Satisfação de Restrições

Capítulo 5

Sumário

- Problemas de Satisfação de Restrições
  - CSPs, do Inglês "Constraint Satisfaction Problems"
- Inferência + Procura com Retrocesso para CSPs

Problemas de Satisfação de Restrições

- Até agora olhámos para resolução de problemas através de procura
  - Problemas de Procura Tradicional
  - Procura num espaço de estados
    - estado é uma "caixa preta" – qualquer estrutura de dados que suporte função sucessores, função heurística, e teste objetivo
  - Utiliza-se informação específica do problema/domínio para guiar processo de procura

- Ideia:
  - Porque não tentar usar informação acerca da estrutura dos estados para guiar o processo de procura?
  - O estado deixa de ser uma caixa preta!

Problemas de Satisfação de Restrições

- Ideia:
  - Representar cada estado por um conjunto explícito de variáveis
  - Variáveis essas que podem ter restrições
- Problemas de Satisfação de Restrições
  - Designados de CSP (Constraint Satisfaction Problems)

Problemas de Satisfação de Restrições

- CSP, definido por 3 componentes:
  - X - Conjunto de variáveis – $\{X_1, \ldots, X_n\}$
  - D - Conjunto de domínios – $\{D_{x_1},..,D_{x_n}\}$
    - Um domínio para cada variável
    - Define os valores que uma variável pode tomar
    - $D_{x_1} = \{v_1,\ldots,v_k\}$
  - C - Conjunto de restrições: especificam combinações possíveis de valores para subconjuntos de variáveis
    - Representação explícita
      - $<(X_1,X_2),[(0,1),(1,0)]>$ para um domínio {0,1}
    - Representação implícita
      - $<(X_1,X_2), X_1 \neq X_2>$

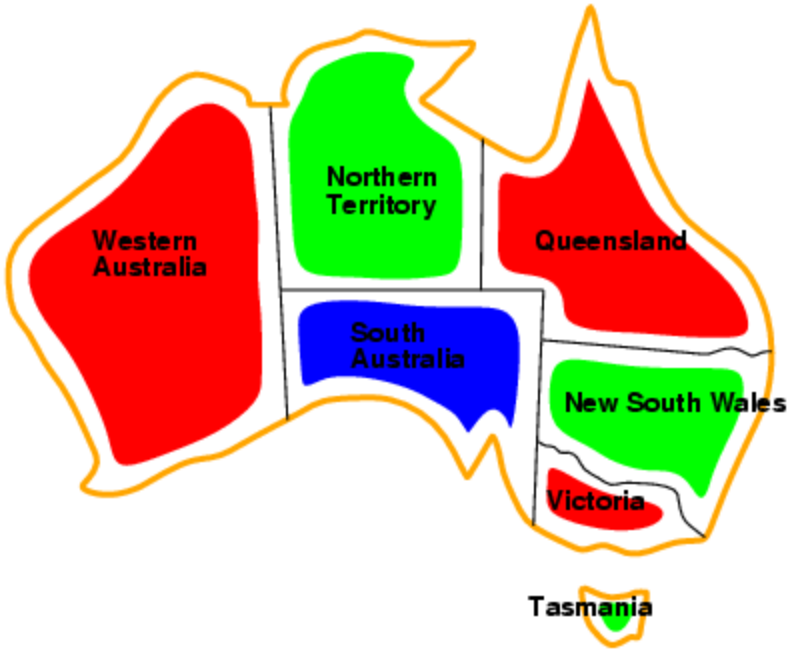
Problemas de Satisfação de Restrições

- Estado num CSP
  - Atribuição de valores a 0 ou mais variáveis do problema.
    - {} – atribuição vazia
    - $\{X_i = v_i, X_j = v_j, \ldots\}$
  - Atribuição parcial
    - atribui valores apenas a algumas variáveis
  - Atribuição completa
    - atribui valores a todas as variáveis do problema
  - Atribuição consistente
    - atribuição que não viola nenhuma restrição
- Solução de um CSP
  - Atribuição completa e consistente

Exemplo: Coloração de Mapa

- Variáveis *WA, NT, Q, NSW, V, SA, T*
- Domínios D_i = {vermelho, verde, azul}
- Restrições: regiões adjacentes com cores diferentes
- e.g., WA ≠ NT, ou (WA,NT) ∈ {(vermelho,verde), (vermelho,azul), (verde,vermelho), (verde,azul), (azul,vermelho), (azul,verde)}

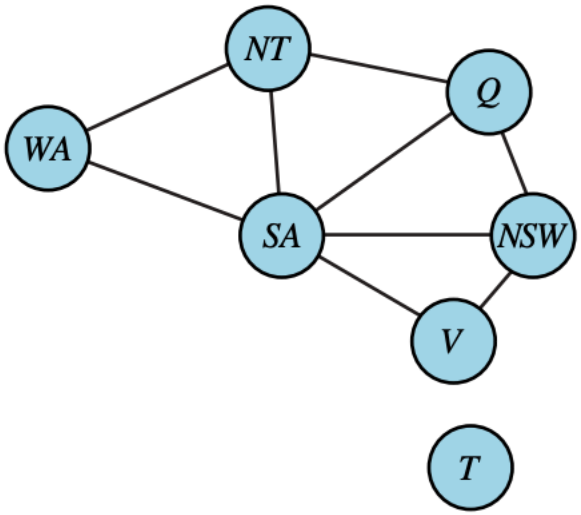
Exemplo: Coloração de Mapa



- Soluções são atribuições completas e consistentes, e.g., WA = vermelho, NT = verde, Q = vermelho, NSW = verde, V = vermelho, SA = azul, T = verde

Grafo de Restrições

- CSP binário: cada restrição relaciona duas variáveis
- Grafo de restrições: nós são variáveis, arcos são restrições



Vantagens de CSP

- Porquê usar CSPs?
- Facilidade de representação
  - Muitos problemas são naturalmente representados por restrições
    - Sudoku
      - Casas com valores entre 1 e 9
      - Não posso ter valores repetidos
        - Linhas
        - Colunas
        - Caixas de 3x3
    - Problemas de escalonamento (*scheduling*)

Vantagens de CSP

- Eficiência
  - Extremamente mais rápidos que procura tradicional para certos tipos de problema
  - Eliminam rapidamente grandes porções do espaço de estados
    - Ex: {SA= azul} → podemos remover a cor azul das variáveis correspondentes aos 5 vizinhos de SA
    - Em vez de considerarmos 3 cores → $3^5 = 243$
    - Consideramos 2 cores → $2^5 = 32$

Variáveis em CSPs

- Variáveis discretas
  - Domínios finitos:
    - n variáveis, domínio dimensão $d \rightarrow O(d^n)$ atribuições completas
    - e.g., CSPs Booleanos, incluindo satisfação Booleana - SAT (NP-completo)
  - Domínios infinitos:
    - inteiros, strings, etc.
    - e.g., escalonamento de tarefas, variáveis têm datas de início/fim para cada tarefa
    - necessidade de uma linguagem de restrições,
      e.g., $InícioTarefa_1 + 5 \leq InícioTarefa_3$
- Variáveis contínuas
  - e.g., datas de início/fim para observações do telescópio espacial Hubble
  - restrições lineares solúveis em tempo linear usando programação linear

Tipos de restrições

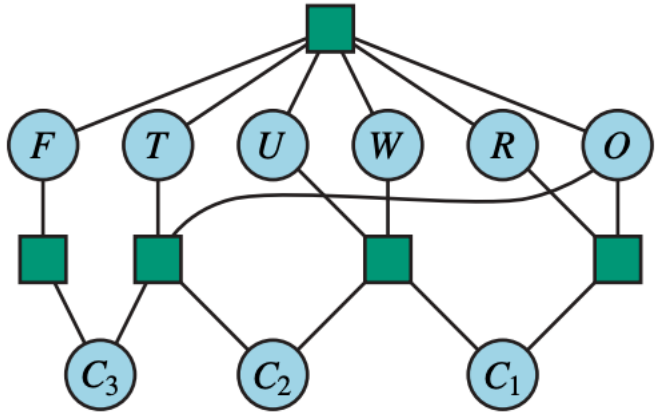
- Restrições unárias referem-se a uma variável,
  - e.g., SA ≠ verde
- Restrições binárias referem-se a pares de variáveis,
  - e.g., SA ≠ WA
- Restrições de ordem superior envolvem 3 ou mais variáveis,
  - e.g., restrições para cripto-aritmética

Tipos de restrições

- Restrições de preferências
  - Indicam preferências de atribuições/soluções face a outras
  - E.g. Criação de horários
    - Restrição absoluta
      - Um professor não pode dar duas aulas ao mesmo tempo
    - Restrição de preferência
      - Prefiro não dar aulas sexta-feira
  - Implementado como custos de atribuições de variáveis
    - Atribuição de uma aula à sexta-feira tem custo 2
    - Aula noutro dia tem custo 1
  - CSP com preferências resolvidos como problemas de otimização (procura sistemática ou local)
    - Constraint Optimization Problem

Exemplo: Cripto-aritmética

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



- Variáveis: $F\ T\ U\ W\ R\ O\ C_1\ C_2\ C_3$
- Domínios: $\{0,1,2,3,4,5,6,7,8,9\}$
- Restrições: *Alldiff (F,T,U,W,R,O)*
  - $O + O = R + 10 \cdot C_1$
  - $C_1 + W + W = U + 10 \cdot C_2$
  - $C_2 + T + T = O + 10 \cdot C_3$
  - $C_3 = F,\ T \neq 0,\ F \neq 0$

CSPs: exemplos reais

- Problemas de atribuição
  - e.g., quem ensina o quê?
- Problemas de horários
  - e.g., que aulas são oferecidas, quando e onde?
- Escalonamento de transportes
- Escalonamento do processo de fabrico

Inferência em CSP

- Num espaço de estados convencional um algoritmo só pode fazer uma coisa:
  - Procurar
- Num CSP um algoritmo pode:
  - Procurar
  - Fazer inferência – Propagação de Restrições

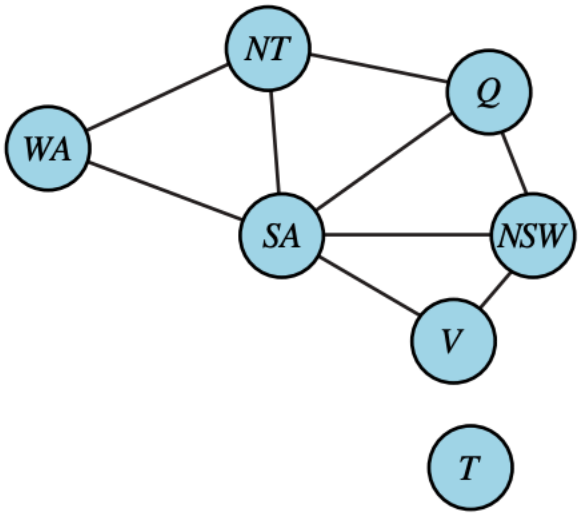
- Propagação de Restrições
  - Usar as restrições para reduzir o domínio de variáveis

Propagação de Restrições

- Propagação de Restrições
  - Pode ser feito alternadamente com procura
  - Ou como pré-processamento antes de a procura começar
    - Por vezes este pré-processamento pode resolver completamente o problema
    - E nem precisamos de procura
- Vamos olhar para um CSP como um grafo
  - Variáveis – nós do grafo
  - Restrições binárias - arcos

Propagação de Restrições

- Grafo de restrições: nós são variáveis, arcos são restrições



Propagação de Restrições

- Consistência de nó
  - Forma de consistência mais simples
  - Uma variável (nó no grafo) é nó-consistente sse
    - Todos os valores no domínio da variável satisfazem as restrições unárias da variável
    - E.g. SA ≠ verde
      - Podemos tornar a variável nó-consistente ao remover verde do domínio de SA
      - D_{SA} = {vermelho,azul}
  - Um grafo é nó-consistente se todas as variáveis do grafo são nó-consistentes

Propagação de Restrições

- Consistência de arcos
  - Consideremos 2 variáveis X e Y ligadas por um arco
  - X é consistente em arco para Y sse
    - Para cada valor do domínio D_X
    - Existe um valor do domínio D_Y que satisfaz a restrição binária do arco X → Y
  - Y é consistente em arco para X sse
    - Para cada valor do domínio D_Y
    - Existe um valor do domínio D_X que satisfaz a restrição binária do arco Y → X
  - Um CSP ou grafo de CSP é arco-consistente sse
    - cada variável é arco-consistente com todas as restantes variáveis

Consistência de Arcos

- Exemplo:
  - Variáveis: X,Y
  - Domínio: D_X={0,1,…,9} D_y = {0,1,..,9}
  - Restrições: $Y = X^2$
  - Para tornar X → Y arco consistente
    - D_X = {0,1,2,3}
  - Para tornar Y → X arco consistente
    - D_Y = {0,1,4,9}
  - Neste caso o CSP é arco-consistente

- Algoritmo mais conhecido para consistência de arco – AC-3

Algoritmo AC-3

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

- Cada restrição binária corresponde a 2 arcos (um por cada direção)
- Se uma variável *X* perde um valor, então os vizinhos de *X* têm de ser revistos
- O algoritmo só termina quando todos os domínios estabilizarem
- Se algum domínio for vazio, então sabemos que não existe solução

Algoritmo AC-3

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised ← *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised ← *true*
  return *revised*

Basta existir um valor em Dj para não apagarmos x

Exemplo: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

- Variáveis
  - A1,…,A9,B1,…,B9…
- Domínios
  - $D_{A1}=D_{A2}=\ldots=\{1,\ldots,9\}$
- Restrições
  - Alldiff(A1,…,A9)
  - Alldiff(A1,…,I1)
  - Alldiff(A1,A2,A3,B1,…,C3)
  - …

Exemplo: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

- Variável E6
  - Restrições Caixa
    - ~~1,2,7,8~~
  - Restrições Linha
    - ~~7,8~~
  - Restrições Coluna
    - ~~5,6,2,8,9,3~~
  - $D_{E6}=\{4\}$

Exemplo: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					4			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

