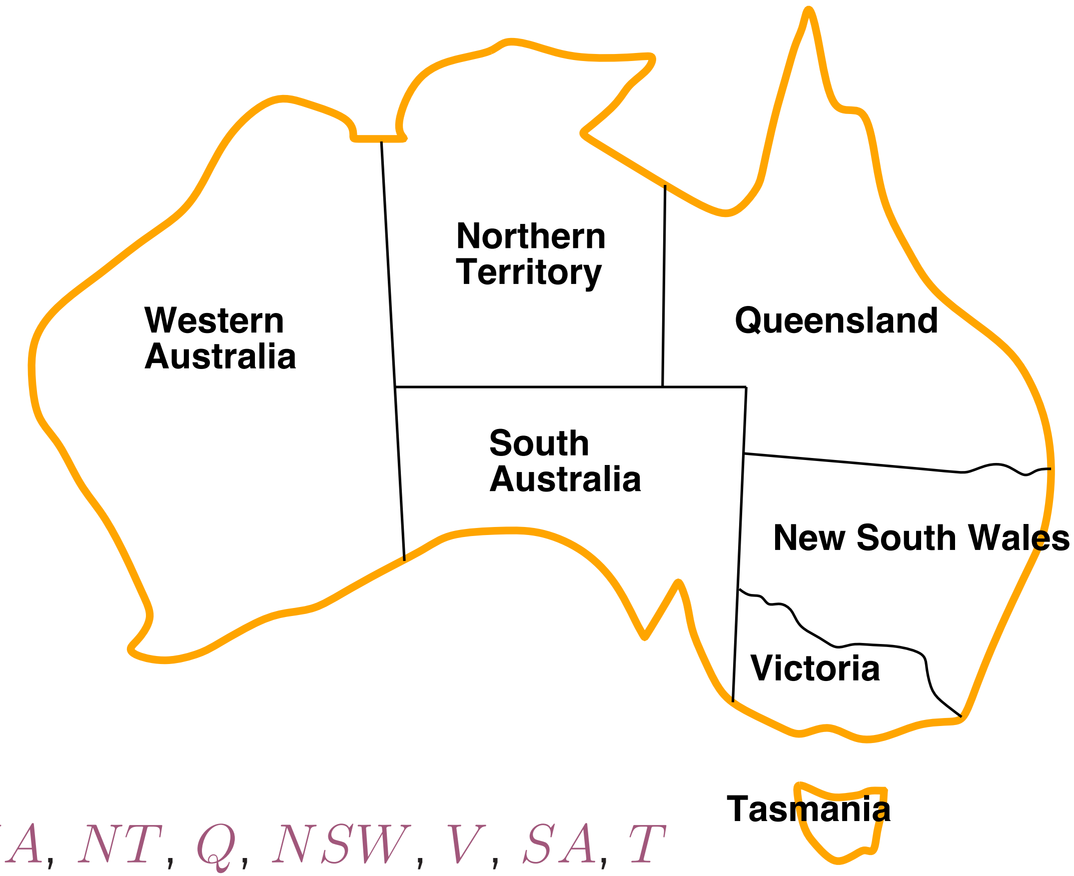
estado é definido por variáveis X_i com valores do domínio D_i

objectivo é um conjunto de restrições que especificam
as combinações permitidas dos valores dos subconjuntos das variáveis

Exemplo simples **linguagem formal para representação**

Permite usar algoritmos**gerais** mais potentes
que algoritmos de pesquisa standart

Exemplo: Coloração de mapas



Variáveis WA, NT, Q, NSW, V, SA, T

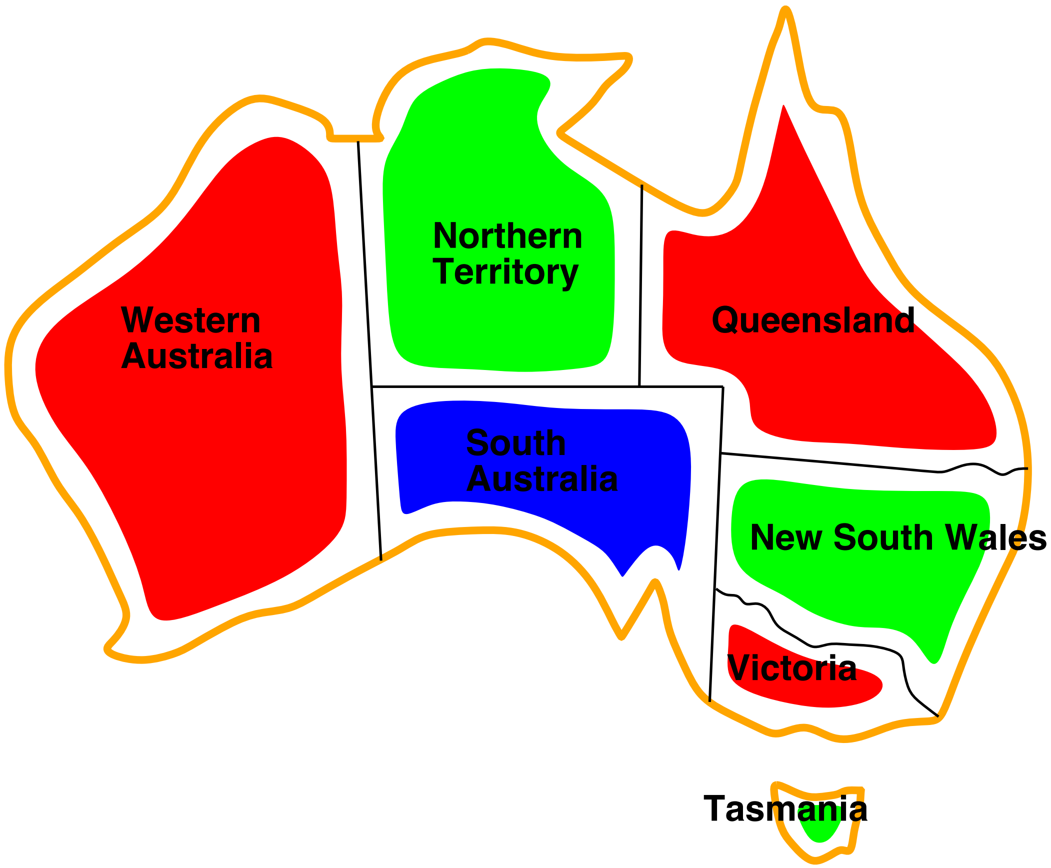
Domínios $D_i = \{red, green, blue\}$

Restrições: regiões adjacentes devem ter cores diferentes

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

$(WA, NT) \in \{(vermelho, verde), (vermelho, azul), (verde, vermelho), (verde, azul$

