

# 制約充足問題

Chapter 5

Section 1 – 3

# 概要

- 制約充足問題 (CSP)
- CSPでの後戻り探索
- CSPでの局所探索

# 制約充足問題(CSPs)

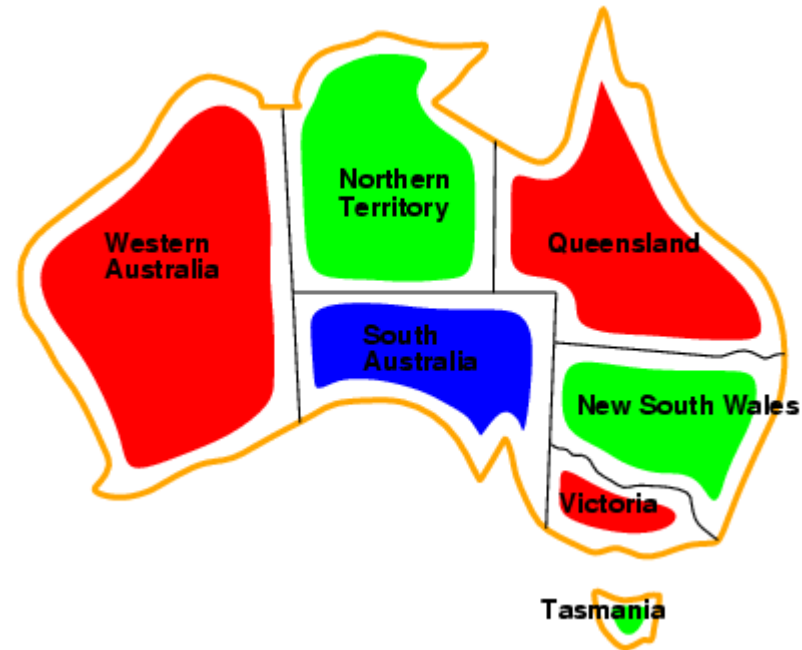
- 標準的な探索問題:
  - 状態は“ブラックボックス”である – 後継関数、発見的関数、ゴールテストが使えるいかなるデータ構造でも良い
- CSP:
  - 状態は領域  $D_i$ からの値を持つ変数  $X_i$ で表される
  - ゴールテストは制約の集合で、これは変数の部分集合に対して許される値の組み合わせを表している
- 形式表現言語はCSPの簡単な例
- 標準的な探索アルゴリズムよりもっと協力で使いやすい汎用的なアルゴリズムを用いることができる

# 例：地図の色分け



- Variables  $WA, NT, Q, NSW, V, SA, T$
- Domains  $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
- e.g.,  $WA \neq NT$ , or  $(WA, NT)$  in  $\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$

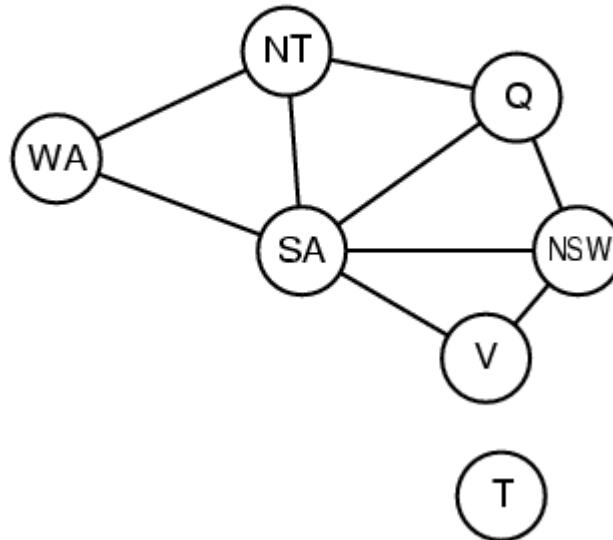
# 例：地図の色分け



- 解は**完全**で**一貫性**のある割り当てである。即ち WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