- Variável I6
  - Restrições Coluna
    - ~~5,6,2,4,8,9,3~~
    - Agora já sabemos que E6=4
  - Restrições Caixa
    - ~~6,9,2,3,1~~
  - $D_{I6}=\{7\}$

Exemplo Sudoku

- Algoritmo AC-3 consegue resolver este exemplo de Sudoku
  - No entanto apenas funciona para os puzzles mais simples
- Para puzzles mais complexos
  - Chegamos a um ponto onde todas as variáveis são arco-consistentes
  - Mas ainda existem variáveis com vários valores possíveis no domínio

Exemplo Sudoku

- Para resolver puzzles mais complexos
  - Ou usamos algoritmos de consistência mais fortes
  - Ou precisamos de misturar inferência com procura
    - Escolhemos um valor possível para uma variável
    - Aplicamos propagação de restrições
    - Se não funcionar, voltamos para trás, escolhemos outro valor para a variável e tentamos de novo
  - Estes algoritmos conseguem resolver os problemas mais difíceis Sudoku
    - < 0.1 segundos :D

Propriedades AC-3

- Complexidade temporal para AC-3:
  - Considerem um CSP com
    - **n** variáveis
    - **d** tamanho máximo de domínio
    - **c** restrições correspondentes a arcos binários
  - Cada arco pode ser inserido na fila no máximo até *d* vezes
    - porque o domínio só tem d valores para ser removidos
  - Temos que fazer c . d verificações
  - A consistência de um arco é verificada em $O(d^2)$
  - AC-3: $O(c \cdot d \cdot d^2) = O(c \cdot d^3)$

Consistência de Arcos

- Problemas de consistência de arcos
  - Em certos problemas não consegue inferir informação suficiente
  - E.g. colorir mapa Austrália

Coloração de Mapa (2 cores)

- Variáveis *WA, NT, Q, NSW, V, SA, T*
- Domínios D_i = {vermelho, azul}
- Restrições: regiões adjacentes com cores diferentes
- e.g., WA ≠ NT

Consistência de Arcos

- No exemplo anterior
  - Consistência de arco não faz nada
    - Para cada arco X,Y
    - Cada valor de D_X tem um valor de D_Y válido
      - X = vermelho, Y = azul
      - X = azul, Y = vermelho
  - No entanto não existe solução
    - Precisamos pelo menos de 3 cores
    - AC3 não consegue descobrir que não existe solução
  - Para resolvermos o problema precisamos de ...

Consistência de Caminhos

- Consistência de Caminhos
  - Em vez de analisar pares de variáveis
  - Vamos analisar trios de variáveis

  - Um conjunto de 2 variáveis $\{X_i,X_j\}$ é consistente em caminho para uma 3.ª variável X_m sse

    - Para cada atribuição {Xi=a,Xj=b} consistente com as restrições $\{X_i,X_j\}$, então tem que existir uma atribuição para X_m que satisfaça:
      - As restrições em $\{X_i,X_m\}$
      - As restrições em $\{X_m,X_j\}$

Consistência de Caminhos

- Exemplo:
  - Tornar {WA,SA} consistente em caminho para NT
  - Atribuições consistentes de {WA,SA}
    - {WA = vermelho, SA = azul}, {WA = azul, SA = vermelho}
  - Para cada uma das atribuições verificamos as atribuições para NT
    - Nenhuma atribuição para NT é válida
      - Conflito com {WA,NT} ou com {NT,SA}
    - Logo $D_{NT} = \{\}$
    - Não existe solução

Consistência de Caminhos

- Algoritmo consistência de Caminhos
  - PC-2 (Path Consistency 2)
  - Muito semelhante ao AC-3

Consistência K

- Consistência K
  - Um CSP é K-consistente sse
    - Para qualquer conjunto de k-1 variáveis
    - E para qualquer atribuição consistente a essas variáveis
    - Podemos sempre atribuir um valor consistente a qualquer nova variável k
  - 1-consistente → consistência de nó
  - 2-consistente → consistência de arco
  - 3-consistente → consistência de caminho
  - …

Consistência K

- Um CSP é **fortemente K-consistente** sse
  - é K-consistente
  - é (K-1)-consistente
  - …
  - é 1-consistente
- Encontrar uma solução num CSP com n variáveis e que seja fortemente n-consistente é relativamente simples
- No entanto, tornar um CSP fortemente n-consistente tem uma complexidade exponencial em n.
  - Quer em tempo
  - Quer em memória
- Portanto esta abordagem não é usada

Restrições globais

- Ocorrem frequentemente em problemas reais
- Normalmente tratadas por algoritmos específicos
  - Mais eficientes que os algoritmos de propagação de restrições genéricos que vimos até agora
  - Exemplo: Alldiff
    - Todas as variáveis envolvidas num Alldiff têm que ter valores distintos

Restrições globais

- Alldiff
  - Deteção de inconsistência muito simples
    - Se temos **m** variáveis no alldiff
    - E no seu conjunto elas têm **n** valores distintos possíveis
    - Se m > n então a restrição nunca poderá ser satisfeita
  - Algoritmo
    - Remover qualquer variável da restrição que tenha um domínio com um único valor
    - Apagar o valor da variável do domínio das restantes variáveis
    - Repetir enquanto houver variáveis com domínio tamanho 1
    - Se a qualquer altura Dx = {} ou existem mais variáveis que valores de domínio
      - Então a restrição Alldiff é violada

Restrições globais

- Atmost (tb designada de resource constraint)
  - Atmost(10,P1,P2,P3,P4)
    - A soma das 4 variáveis tem que ser no máximo 10
  - Podemos detetar inconsistência ao somar os valores mínimos de cada domínio
    - Ex: $D_{P1}=D_{P2}=D_{P3}=D_{P4}=\{3,4,5,6\}$
  - Também podemos forçar consistência ao apagar valores máximos não consistentes com valores mínimos
    - Ex: $D_{P1}=\ldots=D_{P4}=\{2,3,4,5,6\}$
    - Podemos apagar 5,6 de cada domínio
    - Variante do AC-3 focado em restrições em vez de arcos

Inferência em CSP

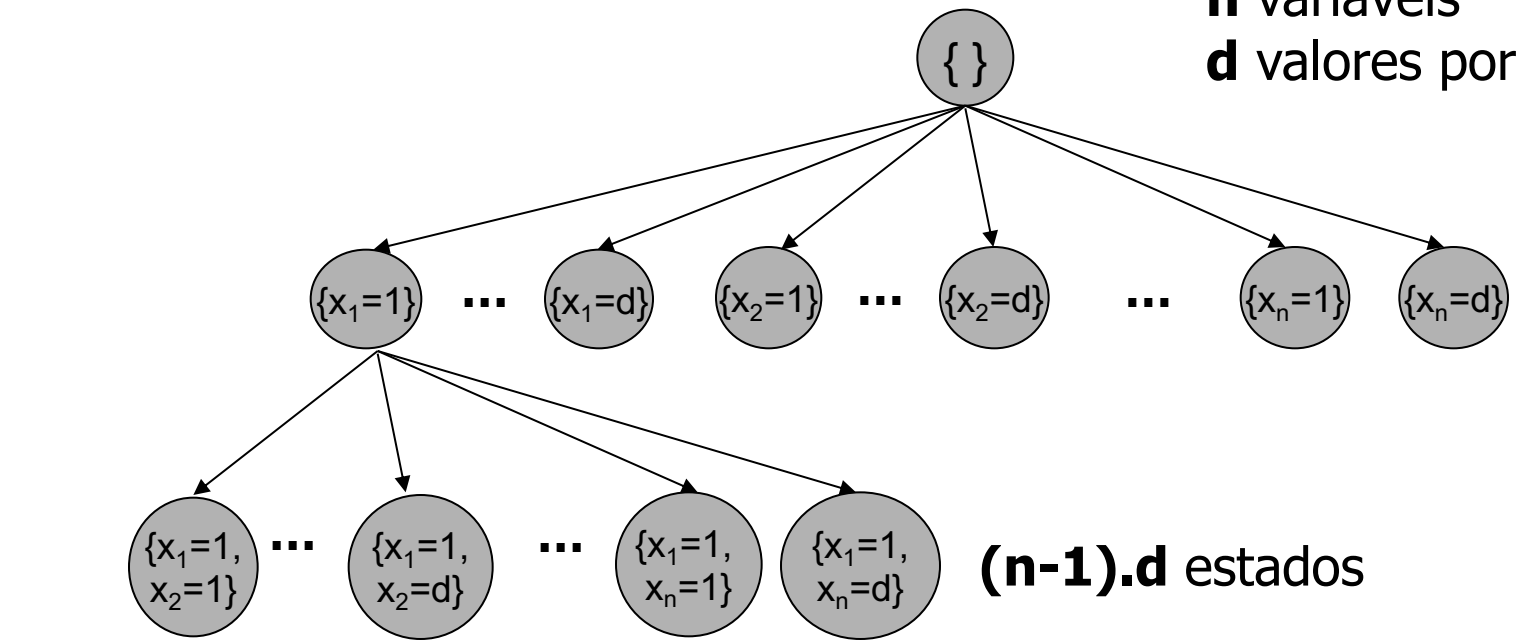
- Num espaço de estados convencional um algoritmo só pode fazer uma coisa:
  - Procurar
- Num CSP um algoritmo pode:
  - Procurar
  - Fazer inferência – Propagação de Restrições

- Propagação de Restrições
  - Usar as restrições para reduzir o domínio de variáveis

Procura em CSP

- Como vimos, nem sempre todos os problemas podem ser resolvidos apenas com inferência
- E será que os podemos resolver apenas com procura?

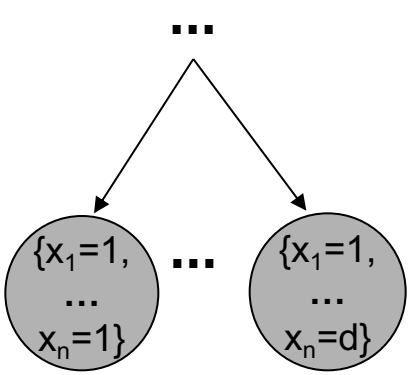
Procura Básica



n variáveis
d valores por variável

n.d estados

(n-1).d estados



d estados

- Total nós folha:
  - $n!.d^n$
- Mas só existem d^n atribuições completas
- Muito ineficiente ☹

Procura Básica

- Definição de um problema de satisfação de restrições como um problema de procura
  - Estados são caracterizados pelas atribuições feitas até ao momento
- Estado inicial: atribuição vazia { }
- Função sucessores: atribui um valor a uma variável não atribuída que não entra em conflito com a atribuição atual
  - → *falha* se não existem atribuições possíveis
- Teste objetivo: atribuição atual é completa

Procura Básica

- Esta formulação é comum a todos os CSPs
- Todas as soluções para *n* variáveis estão à profundidade *n*
  - procura em profundidade primeiro
- Caminho é irrelevante
  - estado final tem a solução

Procura com Retrocesso

- Será que podemos fazer melhor?
  - Atribuições a variáveis são comutativas, i.e.,
    - {X1 = 1, X2 = 2} é o mesmo que {X2 = 2, X1 = 1}
- Só é necessário considerar atribuições a uma única variável em cada nó
  - b = d e existem d^n nós folhas
- Procura em profundidade para CSPs com atribuição a uma única variável em cada nível
  - é denominada procura com retrocesso
- Procura com retrocesso é o algoritmo básico (não informado) para CSPs
  - Consegue resolver *n*-rainhas para $n \approx 25$

Procura com Retrocesso

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

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

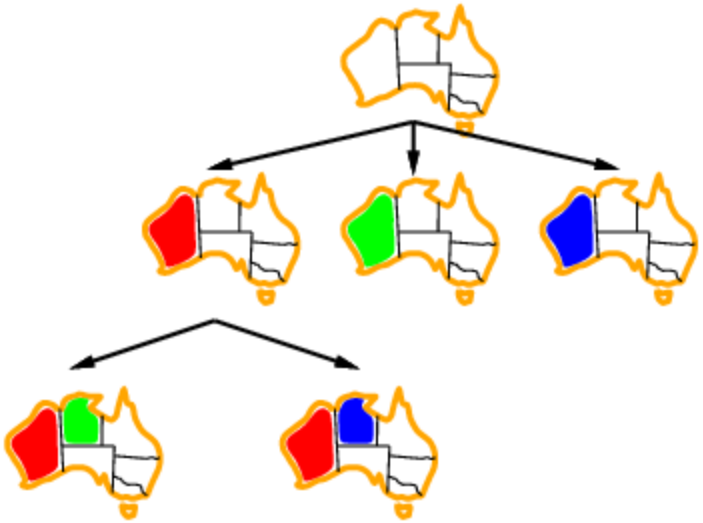
Procura com Retrocesso

- Versão mais simples
  - Select-Unassigned-Variable
    - escolher a próxima variável da lista de variáveis não atribuídas
  - Order-Domain-Values
    - Não ordenar nada, retornar a lista de valores pela mesma ordem recebida
  - Inference
    - Não fazer inferência, retornar um conjunto vazio de inferências