Exemplo: Coloração de mapas

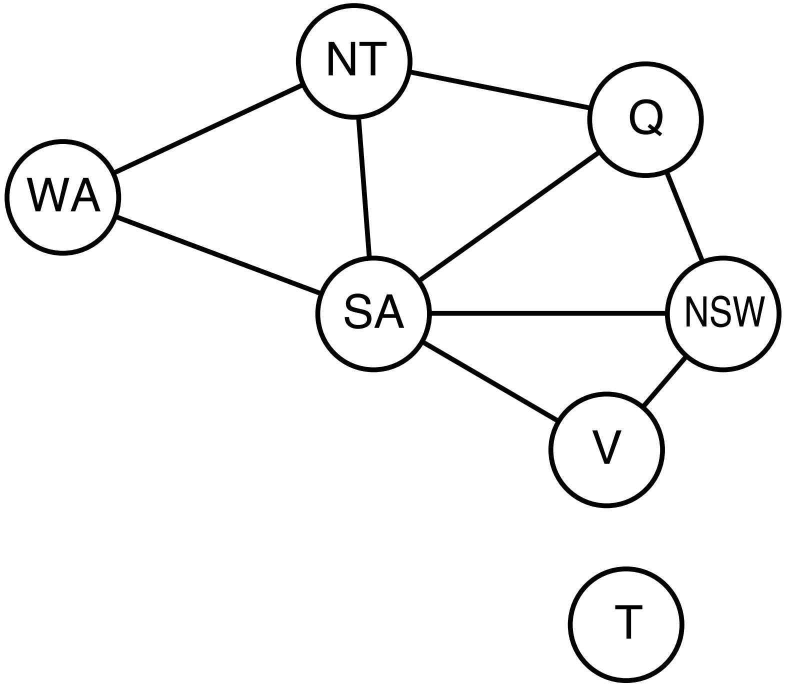


Soluções são afectações que satisfazem todas as restrições, e.g.,
$\{WA = vermelho, NT = verde, Q = vermelho, NSW = verde, V = vermelho, SA = azu$

Grafo de restrições

CSP binário: cada restrição relaciona no máximo duas variáveis

Grafo de restrições: os nós são variáveis, os arcos têm restrições



algoritmos gerais de CSP usam a estrutura em grafo
para acelerar a pesquisa. E.g., Tasmania é um subproblema independente!

Variedades de CSPs

Variáveis discretas

domínios finitos; tamanho $d \Rightarrow O(d^n)$ afectações completas

◊ e.g., CSPs Booleanos, incl. satisfação Booleana (NP-complete)

domínios infinito (inteiros, strings, etc.)

◊ e.g., escalonamento de tarefas, vaiáveis representando o dia de inicio/fim para cada tarefa

◊ é necessária um linguagem de restrições, e.g., $StartJob_1 + 5 \leq StartJob_3$

◊ restrições lineares solúveis, não linear indecidivel

Variáveis continuas

◊ e.g., tempos de inicio/fim times para as observações telescópicas do Hubble

◊ restrições lineares podem ser resolvidas em tempo polinomial usando métodos de programação linear

Variedades de restrições

restrições Unárias envolvem uma só variável,
e.g., $SA \neq verde$

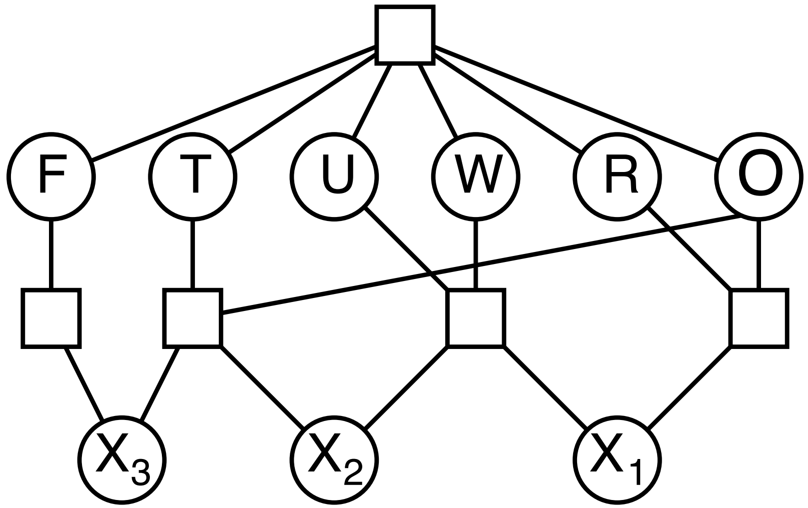
restriçõesBinárias envolvem pares de variáveis,
e.g., $SA \neq WA$

restrições de Ordem Superior envolvem 3 ou mais variáveis,
e.g., cripto aritmética

Preferencias (soft constraints), e.g., $vermelho$é melhor que $verde$
pode ser representado por um custo para cada afectação de uma variável
$\rightarrow$ problema de optimização com restrições

Exemplo: Cripto aritmética

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



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.

Problema do mundo real CSPs

Problemas de afectação
    e.g., quem ensina que que turma

Problemas de horários
    e.g., qual o horário da turma e onde?