# 制約グラフ

- **バイナリー制約のCSP:** それぞれの制約は2つの変数に関する
- **制約グラフ:** ノードは変数で、アークは制約である



# CSPの多様性

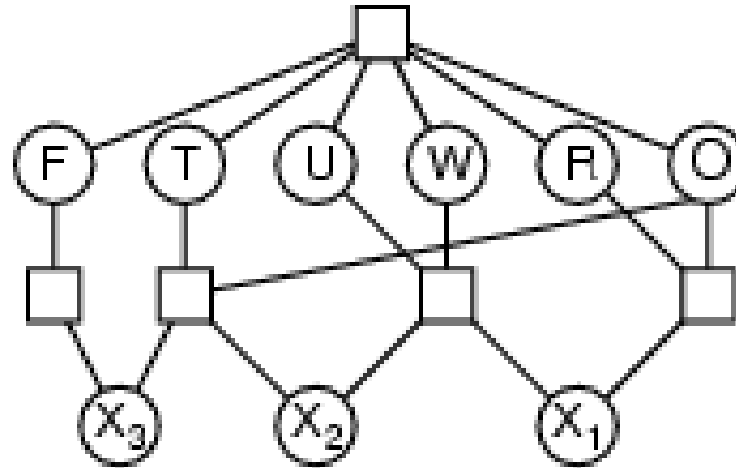
- 離散変数
  - 有限領域:
    - $n$ 変数、領域の大きさ $d \rightarrow O(d^n)$  の完全な割り当て
    - 例えばブーリアン充足問題
  - 無限領域:
    - 整数、文字列など
    - ジョブスケジューリング。変数はそれぞれのジョブに対して開始と終了の日付を持つ
    - 制約充足言語が必要。例:  $StartJob_1 + 5 \leq StartJob_3$
- 連続変数
  - 例: ハッブル望遠鏡の開始、終了時間
  - 線形制約は線形計画法により多項式時間で解くことが可能

# CSPの多様性

- ユーナリー制約は一変数を含む
  - e.g.,  $SA \neq \text{green}$
- バイナリー制約は二変数の対を含む
  - e.g.,  $SA \neq WA$
- 高階制約は3変数以上を含む
  - e.g., cryptarithmic column constraints

# Example: Cryptarithmic

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



- **Variables:**  $F T U W$   
 $R O X_1 X_2 X_3$
- **Domains:**  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- **Constraints:** *Alldiff* ( $F, T, U, W, R, O$ )
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F, T \neq 0, F \neq 0$

# 現実世界のCSP

- 割り当て問題
  - どの教員がどのクラスを教えるか
- 時刻表の問題
  - どのクラスがいつどこで教えられるか
- 輸送のスケジュール
- 工場のスケジューリング
- 多くの現実世界の問題は現実の値をもつ変数も含んでいる

# 標準的な探索の定式化 (増分)

直接的な方法から始めて、修正する

状態はこれまでに値が割り当てられているものとする

- **初期状態**: 空の割当  $\{\}$
  - **後継関数**: 割り当てられていない変数に現在の割当に矛盾しない値を割り当てる
    - 正しい割当ができないとき失敗とする
  - **ゴールテスト**: 現在の割当が完全か
1. これはすべてのCSPに共通である
  2. もし $n$ 変数であれば、全ての解は深さ $n$ で現れる
    - 深さ優先探索を利用
  3. 経路は無関係である。完全な状態の形式化を使うことが可能
  4. 深さ $l$ で分岐数は $b = (n - l)d$ 。従って $n! \cdot d^n$ の葉

領域での  
値の個数

# 後戻り探索

- 変数割当は可換。例:  
[ WA = red then NT = green ] same as [ NT = green then WA = red ]
- それぞれのノードで一変数を割り当てることを考える  
→  $b = d$  で  $d^n$  の葉がある
- 一つの変数の割当を行うCSPに対する深さ優先探索は後戻り探索
- 後戻り探索はCSPに対する知識に基づかない基本的な探索
- $n \approx 25$  に対して  $n$ クイーンを解くことができるか