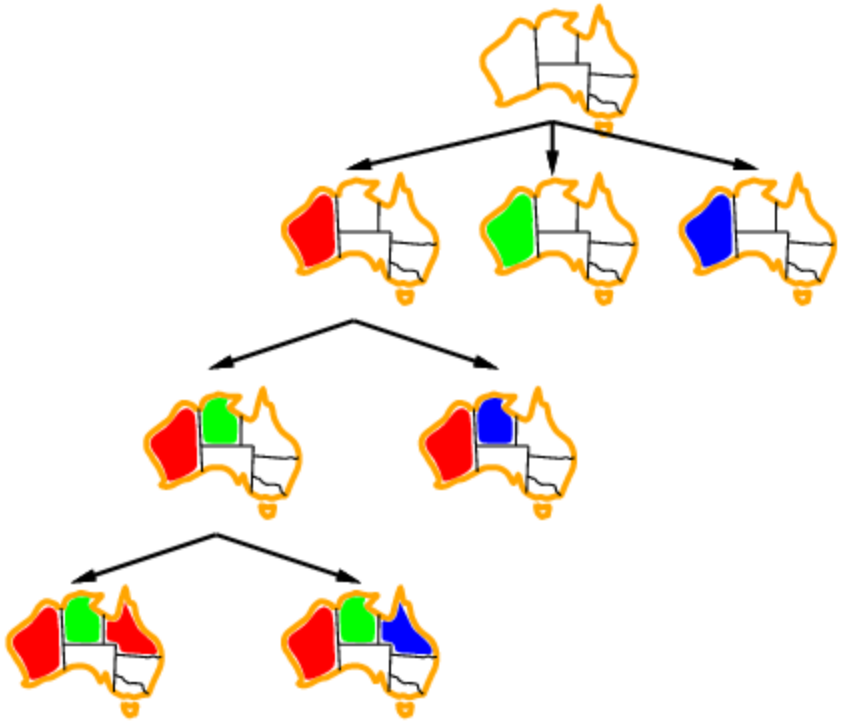
Retrocesso: exemplo

Retrocesso: exemplo

Retrocesso: exemplo



Retrocesso: exemplo



Procura com Retrocesso

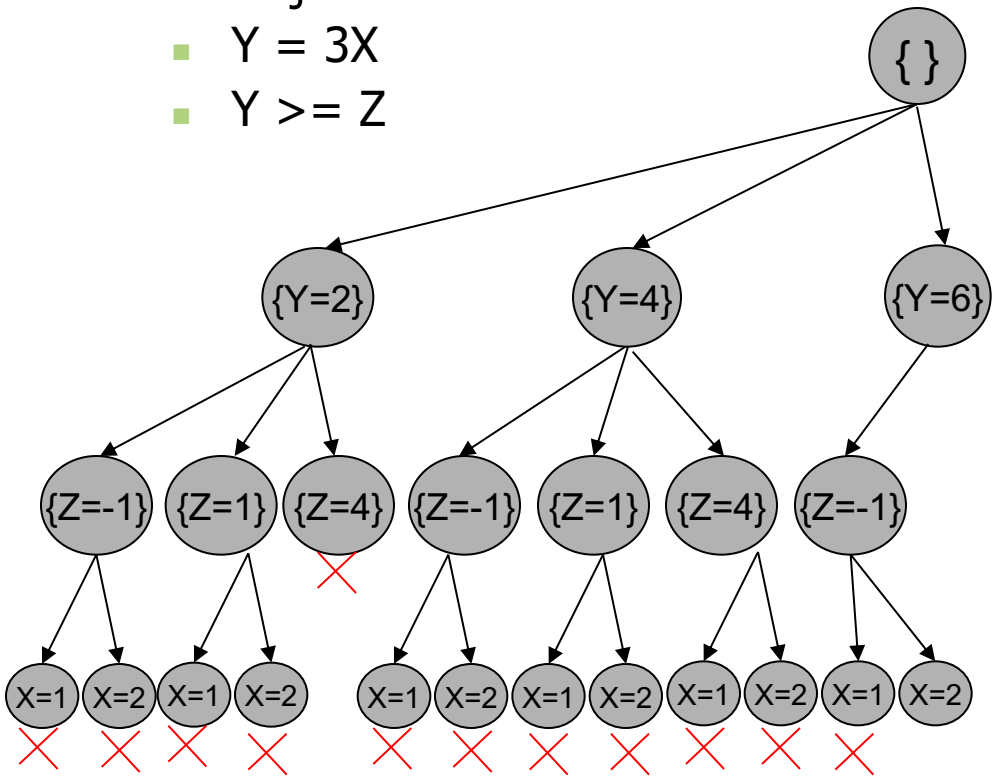
- Versão mais simples não é a mais eficiente
  - Pois é completamente cega
- Algoritmos de procura tradicional usam informação heurística para guiar procura
  - Heurísticas dependentes do domínio
- Ideia - usar informação heurística para guiar Procura com Retrocesso:
  - Escolher a próxima variável a experimentar
  - Escolher o próximo valor do domínio a experimentar
- Grande vantagem
  - Heurísticas independentes do domínio
  - Funcionam para qualquer problema de satisfação de restrições!!!

Escolher a próxima variável

- Que variável deve ser atribuída de seguida?
  - Ideia
    - Escolher a variável com maior probabilidade de causar uma falha
    - Quanto mais cedo detetarmos uma falha, mais cortes vamos fazer na árvore de procura

Escolher a próxima variável

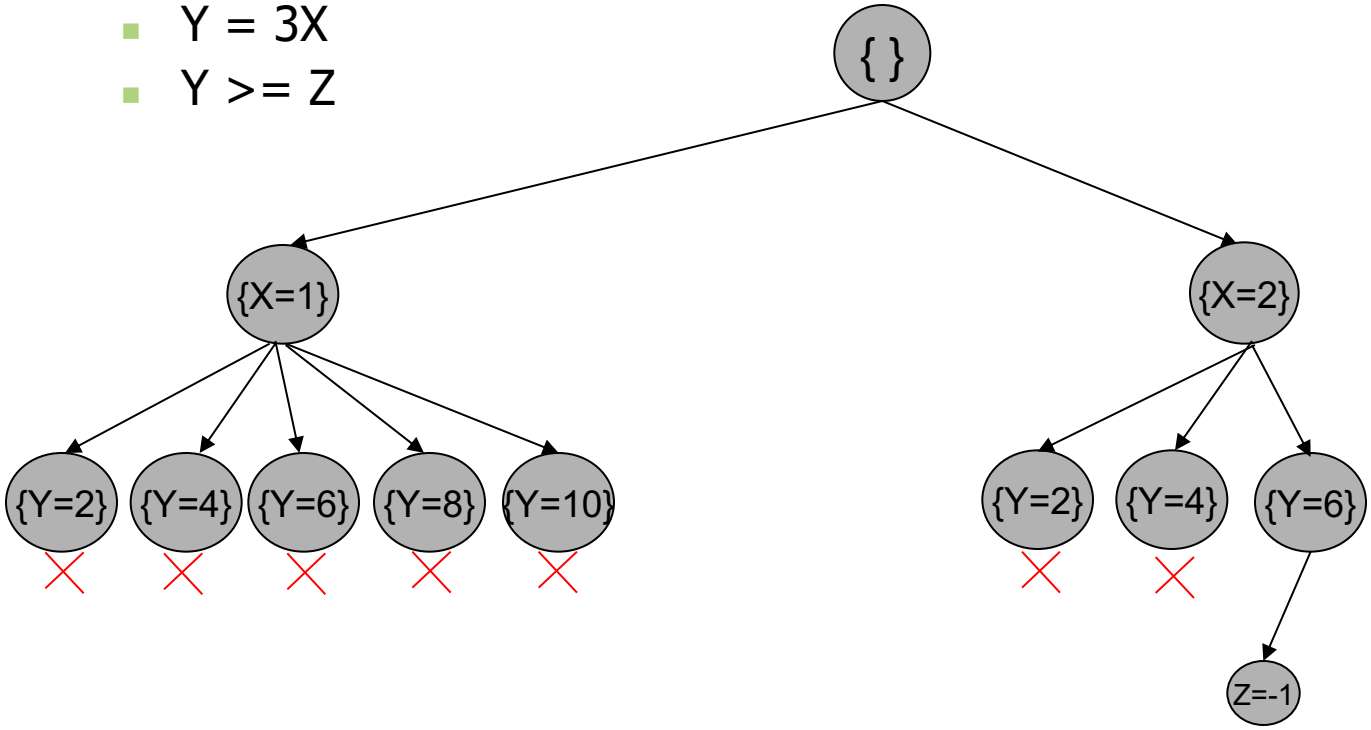
- Variáveis
  - X,Y,Z
- Domínio
  - Dx={1,2}, Dy = {2,4,6,8,10}, Dz = {-1,1,4}
- Restrições
  - Y = 3X
  - Y >= Z



22 testes
consistência

Escolher a próxima variável

- Variáveis
  - X,Y,Z
- Domínio
  - Dx={1,2}, Dy = {2,4,6,8,10}, Dz = {-1,1,4}
- Restrições
  - Y = 3X
  - Y >= Z



11 testes consistência

Escolher a próxima variável

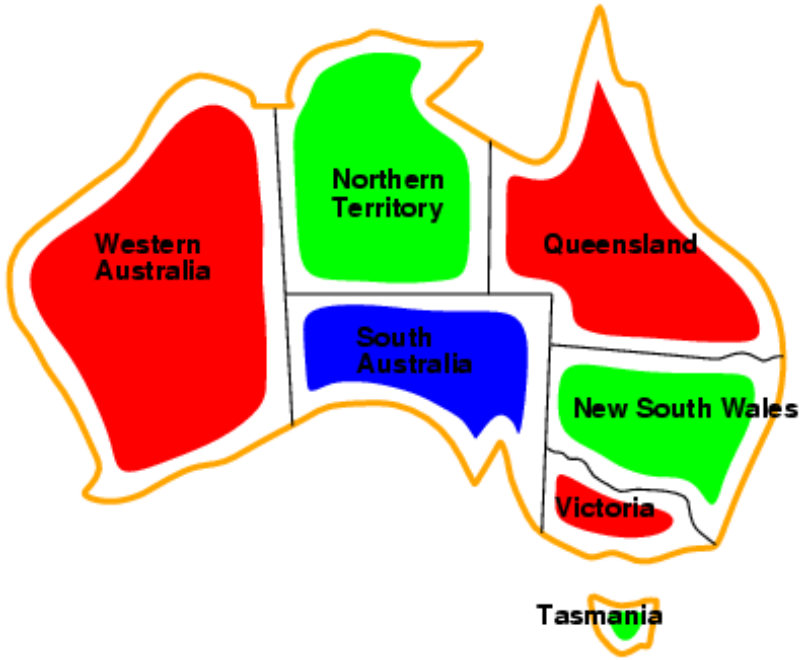
- Heurística dos valores remanescentes mínimos
  - Minimum Remaining Values (MRV) heuristic
  - Escolher a variável com o menor número de valores possíveis no domínio
    - Menos valores possíveis → Maior probabilidade falhar
  - Caso especial
    - Variável com domínio vazio
    - Escolhida imediatamente
      - Falha detetada imediatamente
  - Pode chegar a ser 1000x melhor que ordenação estática/aleatória para certos problemas

Escolher a próxima variável

- Por vezes a heurística MRV não funciona
  - Ex: colorir o mapa
    - Inicialmente todas as regiões têm 3 valores possíveis para a cor.
- Que fazer?
  - Tentar reduzir o fator de ramificação de decisões futuras
    - olhando para variáveis que estão envolvidas em mais restrições

Escolher a próxima variável

- Heurística do maior grau *(Degree Heuristic)*
  - Escolhe a variável que está envolvida no maior número de restrições com variáveis ainda **não atribuídas**
  - South Australia é adjacente a todas as outras regiões!



Escolher a próxima variável

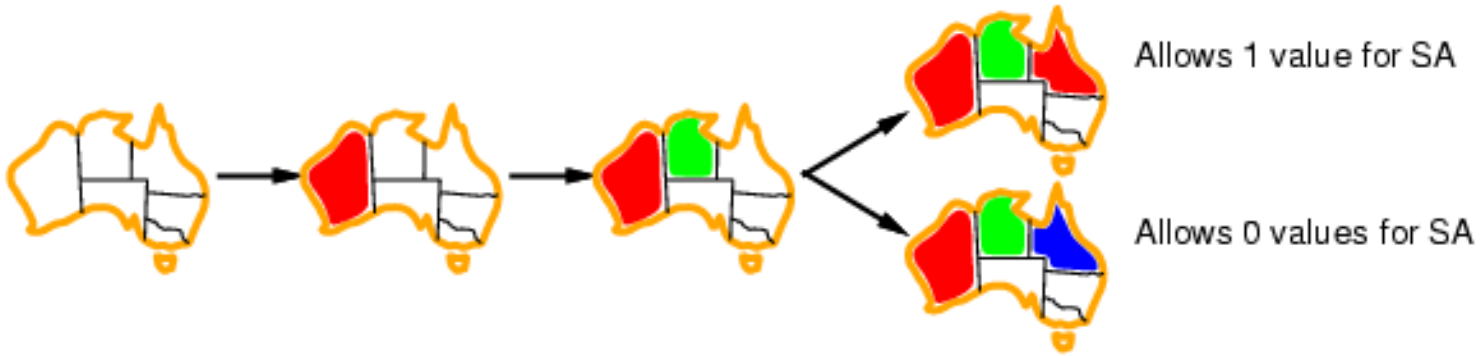
- A heurística MRV é normalmente mais eficaz que a heurística de maior grau
  - Heurística MRV usada em primeiro lugar
  - Heurística de maior grau normalmente usada para desempatar variáveis com mesmo valor MRV

Escolher o próximo valor

- Uma vez escolhida a variável a atribuir
  - É necessário escolher qual o valor a experimentar primeiro
  - No entanto, enquanto na escolha de variável o critério era falhar o mais rápido possível
- Na escolha de valor o critério é falhar o menos possível
  - A escolha do valor a experimentar primeiro não vai influenciar cortes no espaço de estado
    - Porque para haver backtracking (voltar para trás) é preciso experimentar todos os valores para uma variável
  - Como só queremos uma única solução
    - Então devemos experimentar primeiro os valores com maiores hipóteses de corresponder a uma solução
  - Se quiséssemos todas as soluções
    - A ordem dos valores não interessaria para nada

Escolher o próximo valor

- Heurística do valor menos restritivo
  - Least-constraining-value
- Dada uma variável, escolher o valor que tem menos restrições:
  - Valor que elimina menos valores no domínio das outras variáveis