Configurações de Hardware

Folhas de calculo

Escalonamento de transportes

Escalonamento de fabrico

Notem que muitos problemas reais envolvem variáveis de valores reais

Formulação de pesquisa (incremental)

Os estados são definidos pelos valores já afectados

◇ Estado inicial: nenhuma afectação, $\{\,\}$

◇ Função sucessor: afecta um valor a uma variável
que não entre em conflito com as afectações anteriores.
$\Rightarrow$ falhe se não há nenhuma afectação possível

◇ Teste de objectivo atingido: a afectação corrente está completa

1) É o mesmo para todos os CSPs! 😆
2) Todas as soluções aparecem na profundidade n com n variáveis
$\Rightarrow$ usar a pesquisa em profundidade
3) O caminho é irrelevante
4) $b = (n - \ell)d$ na profundidadeℓ, com $n!d^n$ folhas!!!! ☹

Pesquisa Backtracking

A afectação de variáveis é comutativa, i.e.,
[$WA = vermelho$ então $NT = verde$] é o mesmo que [$NT = verde$ então $WA = vermelho$]

Só é necessário considerar a afectação de uma variável em cada nó
$\Rightarrow$ $b = d$ e existem d^n folhas

A pesquisa em profundidade para CSPs com afectação de uma única variável chama-se pesquisa backtracking

Pesquisa Backtracking é um algoritmo de pesquisa não informada para CSPs

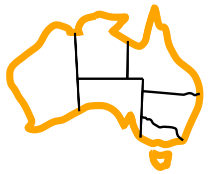
Pode resolver n-rainhas para $n \approx 25$

Pesquisa Backtracking

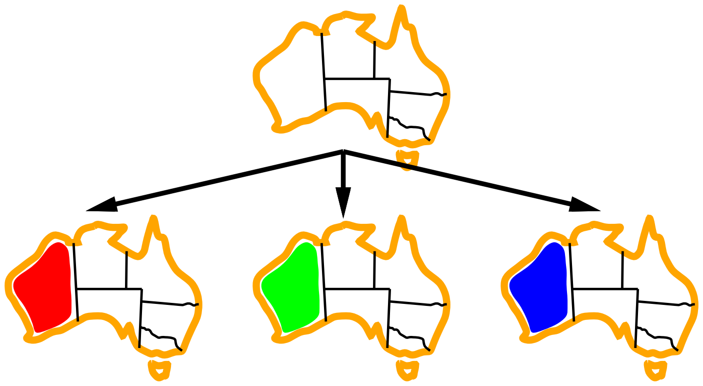
function BACKTRACKING-SEARCH(*csp*) **returns** solution/failure
  return RECURSIVE-BACKTRACKING({ }, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** soln/failure
  if *assignment* is complete **then return** *assignment*
  var ← 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 ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if $result \neq failure$ **then return** *result*
      remove {$var = value$} from *assignment*
  return *failure*