# 後戻り探索

```
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
```

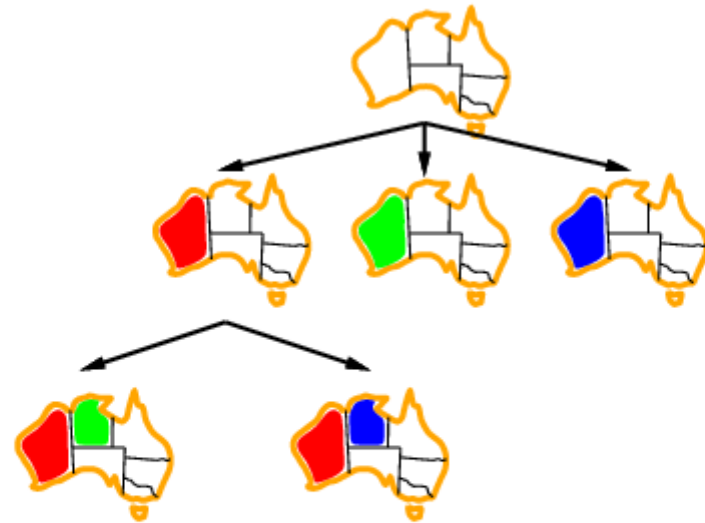
# 後戻り探索の例



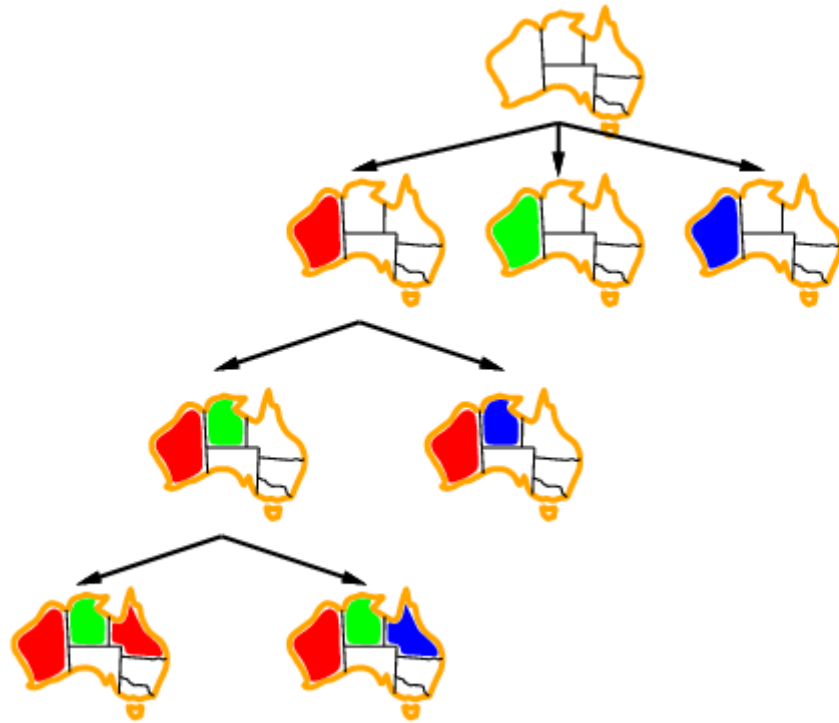
# 後戻り探索の例



# 後戻り探索の例



# 後戻り探索の例

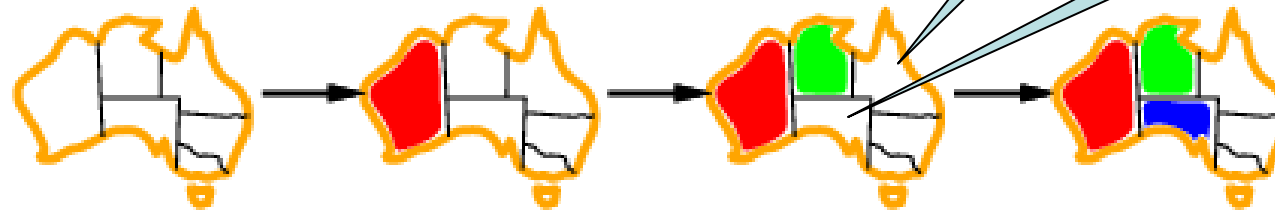


# 後戻り探索の効率を向上

- **汎用的**な方法は速度を飛躍的に改善:
  - どの変数が割り当てられるべきか
  - どの順番で値が試みられるべきか
  - 必ず失敗に導くものを早い時期に見つけることができるか