- A combinação destas heurísticas permite resolver o problema das 1000-rainhas

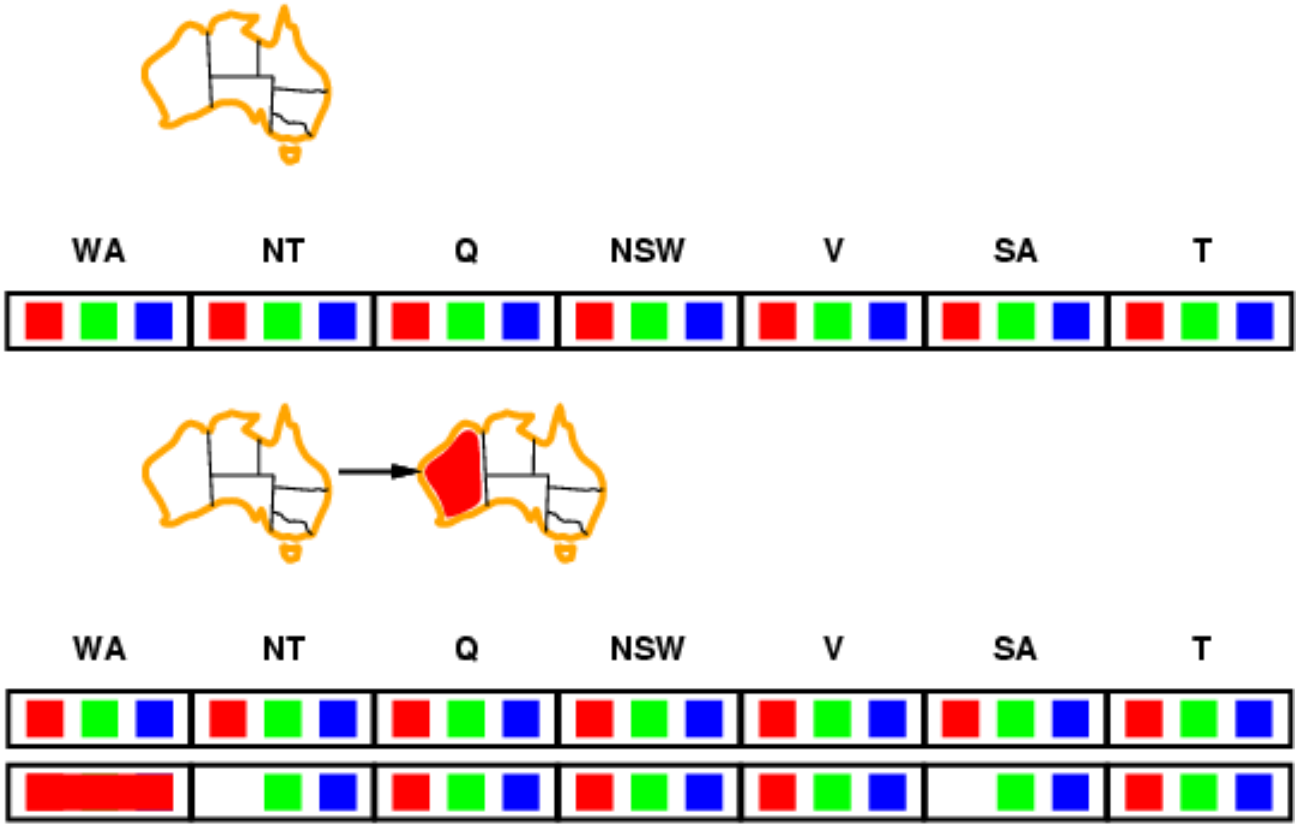
Procura e Inferência

- Para resolver CSP podemos usar
  - Inferência
  - Procura
- Mas porque não misturar as duas?
  - Escolhemos um valor possível para uma variável
  - Aplicamos inferência/propagação de restrições
  - Se não funcionar, voltamos para trás, escolhemos outro valor para a variável e tentamos de novo

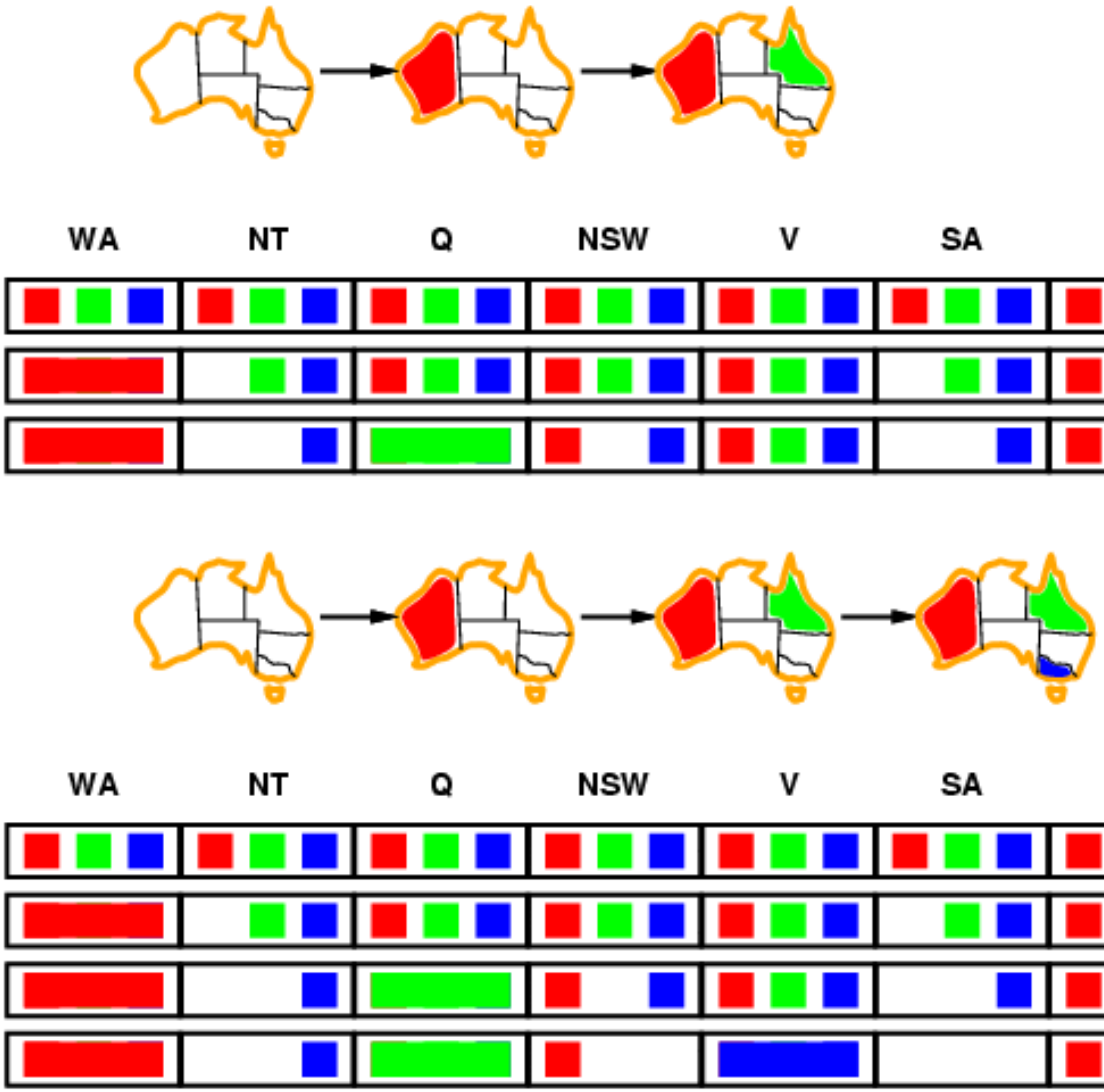
Procura e Inferência

- Forward checking:
  - Verificação posterior/olhar em frente
  - Forma mais simples de inferência
  - Quando uma variável X é atribuída
    - Tornar as outras variáveis consistentes em arco para X
      - Para cada variável Y ligada a X por uma restrição
      - Apagar do domínio de Y todos os valores inconsistentes com a atribuição feita para X
    - Se o domínio de uma variável ficar {}
      - Então a atribuição não é consistente
      - Temos de escolher outro valor para a variável
      - Ou então fazer backtracking para outra variável

Forward checking



Forward checking



Forward checking

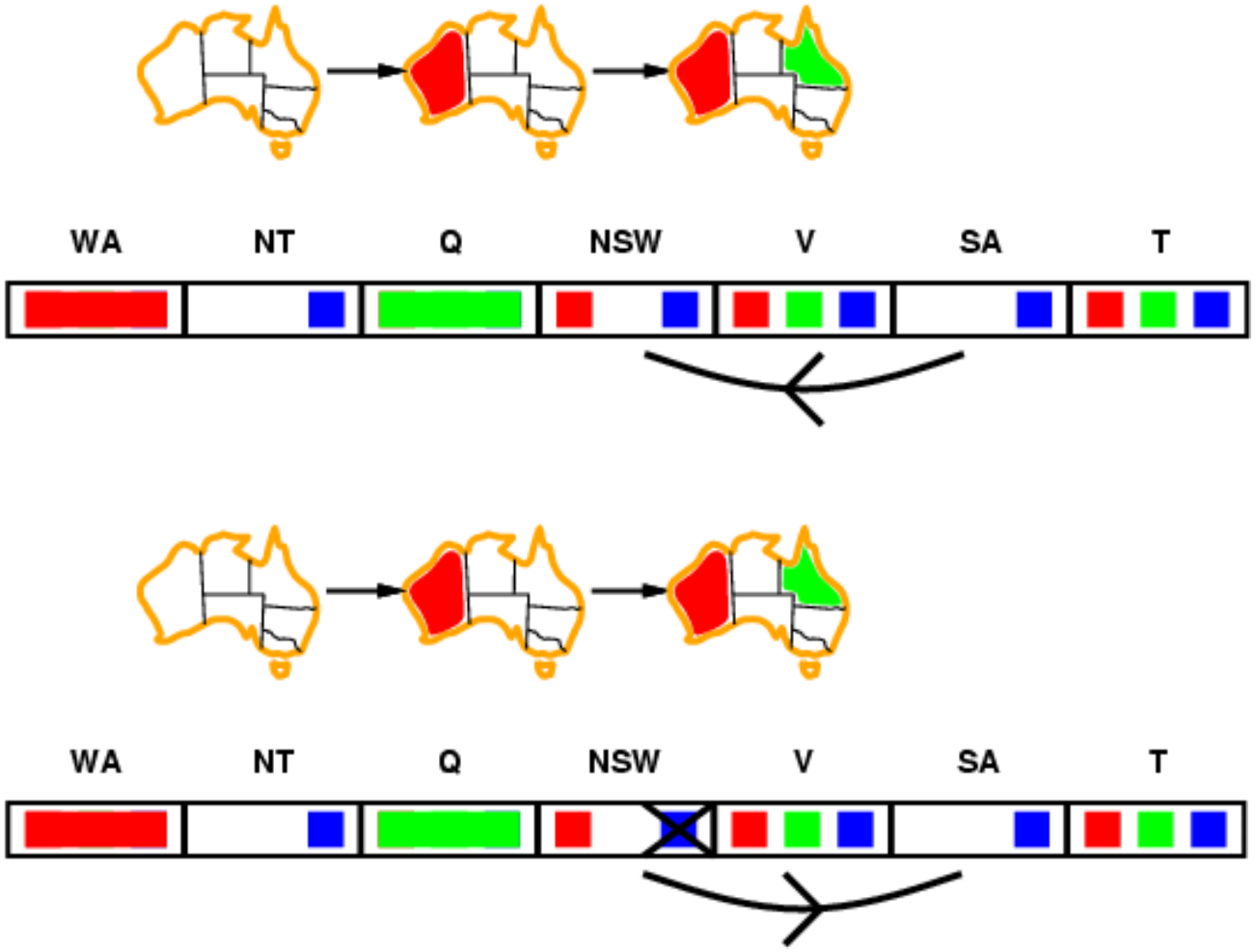
- Forward checking deteta muitas inconsistências
  - Mas não deteta todas
- Torna todas as variáveis consistentes em arco para X
  - Mas não as torna consistentes em arco umas para as outras
  - Não consegue detetar certas falhas antecipadamente

- Ao atribuir verde à variável Q
  - Reduzimos o domínio de NT e SA
  - Mas NT e SA não podem ser ambos azuis!