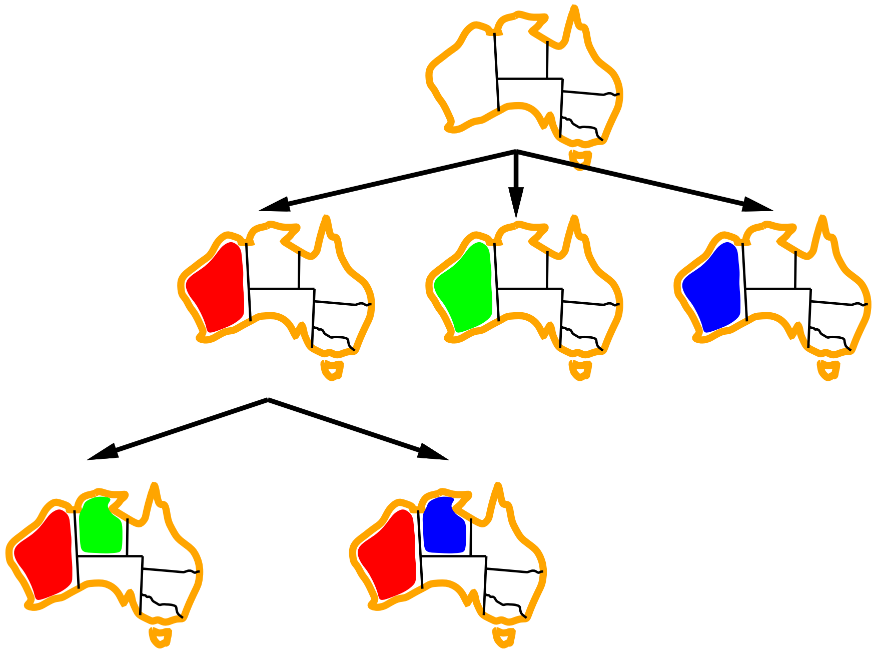
Pesquisa Backtracking



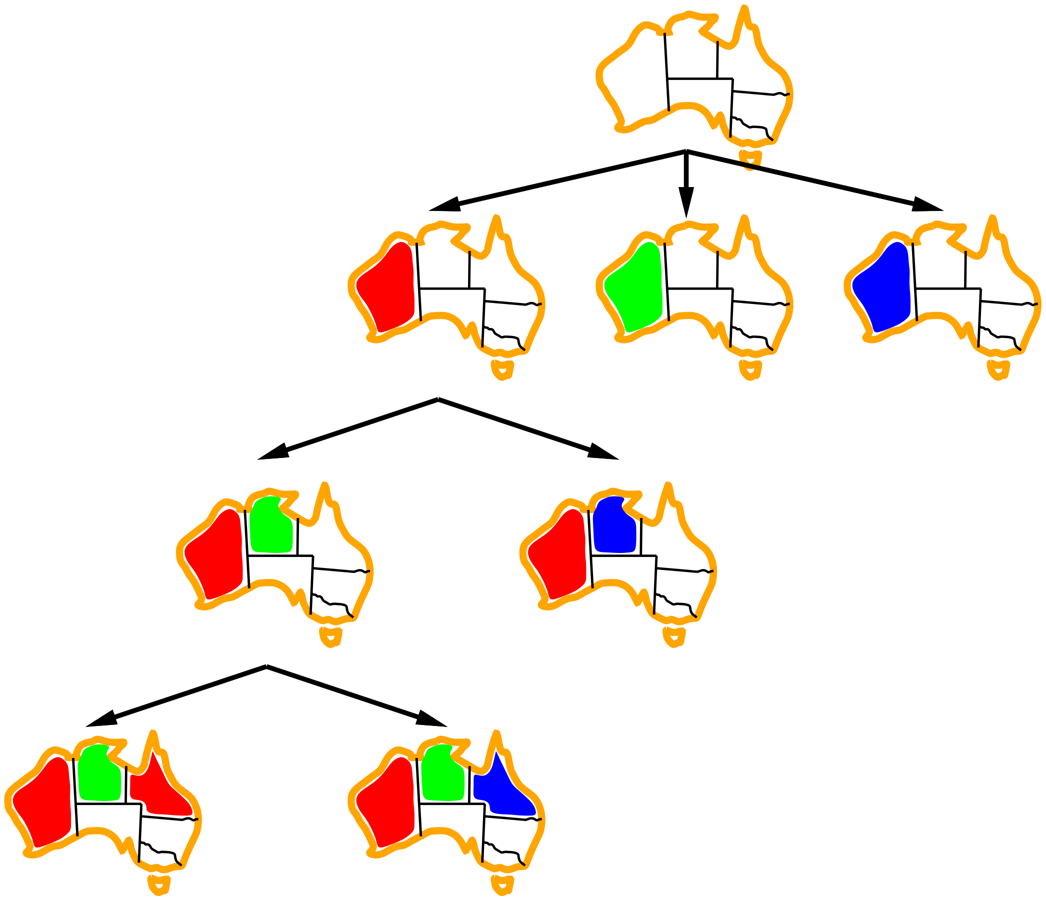
Pesquisa Backtracking



Pesquisa Backtracking



Pesquisa Backtracking



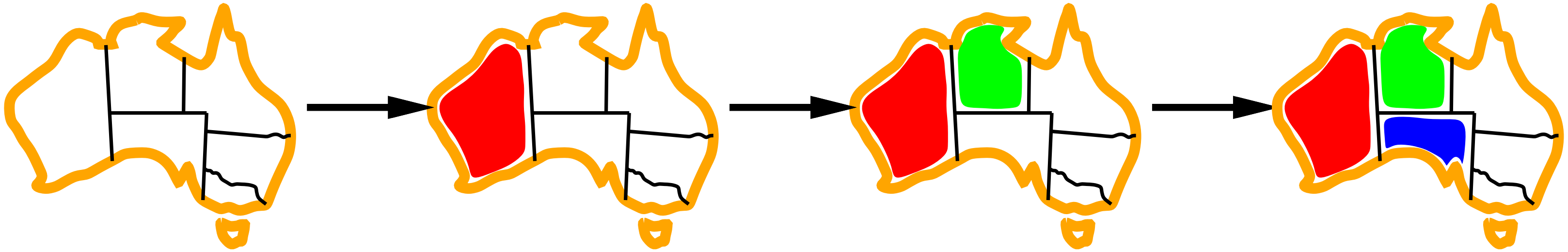
Melhorar a eficiência da pesquisa backtracking

Métodos **Gerais** podem originar ganhos elevados na velocidade:

1. Qual a próxima variável a ser afectada?
2. Por que ordem se devem experimentar os valores possíveis?
3. Pode-se detectar falhas inevitáveis mais cedo?
4. Pode-se ter em conta a estrutura do problema?

Menos valores no domínio - Minimum remaining values

Menos valores no domínio - Minimum remaining values (MRV):
escolhe as variáveis com menos valores possíveis.

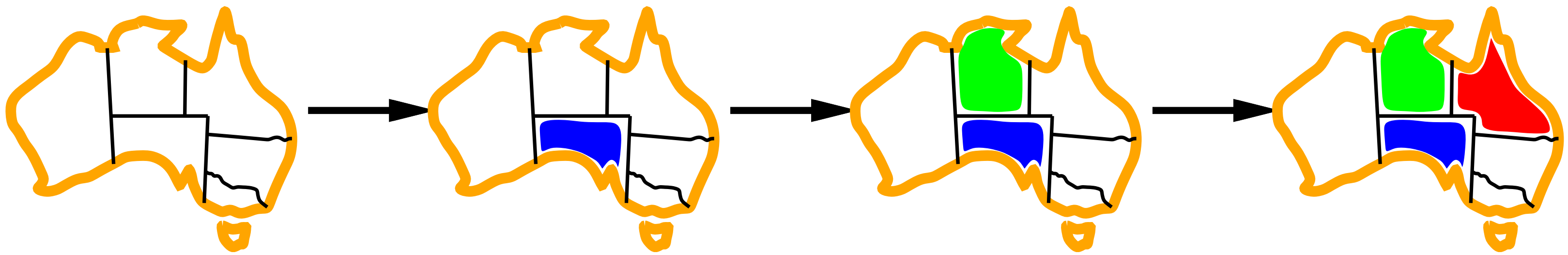


Variável com mais restrições

ligado às variáveis MRV

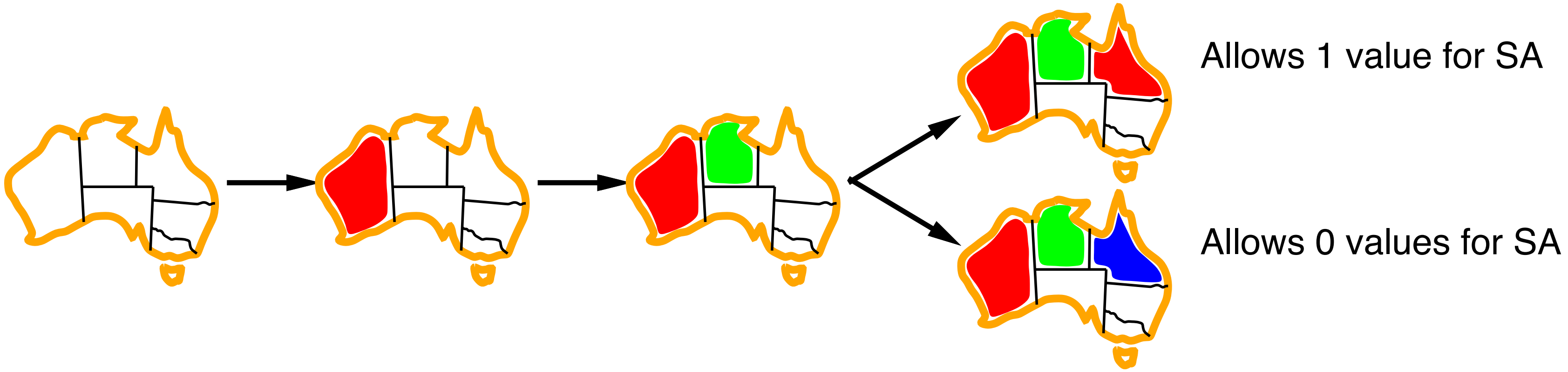
Variáveis com mais restrições:

escolher a variável com mais restrições com as variáveis por instanciar



O valor menos restringido - Least constraining value

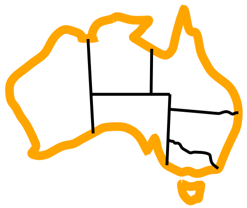
Dada uma variável escolher o valor menos restringido:
o que retira menos valores aos domínios das restantes variáveis