# 最も制約された変数

- 最も制約された変数:  
割当可能な値が最小の変数を選ぶ



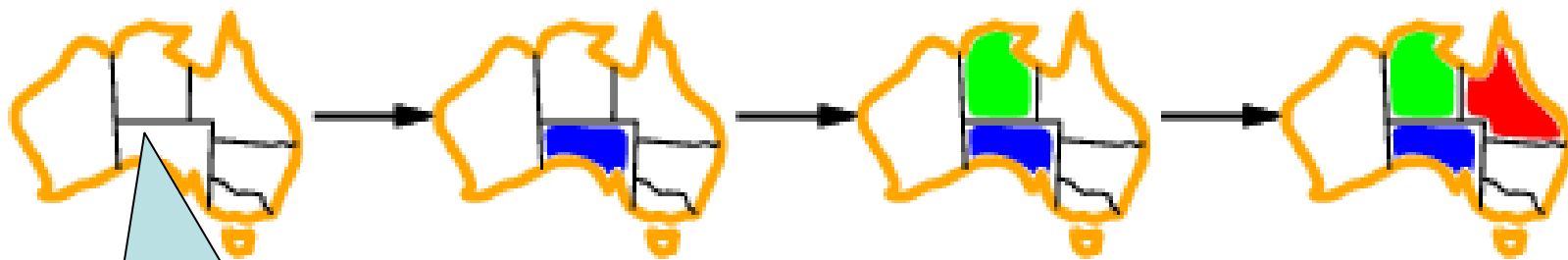
正しい値  
の数が2

正しい値  
の数が1

- 別名：最小残余値(MRV)での発見的手法

# 最も制約している変数

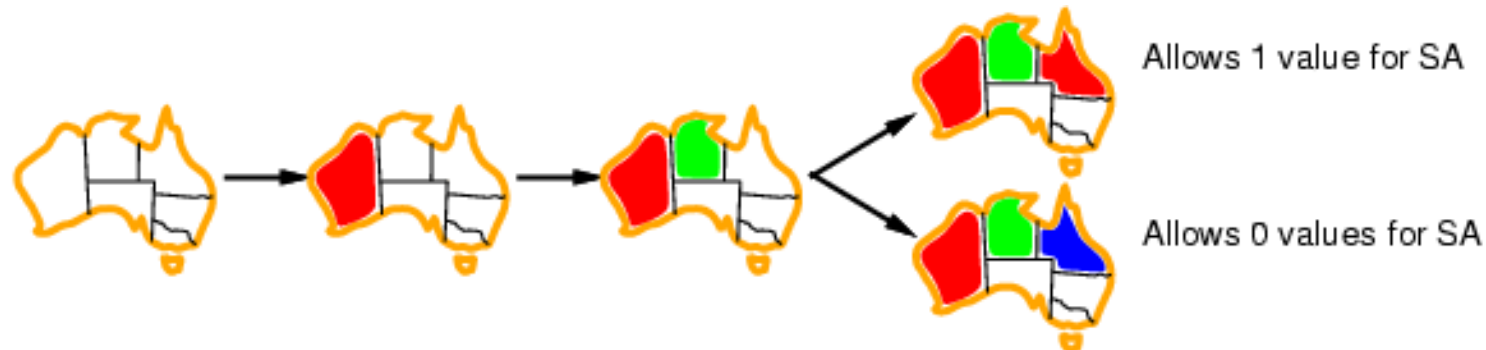
- 最も制約された変数がいくつかあったときの  
タイプブレーク
- 最も制約している変数:
  - 残る変数に最も制約を与える変数を選ぶ