Maintaining Arc Consistency

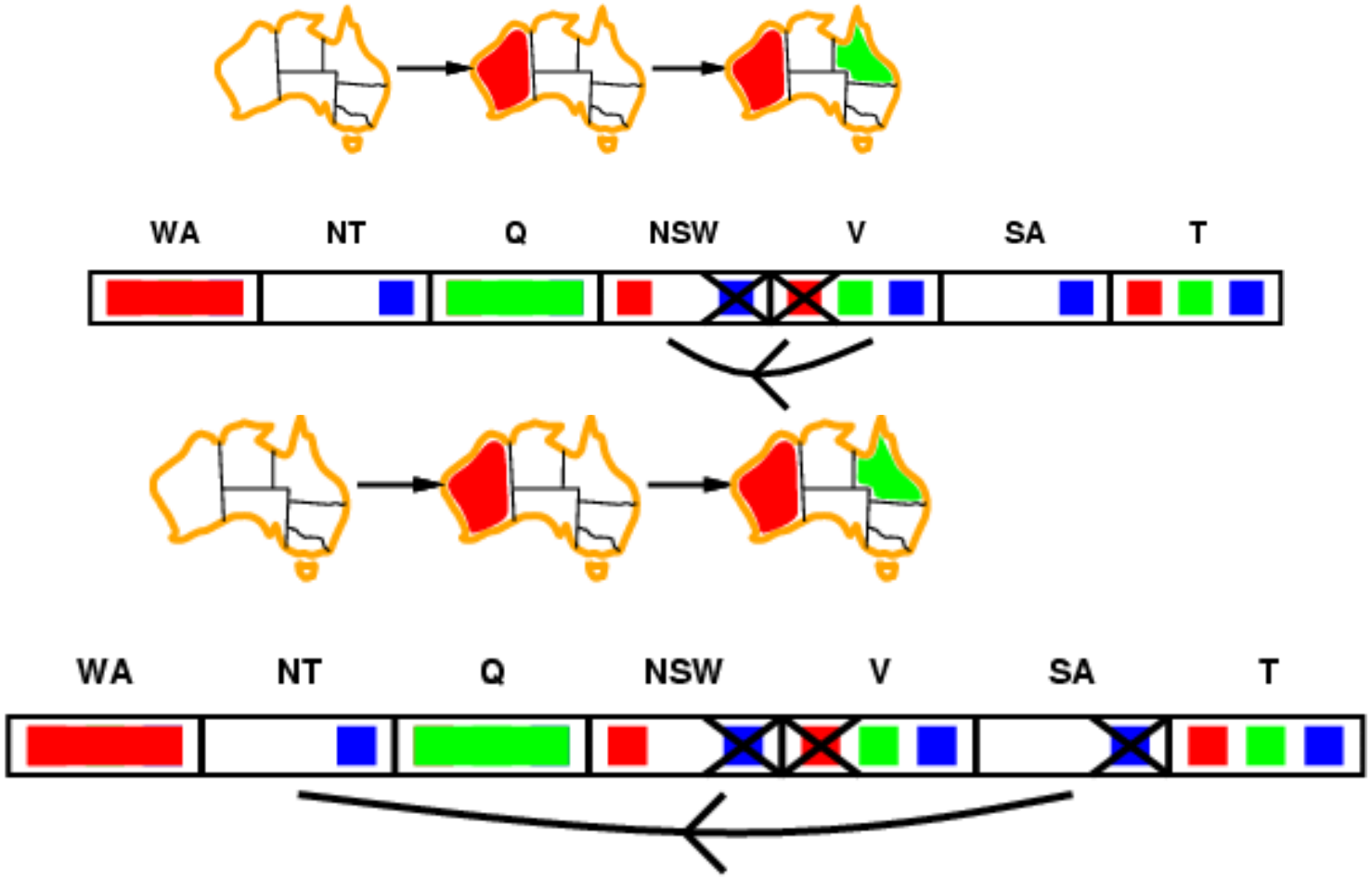
- Maintaining Arc Consistency (MAC)
  - Algoritmo que garante consistência de arco entre todas as variáveis
  - Sempre que uma variável Xi é atribuída
    - O procedimento Inference chama o AC3
    - Mas em vez de inicializarmos o algoritmo com todos os arcos
    - Usamos todos os arcos (Xj,Xi) tais que Xj corresponde a uma variável ainda não atribuída que é vizinha de Xi
    - Fazemos consistência de arco de Xj para Xi (tal como forward …)
      - Caso especial de Xi já estar atribuída
        - Testa-se cada valor domínio Xj contra valor Xi
    - Mas só paramos de fazer consistência quando todos os domínios estabilizarem

Maintaining Arc Consistency

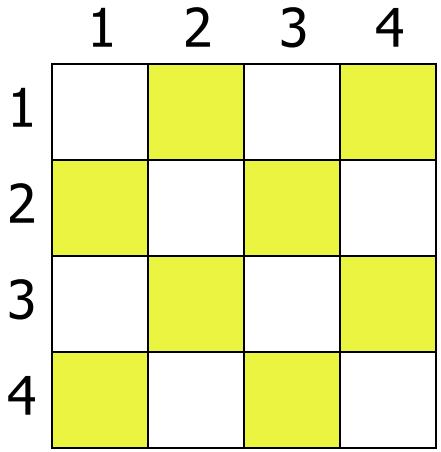


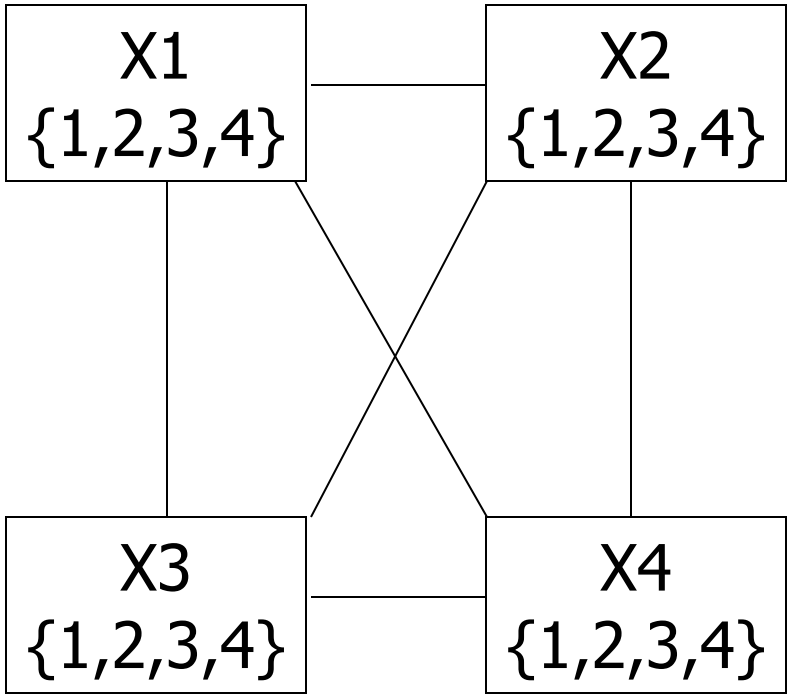
Maintaining Arc Consistency

- Se *X* perde um valor, então os vizinhos de *X* têm de ser revistos



Retrocesso + Forward Checking

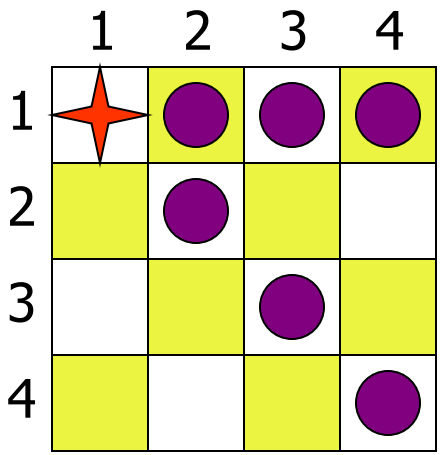


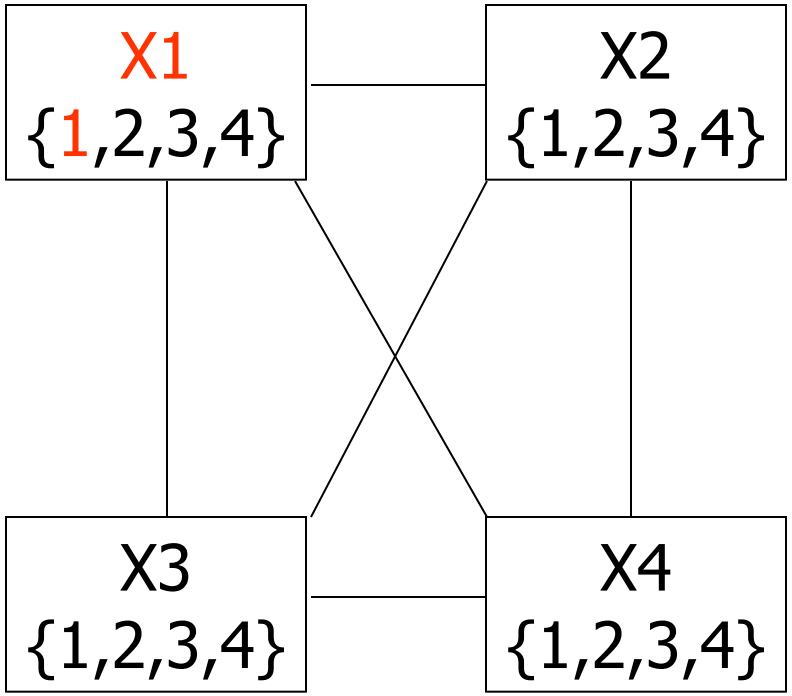


Variáveis {X1,X2,X3,X4}
Domínio X1 = Dom X2 = Dom X3 = Dom X4 = {1,2,3,4}
Xj = i significa que rainha j está na posição (i,j)

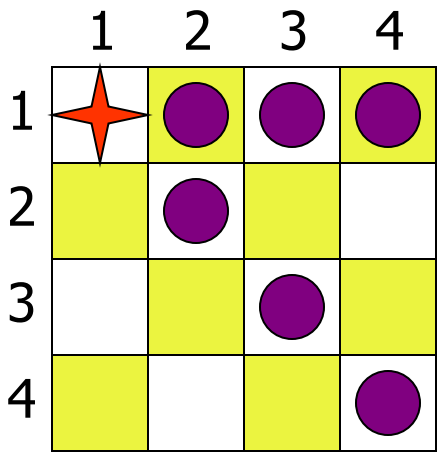
[Slides de B.J. Dorr - CMSC 421 course on AI]