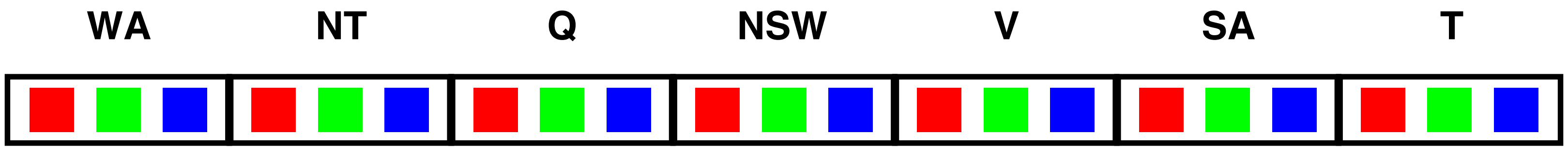


A combinação destas heurísticas torna o problema das 1000 rainhas solúvel

Forward checking

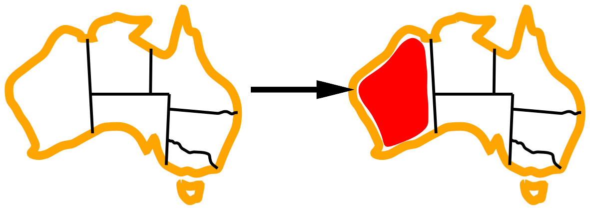
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

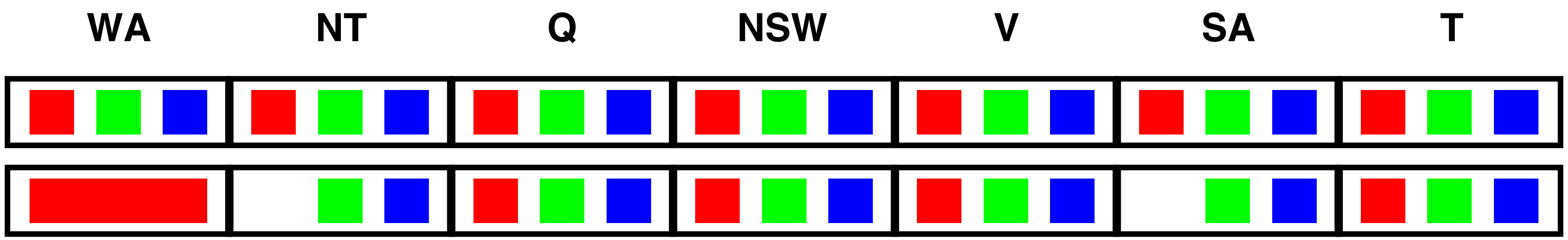




Forward checking

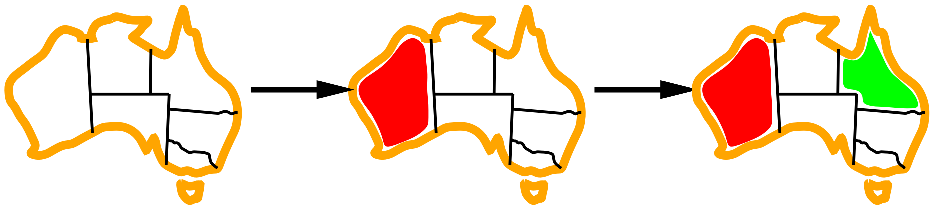
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





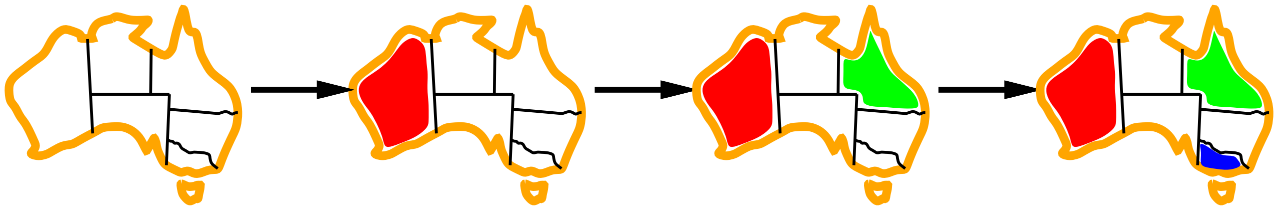
Forward checking

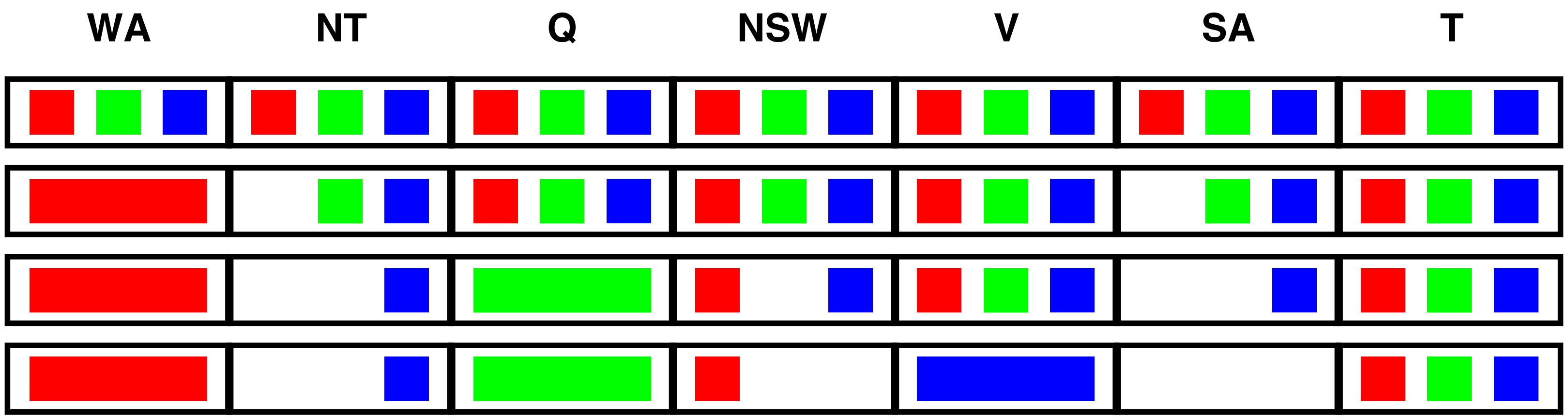
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Forward checking