ここを選ぶと、他  
は全て割当可能  
な値は2となる

# 最も制約されていない値

- 変数が与えられたとき、最も制約しない値を選ぶ：
  - この後の変数の割当てで最も柔軟性が高いもの



- これらの発見的方法を組み合わせることで1000クイーンまで可能になる

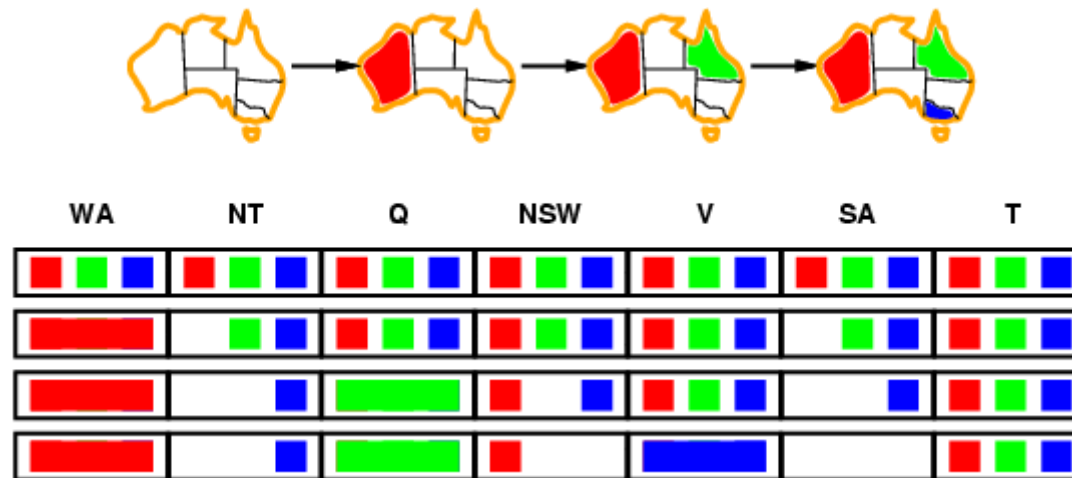






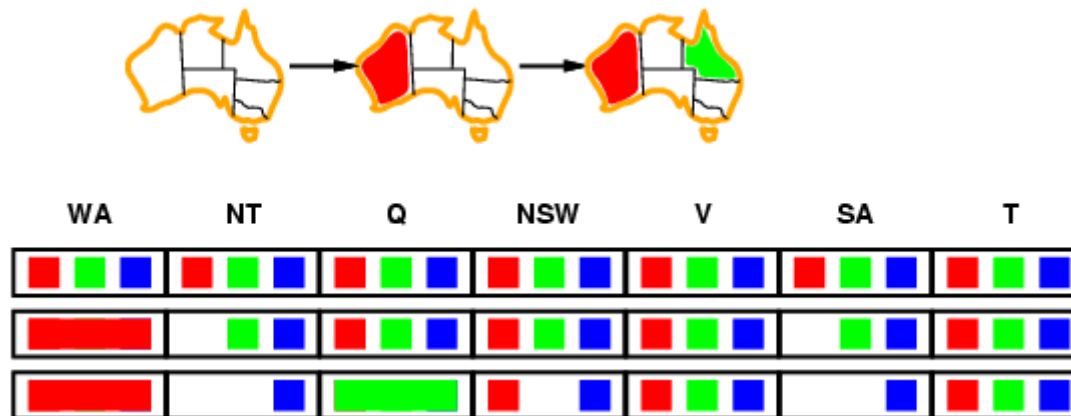
# 前方向の検査

- 考え方:
  - 割り当てられていない変数に残っている正しい値を追跡する
  - ある変数が正しい値を持たなくなったとき探索を停止する



# 制約伝播

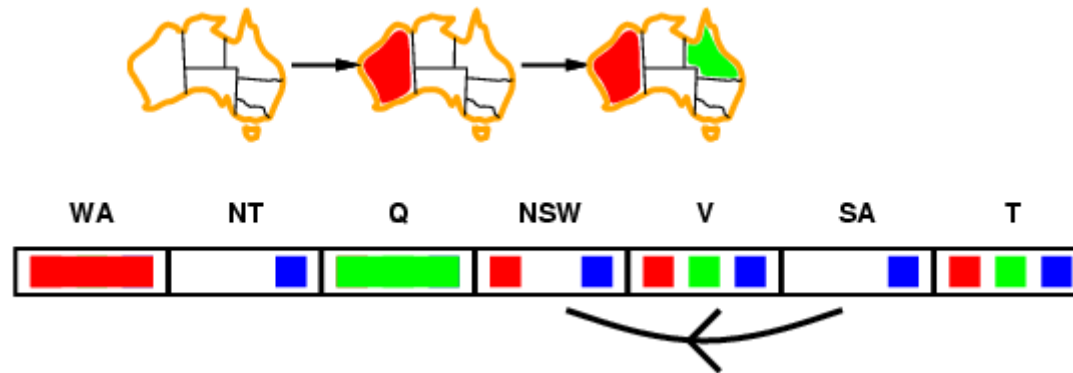
- 前方向検査は割当てられた変数から割当てられていない変数へ広がっていく。しかし、全ての失敗に対して早い段階での検出は行えない:



- NTとSA は両方ともblueではありえない
- 制約伝播は制約の局所化を繰り返し強める

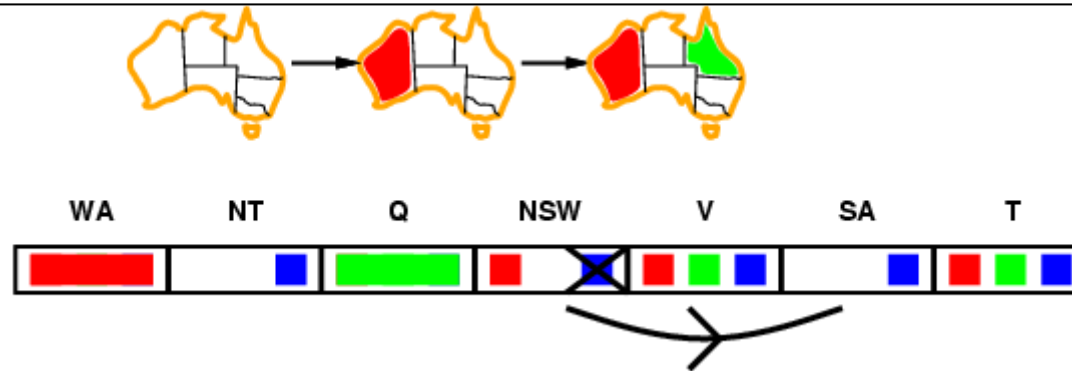
# アークの一貫性

- 伝播の簡単な形式はそれぞれのアークに一貫性を保たせる
- $X \rightarrow Y$ が一貫性であるときかつそのときに限り $X$ の全ての値 $x$ は**ある**許された $y$ を有する



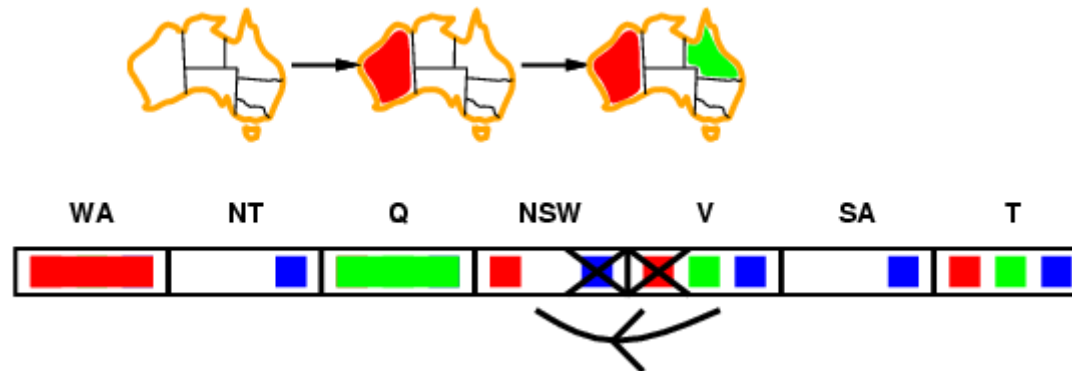
# アークの一貫性

- 伝播の簡単な形式はそれぞれのアークに一貫性を保たせる
- $X \rightarrow Y$ が一貫性であるときかつそのときに限り $X$ の全ての値 $x$ は**ある**許された $y$ を有する



# アークの一貫性

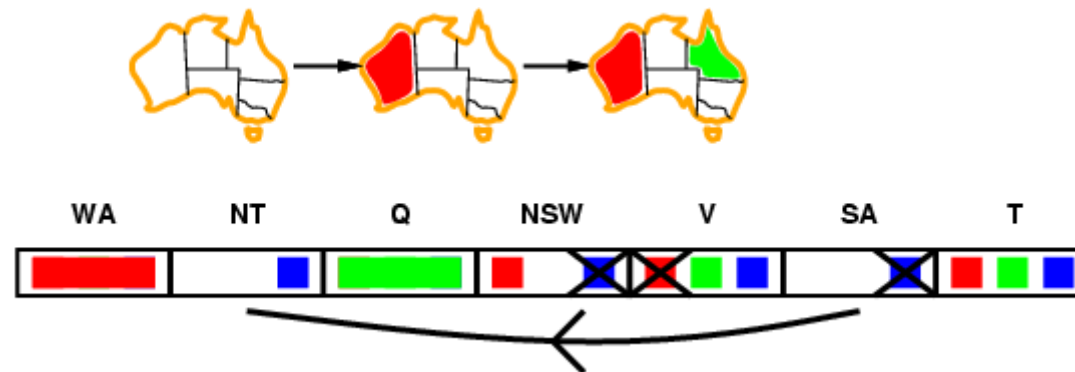
- 伝播の簡単な形式はそれぞれのアークに一貫性を保たせる
- $X \rightarrow Y$  が一貫性であるときかつそのときに限り  $X$  の全ての値  $x$  はある許された  $y$  を有する