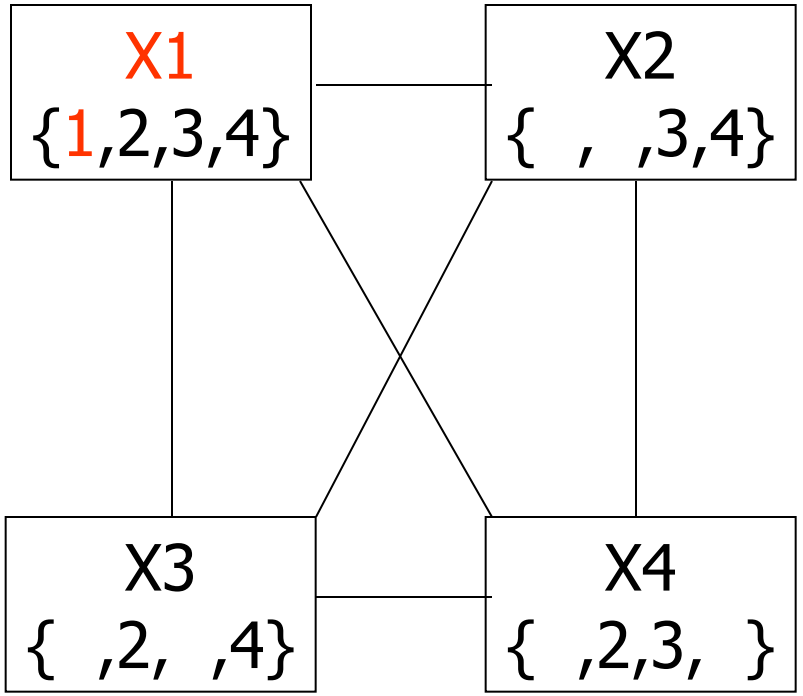
Exemplo: 4-Rainhas



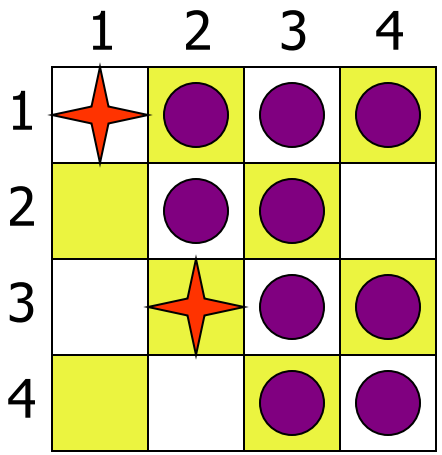


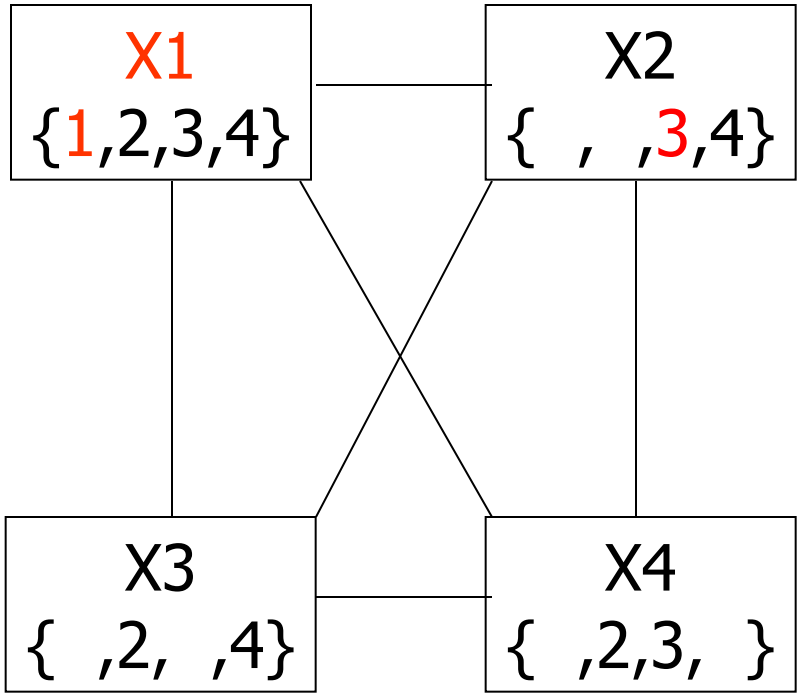
Exemplo: 4-Rainhas



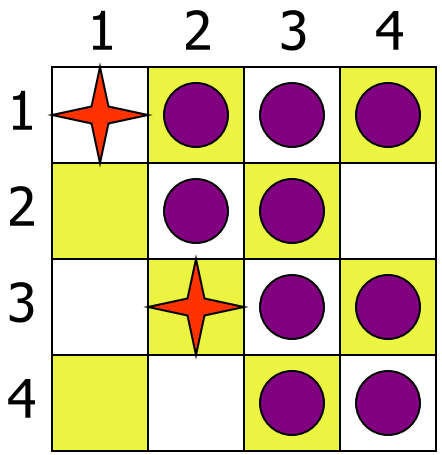


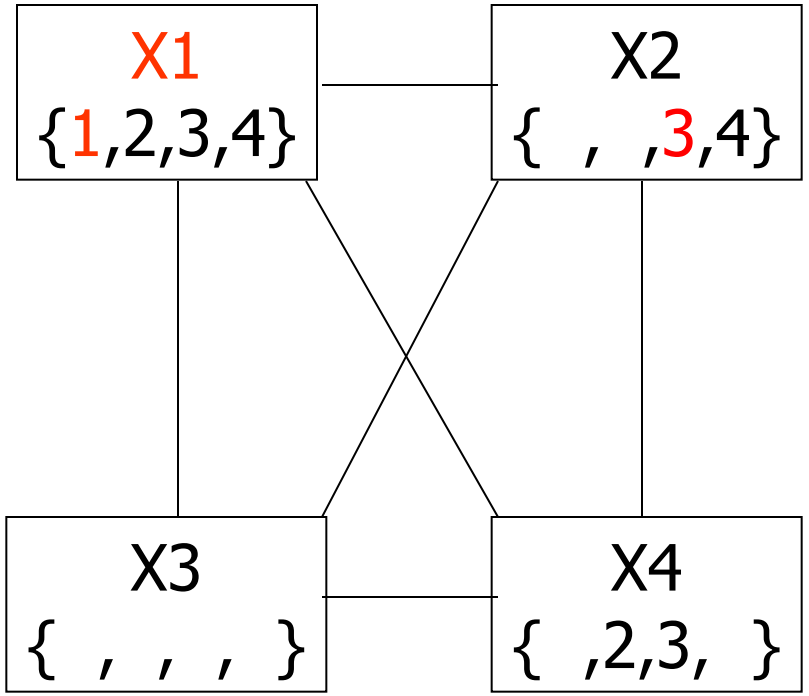
Exemplo: 4-Rainhas



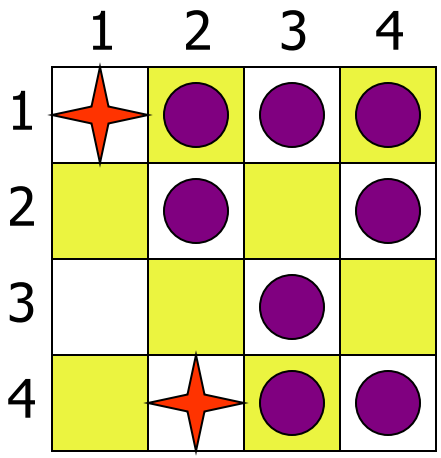


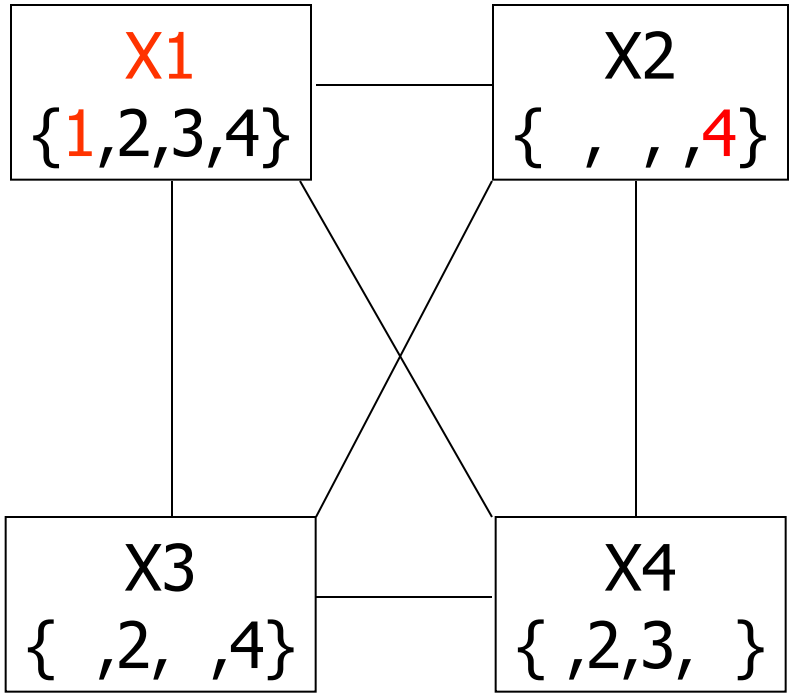
Exemplo: 4-Rainhas



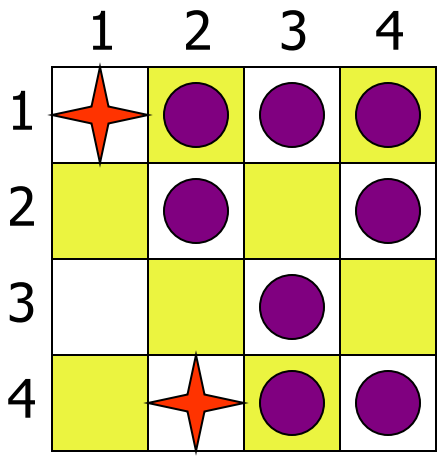


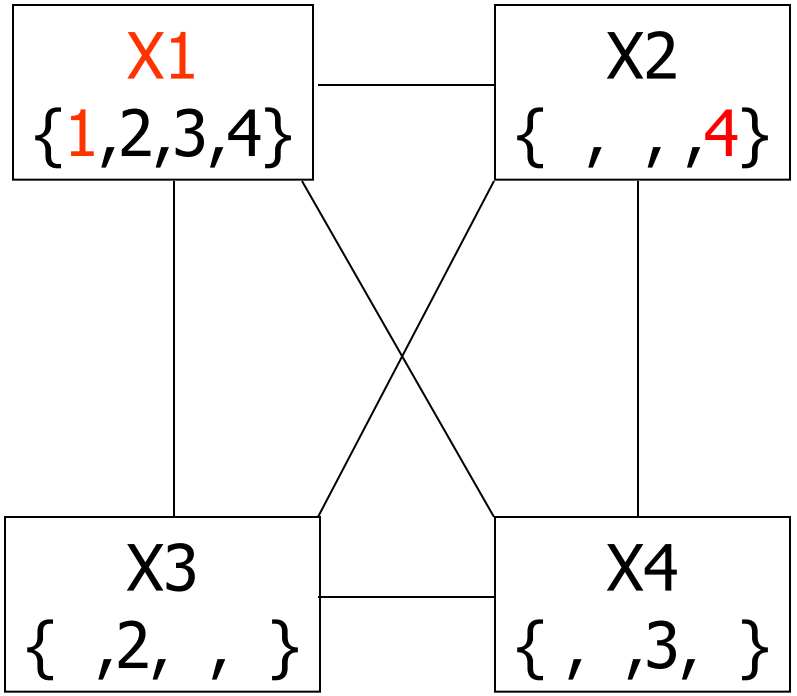
Exemplo: 4-Rainhas





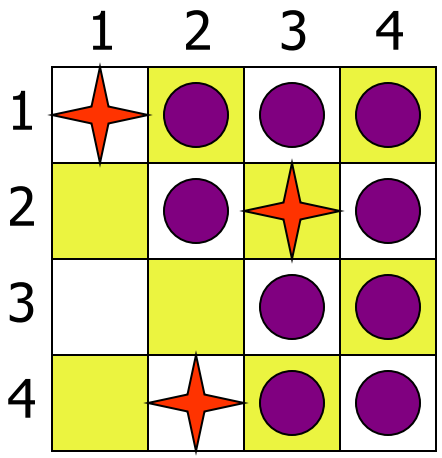
Exemplo: 4-Rainhas

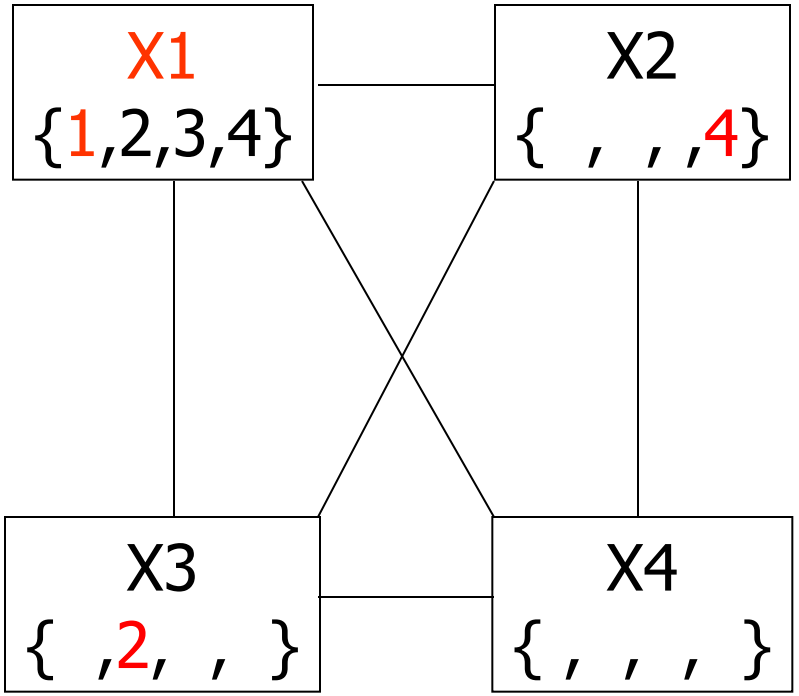




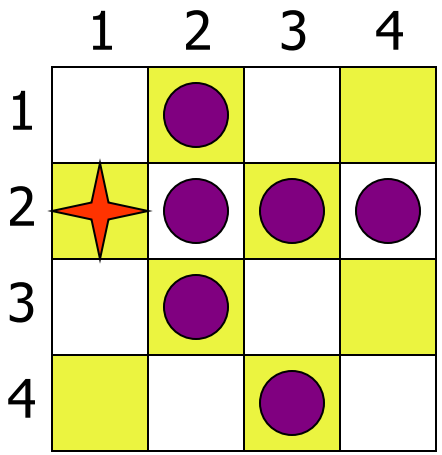
Consistência de arcos já teria detectado inconsistência!