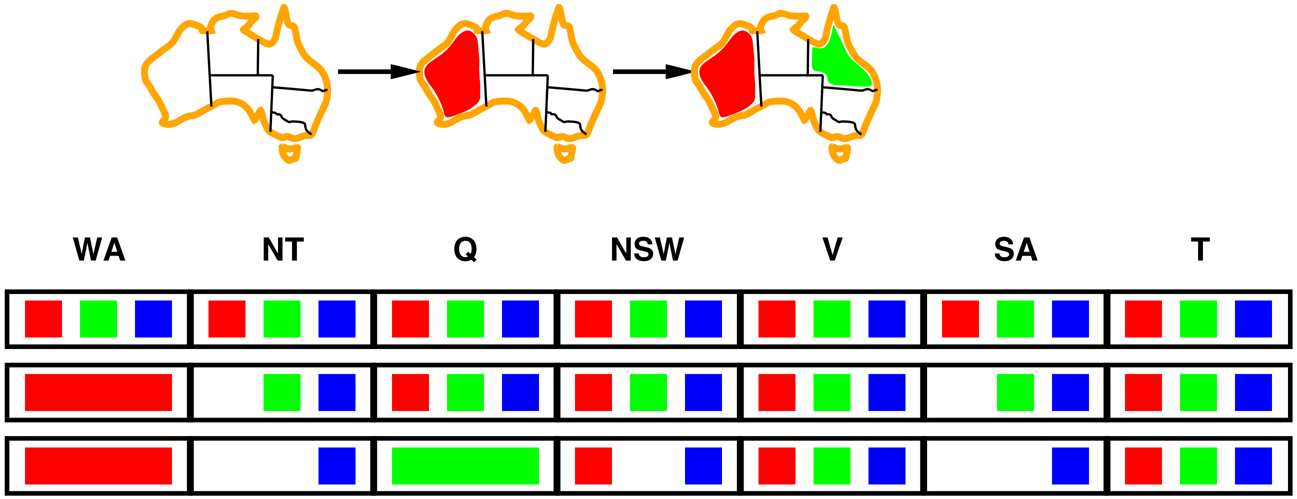
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



NT and SA cannot both be blue!

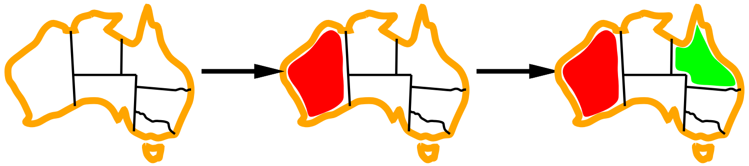
Constraint propagation repeatedly enforces constraints locally

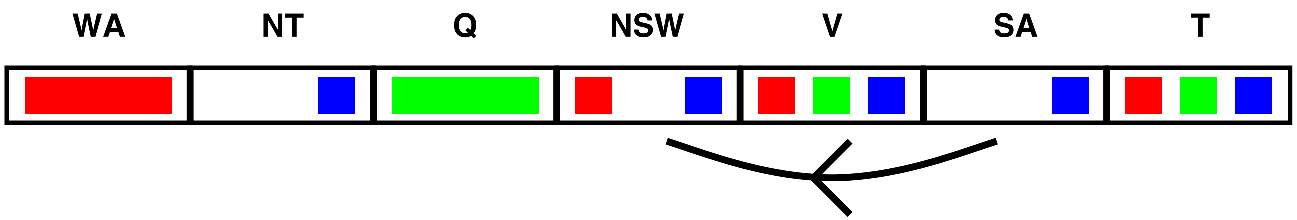
Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

for **every** value x of X there is **some** allowed y



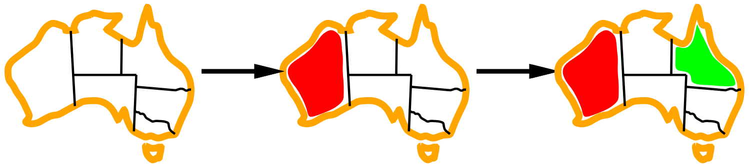


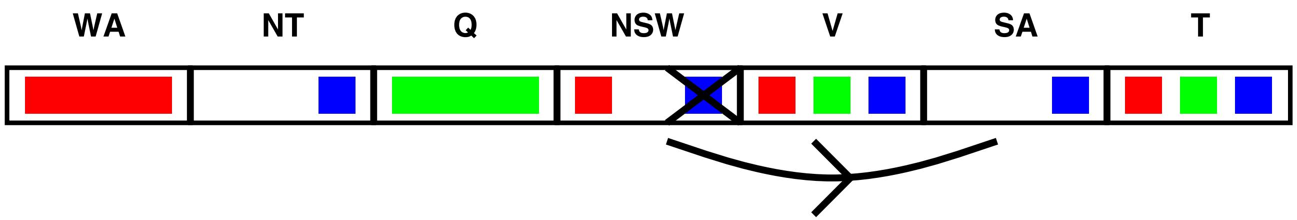
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y



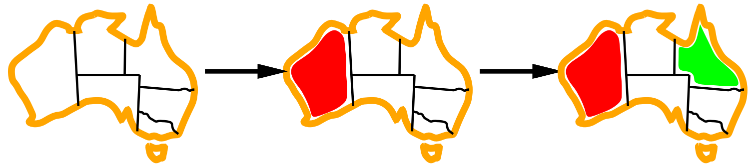


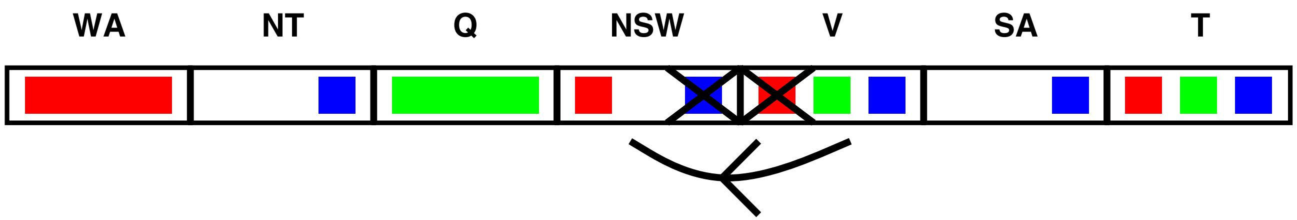
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y