- もし  $X$  が値を失ったなら  $X$  の隣は調べなおす

# アークの一貫性

- 伝播の簡単な形式はそれぞれのアークに一貫性を保たせる
- $X \rightarrow Y$  が一貫性であるときかつそのときに限り  $X$  の全ての値  $x$  はある許された  $y$  を有する



- もし  $X$  が値を失ったなら  $X$  の隣は調べなおす
- アークの一貫性は前向き検査よりも早いときに失敗を発見する
- 各割当のとき、前処理あるいは後処理として走らせることができる

# Arc consistency algorithm 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
  removed  $\leftarrow$  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[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

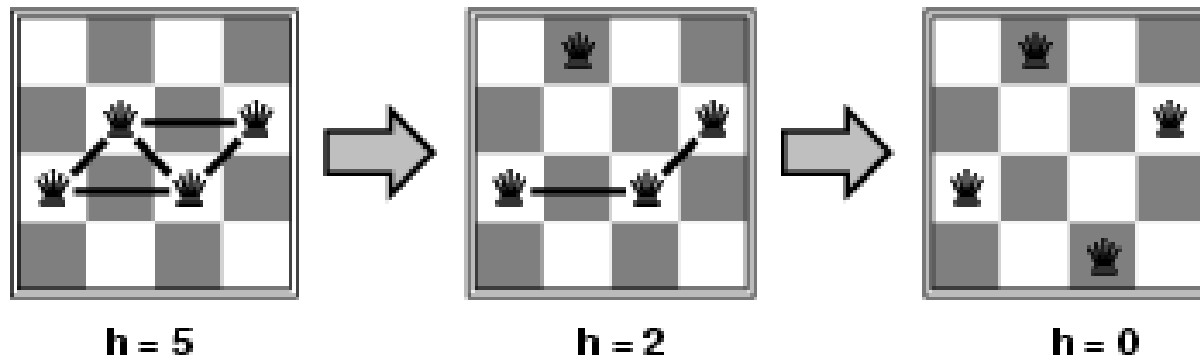
- Time complexity:  $O(n^2d^3)$

# CSPの局所探索

- 山登り法、焼きなまし法は“完全な”状態、すなわち割当てられた全ての変数、とともに機能する
- CSPに適応するために:
  - 満たされていない制約での状態を許す
  - オペレータは変数の値を再割当する
- 変数の選択: ランダムに衝突している変数の中から選ぶ
- 最小衝突の発見的方法による値の選択:
  - 最も少ない制約を壊す値を選択する
  - 即ち、 $h(n)$  = 破っている制約の総数 での山登り法

# Example: 4-Queens

- 状態: 4コラムでの4クイーン ( $4^4 = 256$  states)
- 動作: コラムでクイーンを動かす
- ゴールテスト: アタックがないようにする
- 評価:  $h(n) = \text{アタックの数}$



- ランダムな初期状態で、相当の確率で任意の  $n$  に対してほぼ制限時間の中で  $n$ -クイーンを解くことができる (例えば、 $n = 10,000,000$ )

# まとめ

- CSPは特別な問題である:
  - 状態は変数の集合への値により定義される
  - ゴールテストは変数の値での制約により定義される
- 後戻り法 = 各ノードに一変数を割当てて深さ優先探索
- 変数の順序付けと値の選択による発見的方法は非常に有効である
- 前向き検査は後で失敗が起こるものを妨げる
- 制約の伝播は値の制約と一貫性での矛盾の検出に更なる作業をする
- 繰り返しの最小衝突は実用上では効率的である