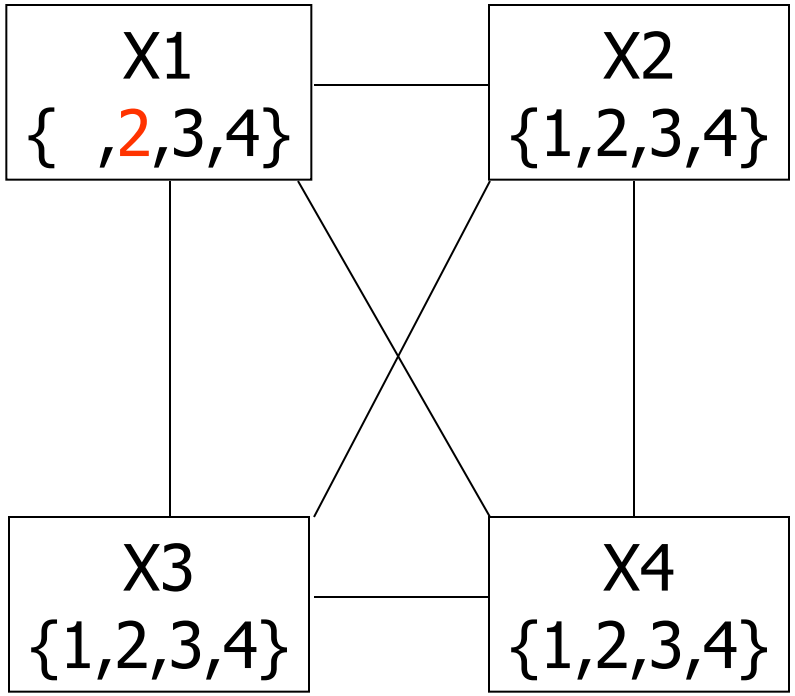
Exemplo: 4-Rainhas



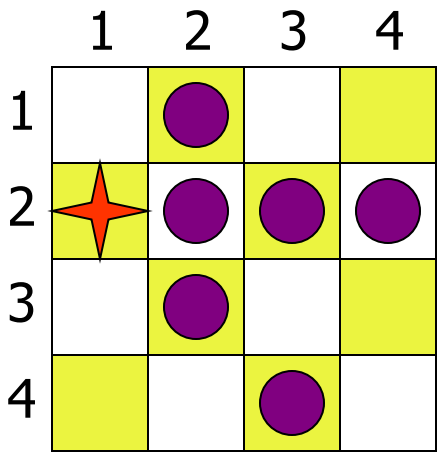


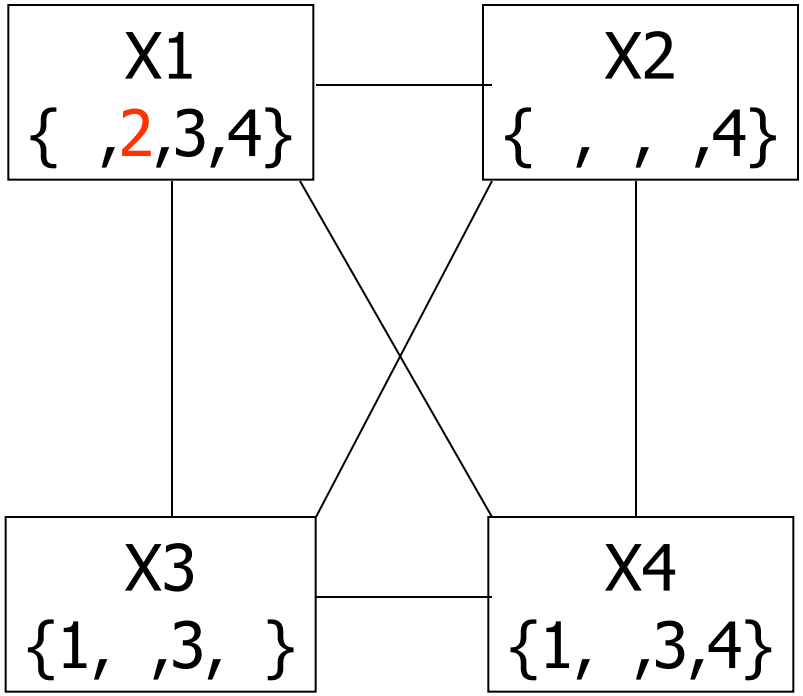
Exemplo: 4-Rainhas



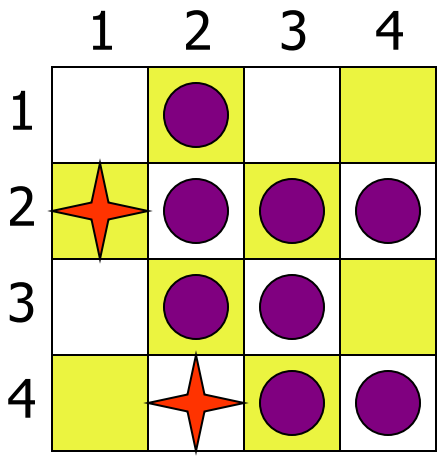


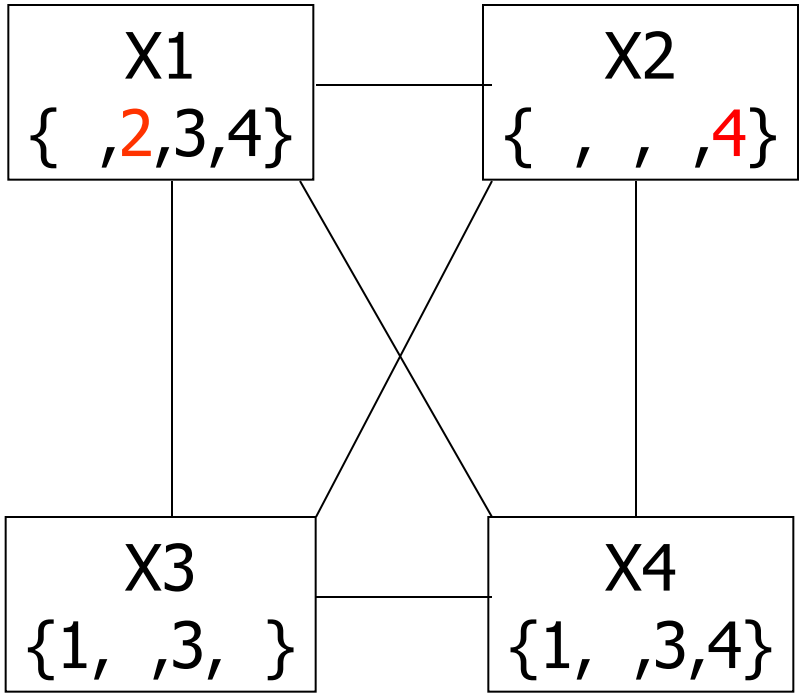
Exemplo: 4-Rainhas



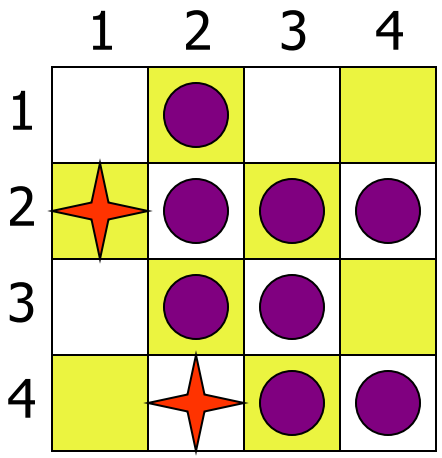


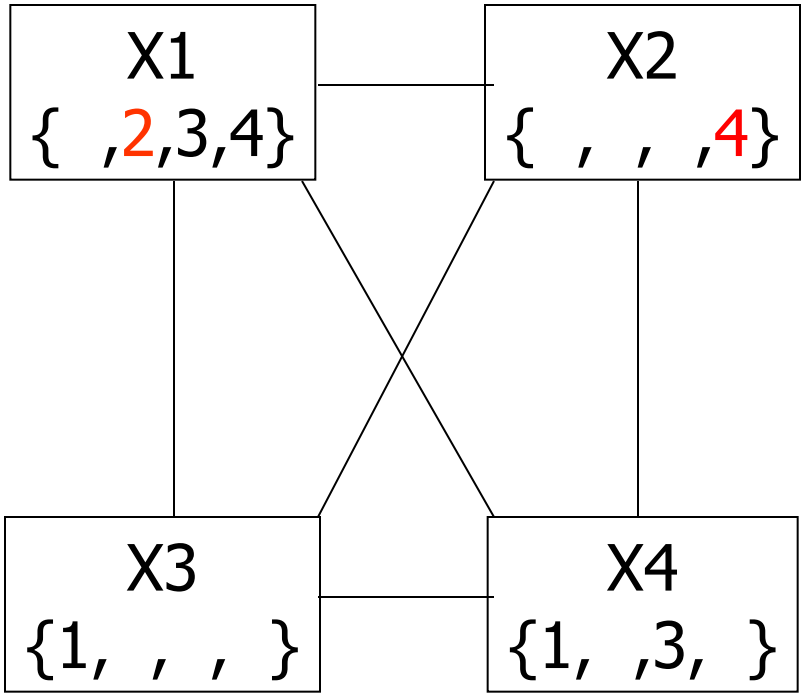
Exemplo: 4-Rainhas





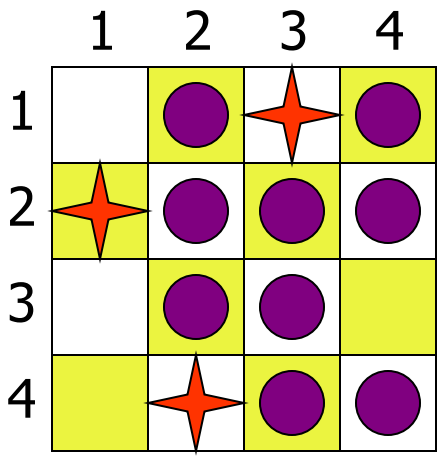
Exemplo: 4-Rainhas

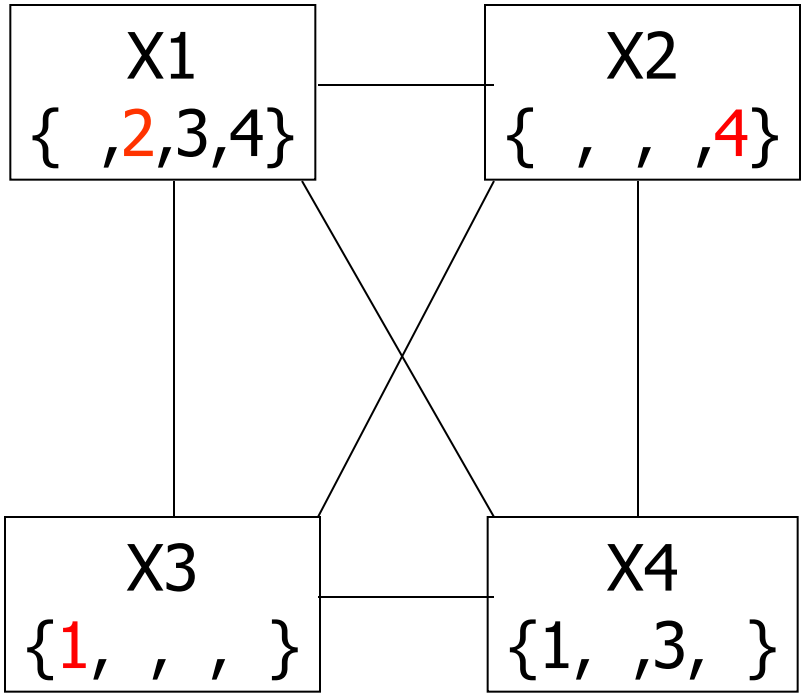




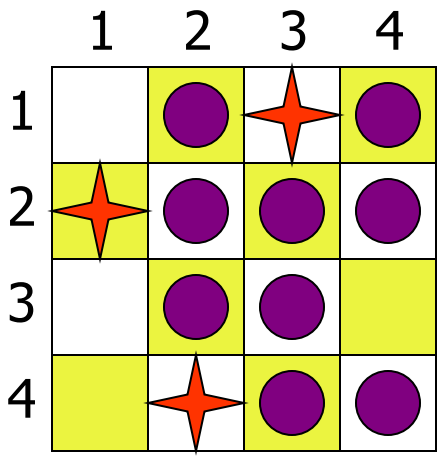
Consistência de arcos já teria identificado a solução!

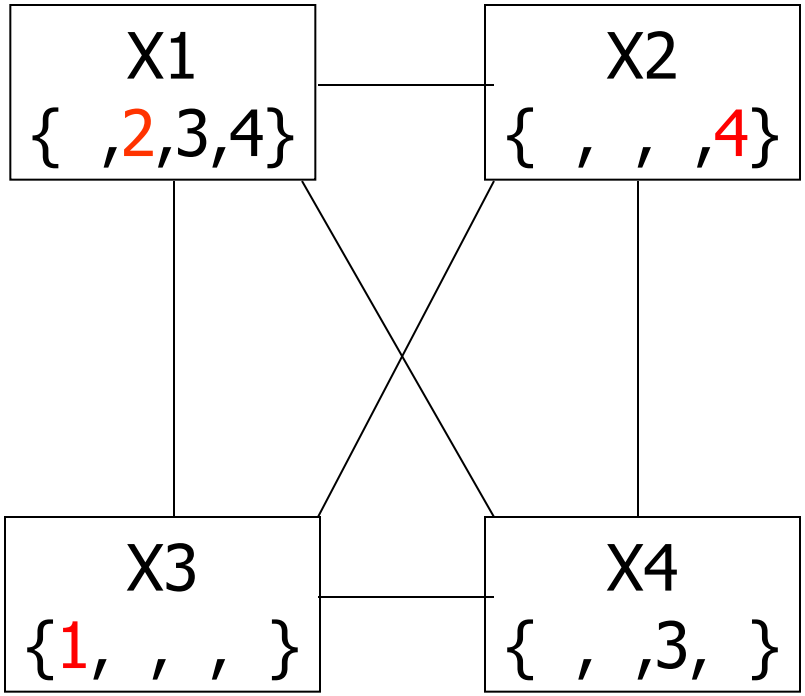
Exemplo: 4-Rainhas



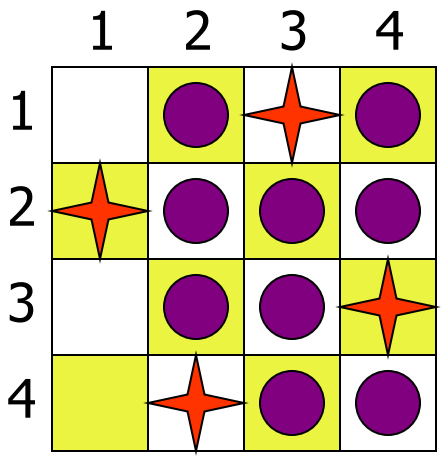


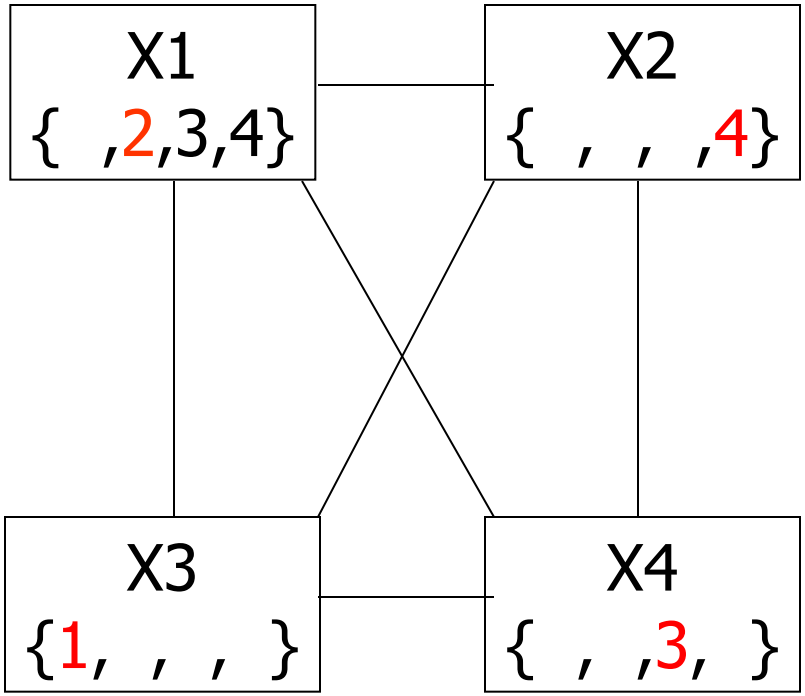
Exemplo: 4-Rainhas





Exemplo: 4-Rainhas





Retrocesso inteligente

Retrocesso com salto (Backjumping):

- Q = vermelho, NSW = verde,
  V = azul, T = vermelho,
  SA = … conflito!
  - Retrocesso → alterar valor de T
  - Mas T não foi responsável pelo conflito!
- Conjunto de conflito de uma variável X
  - Conjunto de atribuições que estão em conflito com algum valor para a variável X, i.e. que eliminaram valores do domínio de X
  - Conj de conflito para SA = {Q=vermelho,NSW=verde,V=azul}
  - Retroceder "inteligentemente" para a variável que consta no conjunto de conflito e foi atribuída mais recentemente, i.e. V
- Forward checking é equivalente a backjumping
  - Ou se usa um ou o outro

Retrocesso inteligente

Outro exemplo:

- WA = vermelho, NSW = vermelho, T = vermelho
- De seguida atribuir NT, Q, V, SA
  - Esgotamos cores possíveis para NT...
  - Retrocedemos para T?
  - Mas faria sentido retroceder para NSW!
- Conjunto de conflito tem de ir para além de relações diretas...

Retrocesso inteligente

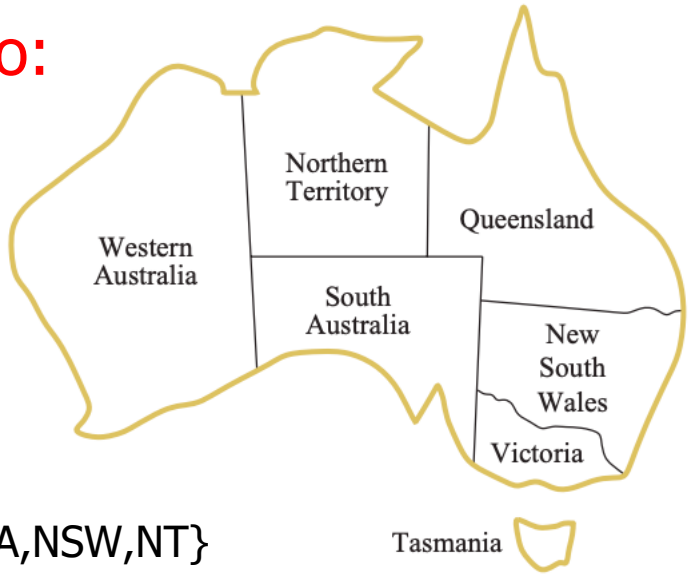
Retrocesso com salto dirigido ao conflito:

- Conflict-directed backjumping
- Definição mais completa de conj. conflito para uma variável X
  - Conjunto de variáveis atribuídas antes de X, tal que
    - Juntamente com outras variáveis que serão atribuídas mais à frente
    - Vão fazer com que X não tenha nenhum valor consistente
- Calcular conj. conflito
  - conf(X) inicializado {}
  - Sempre que testamos uma atribuição para X e uma restrição entre X e Y falhe, adicionar Y ao conf(X).
  - Xj – variável atual; conf(Xj) – conj. conflito para Xj
  - Se todos os valores para Xj falham, retroceder para a variável mais recente Xi em conf(Xj)
  - Alterar conf(Xi) ← conf(Xi) U conf(Xj) – {Xi}

Retrocesso inteligente

Retrocesso com salto dirigido ao conflito:

- e.g. WA=vermelho, NSW=vermelho, T=vermelho, NT=azul, Q=verde, V, SA
  - SA falha
    - conf(SA)={WA,NT,Q}
  - Retrocesso inteligente para Q
    - conf(Q) inicialmente {NSW,NT}
    - Alterado para conf(Q) U conf(SA) – {Q} = {WA,NSW,NT}
  - Retrocesso inteligente para NT
    - conf(NT) inicialmente {WA}
    - Alterado para conf(NT) U conf(Q) – {NT} = {WA,NSW}
  - Retrocesso para NSW!



Retrocesso Inteligente

- O retrocesso inteligente não evita que o mesmo conflito venha a aparecer novamente noutro ramo da árvore, e.g.
  - NT = vermelho, WA = vermelho, NSW = vermelho
  - NT = azul, WA = vermelho, NSW = vermelho
- Repetição de erros pode ser evitada com aprendizagem!

Aprendizagem de Restrições

- Aprendizagem de Restrições
  - Constraint Learning
  - Quando chegamos a uma contradição
  - Encontrar o conjunto mínimo de variáveis do conjunto de conflitos que está a causar o problema
  - Este conjunto de variáveis e os seus valores – **no-good**
  - no-goods devem ser registados e lembrados
    - Ou como restrições adicionais
    - Ou com utilização de lista de no-goods
  - Adicionar uma restrição que não permita que volte a acontecer WA = vermelho $\wedge$ NSW = vermelho
    - WA ≠ vermelho $\vee$ NSW ≠ vermelho

Procura Local para CSPs

- Procuras locais muito eficazes para resolver muitos CSPs
- Usar algoritmos que usam estados "completos", i.e. todas as variáveis atribuídas
- Para aplicar procura local a CSPs:
  - Permitir estados em que não são satisfeitas todas as restrições
  - Transições entre estados consiste na re-atribuição de valores a variáveis

Procura Local para CSPs

- Ordenação de variáveis: selecionar aleatoriamente qualquer variável para a qual exista um conflito

- Seleção de valores com a heurística menor número de conflitos (min-conflicts):
  - Escolher valor que viola o menor número de restrições
  - Se existirem vários valores nestas condições escolher um deles aleatoriamente

Min-Conflicts

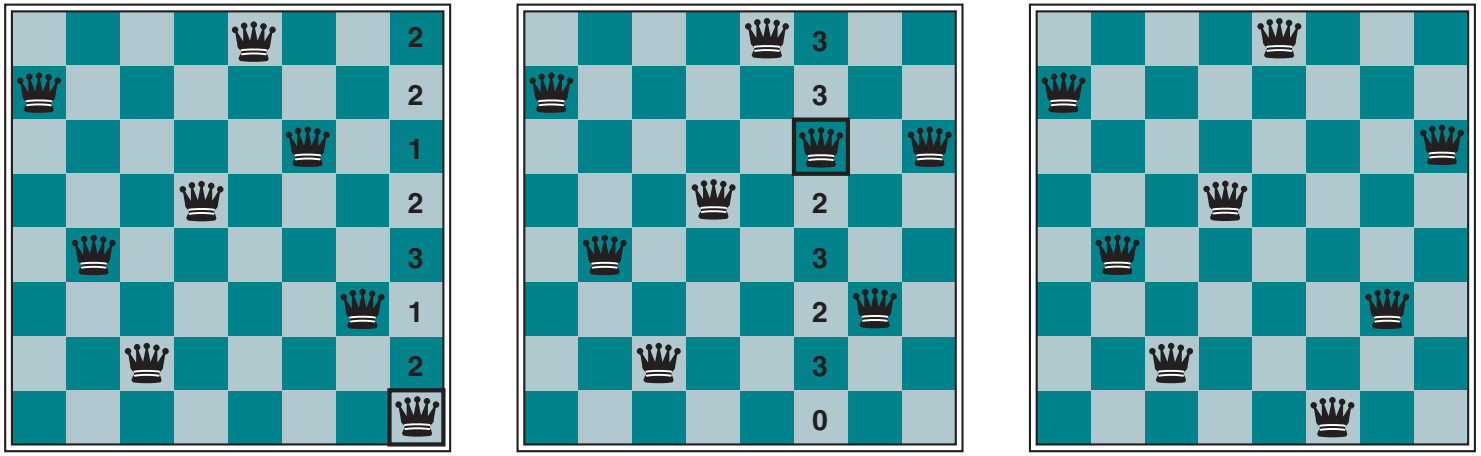
function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or *failure*
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for *csp*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value *v* for *var* that minimizes CONFLICTS(*csp*, *var*, *v*, *current*)
    set *var* = *value* in *current*
  return *failure*

Exemplo: 4-Rainhas

- Estado: 4 rainhas em 4 colunas (4^4 = 256 estados)

Exemplo: 8-Rainhas



- Neste exº, 2 iterações suficientes para resolver o problema
- Em cada iteração é escolhida uma rainha que esteja em conflito para mudar de posição
- Nova posição minimiza o número de ataques (local e globalmente)
- Escolha aleatória entre posições que minimizam de igual modo o número de ataques

Procura Local para CSPs

- Min-conflicts eficiente para muitos CSPs
  - n-rainhas
    - Dado um estado inicial aleatório, existe uma grande probabilidade de resolver o problema das *n*-rainhas em tempo quase constante para *n* arbitrário (e.g., *n* = 10.000.000)
    - n-rainhas é fácil para procura local porque as soluções estão densamente distribuídas pelo espaço de estados
    - só nos interessa encontrar uma solução qualquer

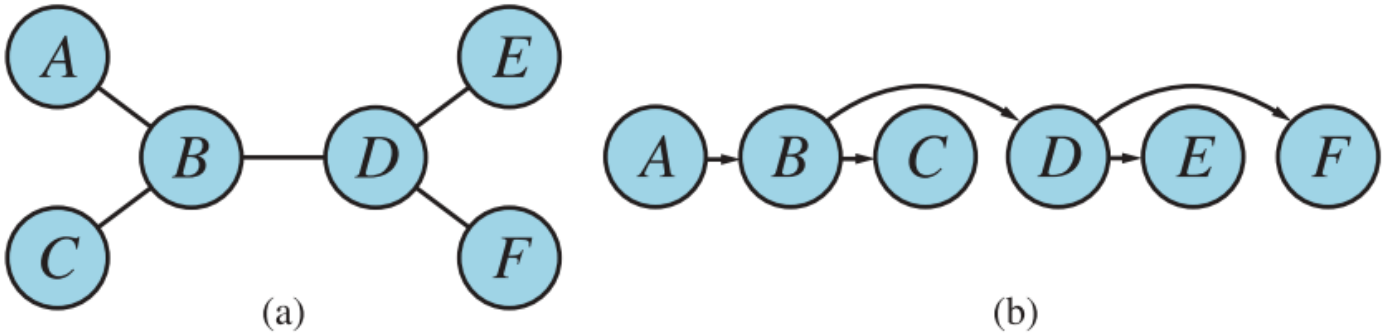
Procura Local vs. Retrocesso

- Vantagens
  - Encontra soluções para problemas de grandes dimensões (10.000.000 rainhas)
  - Tempo de execução da heurística do menor número de conflitos está pouco dependente da dimensão do domínio
- Desvantagens
  - Não permite provar que não há solução porque não mantém um registo dos estados já visitados

Estrutura de Problemas

- A estrutura de um problema, obtida através da representação em grafo, pode ser usada para facilitar a resolução do problema
- É importante detetar sub-problemas: melhorias no desempenho
  - Que decisões afetam outras decisões?
  - E.g. não existe nenhuma relação entre Tasmânia e as outras regiões

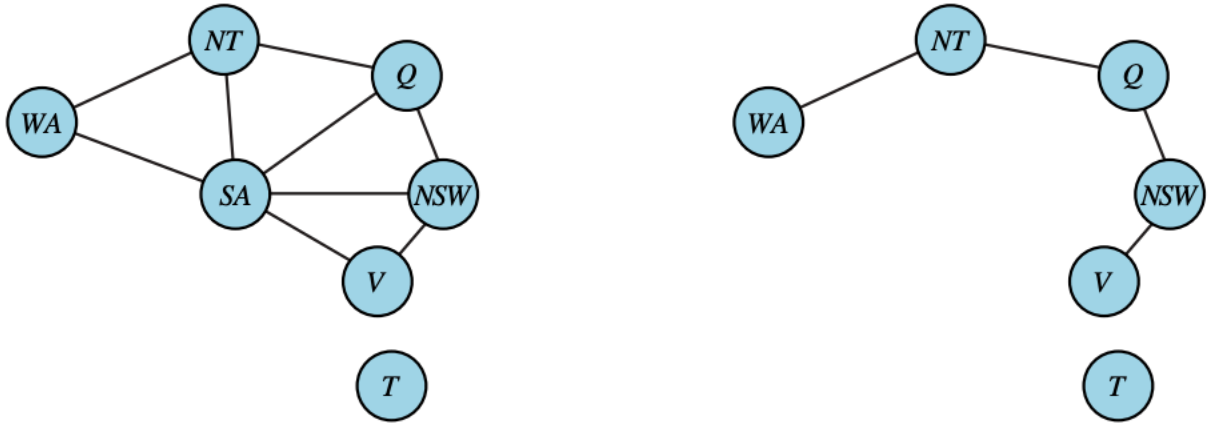
Estrutura em árvore



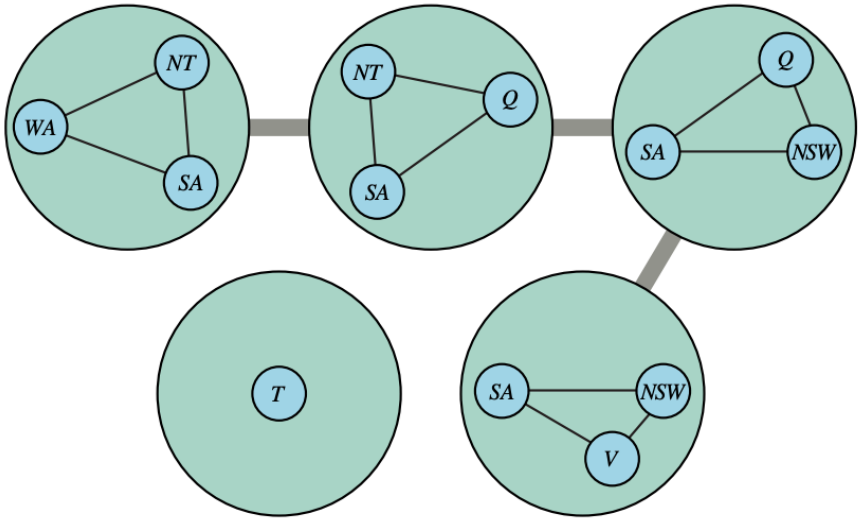
- Teorema: se um CSP não tem ciclos, então pode ser resolvido em tempo $O(nd^2)$ em vez de $O(d^n)$
- Tipicamente um CSP que tem uma estrutura em árvore pode ser resolvido em tempo linear no número de variáveis

Aproximação para Árvore

- Podemos transformar um CSP numa estrutura em árvore adaptando o problema: remoção e colapsagem de nós
- Remoção de nós: atribuir valores a algumas variáveis t.q. variáveis não atribuídas formem uma árvore
  - Compensa se o número de variáveis a atribuir é pequeno
  - Identificar estas variáveis é NP-difícil! → uso de aproximações
  - E.g. atribuir SA



Aproximação para Árvore



- Colapsagem de nós
  - Problema resultante é uma árvore cujos nós são sub-problemas
  - Cada sub-problema é resolvido separadamente
  - Soluções resultantes são combinadas

Conclusões

- CSPs são um tipo especial de problemas:
  - Estados definidos por valores atribuídos a um conjunto específico de variáveis
  - Teste objectivo definido a partir de restrições nos valores das variáveis
- Retrocesso = procura em profundidade primeiro com uma variável atribuída por cada nível de profundidade
- Ordenação de variáveis e valores é importante
- Forward checking evita atribuições que garantidamente levarão a conflitos no futuro
- Propagação de restrições (e.g. consistência de arcos) deteta inconsistências adicionais
- Retrocesso inteligente identifica fonte do conflito
- Procura local + h. menor número de conflitos é eficiente
- Conhecimento da estrutura do problema pode melhorar desempenho

Vídeos

- https://www.youtube.com/watch?v=TXJ-k9ljDo0
- https://www.youtube.com/watch?v=4LAMiuNd6LI
- https://www.youtube.com/watch?v=YTfcDTsyQHU
- https://www.youtube.com/watch?v=X6m0DXt95bs