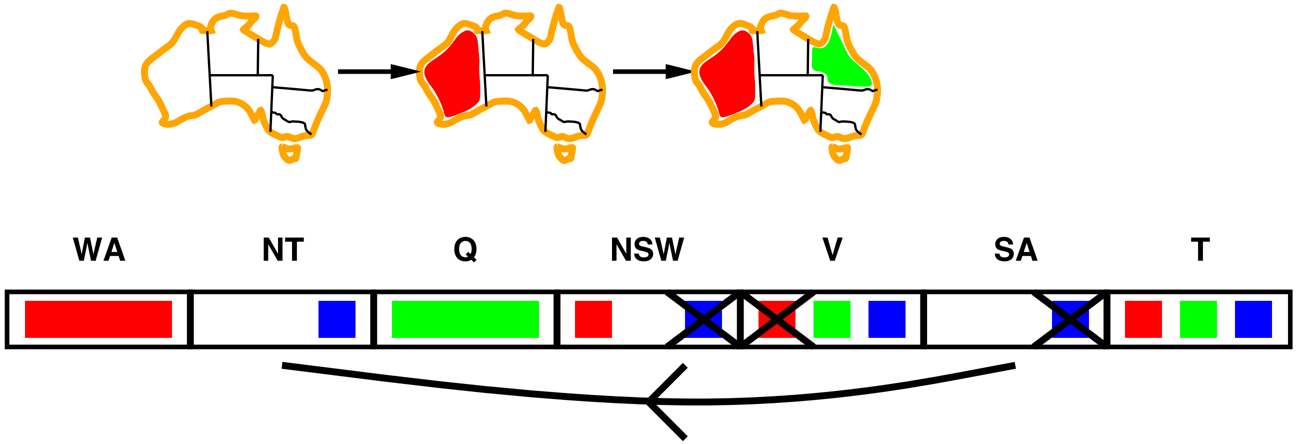
If X loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for **every** value x of X there is **some** allowed y



If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

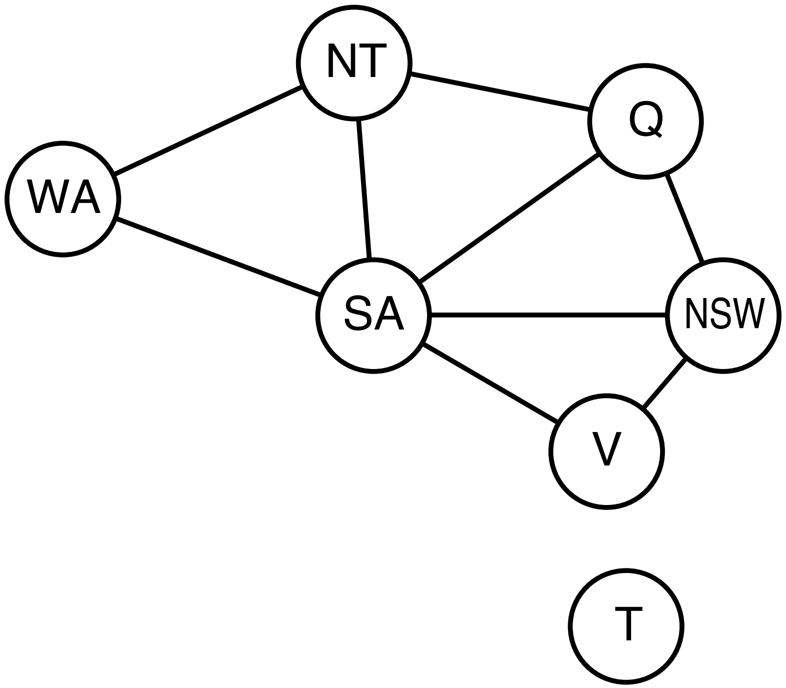
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS$[X_i]$ **do**
        add (X_k, X_i) to *queue*

---

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*

$O(n^2 d^3)$, can be reduced to $O(n^2 d^2)$ (but detecting **all** is NP-hard)

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

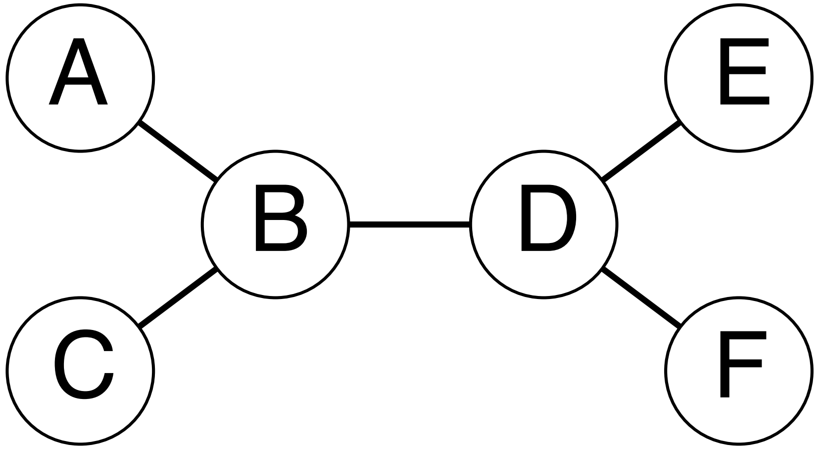
Problem structure contd.

Suppose each subproblem has c variables out of n total

Worst-case solution cost is $n/c \cdot d^c$, **linear** in n

E.g., $n = 80$, $d = 2$, $c = 20$
2^{80} = 4 billion years at 10 million nodes/sec
$4 \cdot 2^{20}$ = 0.4 seconds at 10 million nodes/sec

Tree-structured CSPs



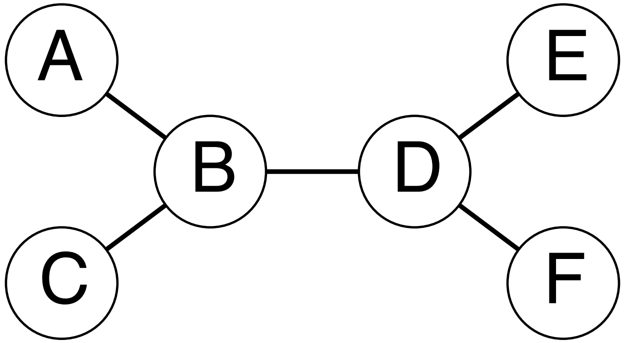
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

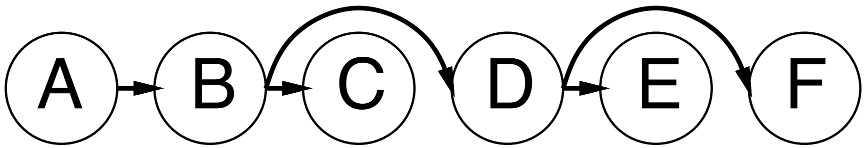
Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions
and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



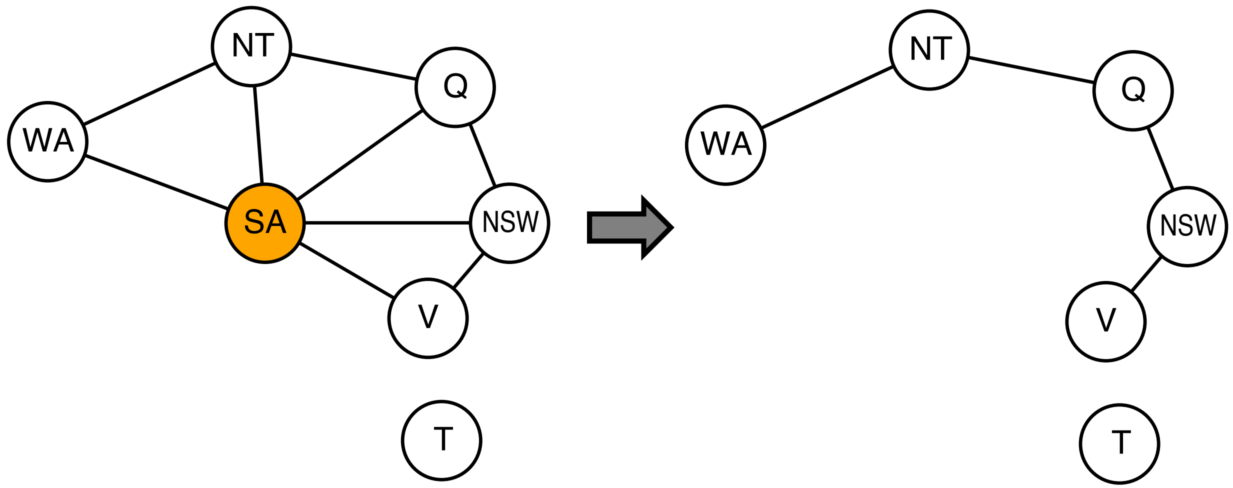


2. For j from n down to 2, apply RemoveInconsistent$(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c $\Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

To apply to CSPs:
- allow states with unsatisfied constraints
- operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by min-conflicts heuristic:
- choose value that violates the fewest constraints
- i.e., hillclimb with $h(n) =$ total number of violated constraints

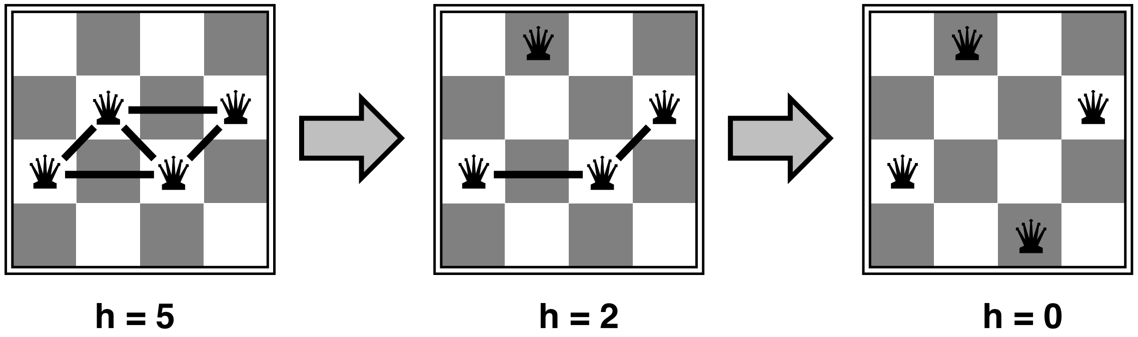
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n) =$ number of attacks

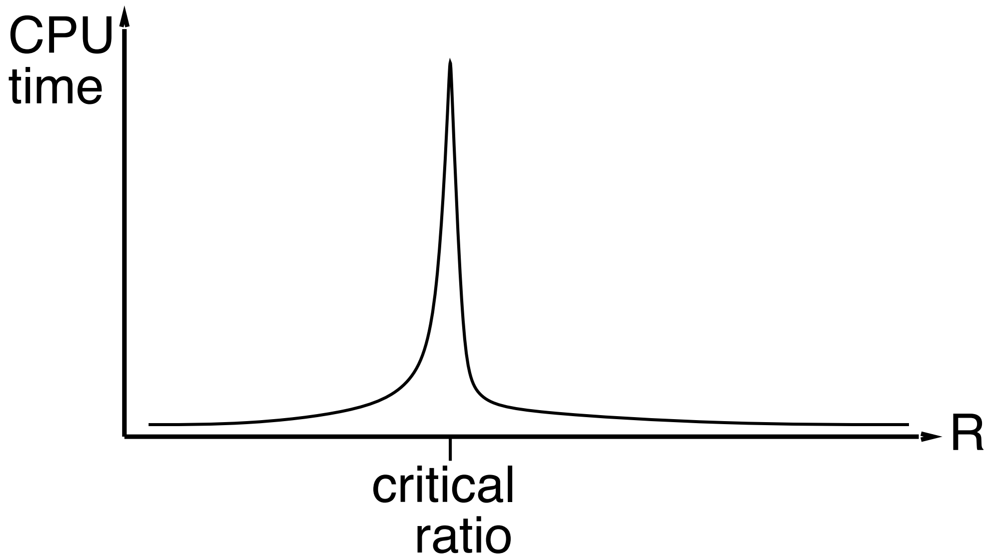


Performance of min-conflicts

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)

The same appears to be true for any randomly-generated CSP
except in a narrow range of the ratio

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



Summary

CSPs are a special kind of problem:
states defined by values of a fixed set of variables
goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work
to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice