

CSCI 561 - Foundation for Artificial Intelligence

05 -- Constraint Satisfaction

06 -- Game Playing

wk3 -- Discussions

Professor Wei-Min Shen
University of Southern California

Constraint Satisfaction

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

Constraint satisfaction problems

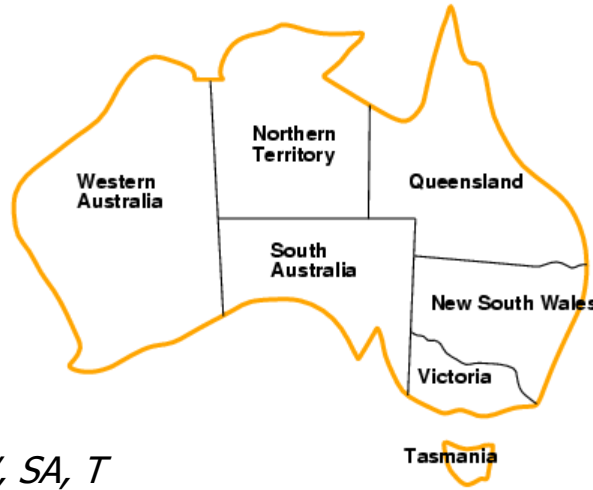
Standard search problem:

- state is a “black box” – any data structure that supports successor function (actions), heuristic evaluation function, and goal tests

CSP:

- State: is defined by variables X_i with values from domains D_i
- Goal Test: a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: map coloring problem



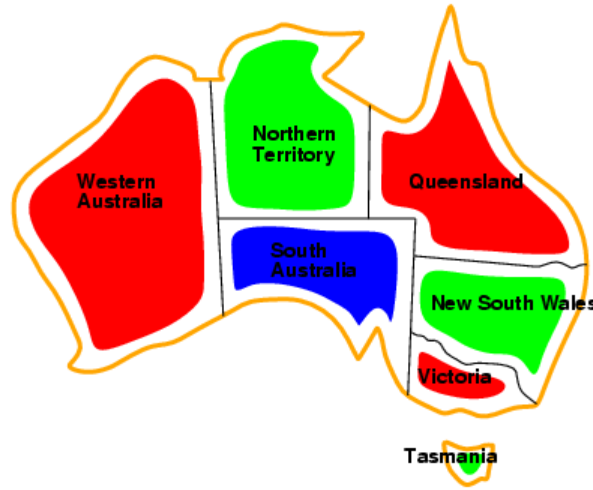
- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** $D_i = \{\text{red, green, blue}\}$ (one for each variable)
- **Constraints:** $C_i = \langle \text{scope}, \text{rel} \rangle$ where *scope* is a tuple of variables and *rel* is the relation over the values of these variables
 - E.g., here, adjacent regions must have different colors
e.g., $WA \neq NT$, or (WA, NT) in $\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$

Example: A Map Coloring Problem



- **Assignment:** values are given to some or all variables
- **Consistent (legal) assignment:** assigned values do not violate any constraint
- **Complete assignment:** every variable is assigned a value
- **Solution to a CSP:** a consistent and complete assignment

Example: A Map Coloring Problem



- **Solutions** are **complete** and **consistent** assignments,
- e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Demo

USA Mainland Coloring using CCM (Chemical Casting Model) -- ver 1.13, 1996-11-23 --

Stop

Restart

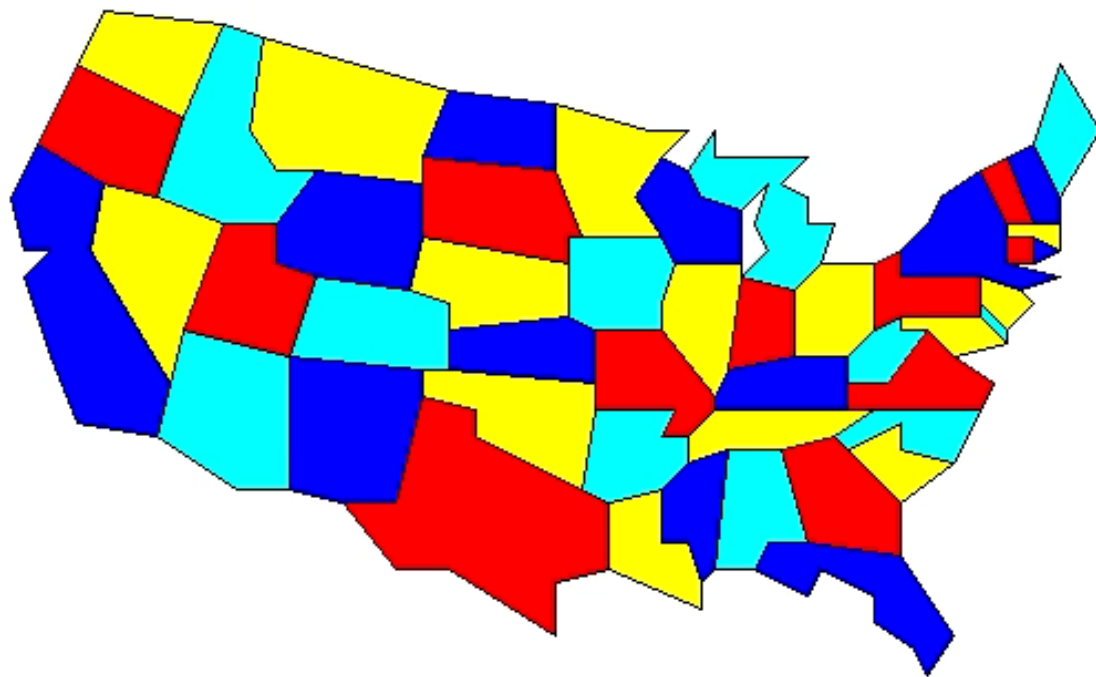
Frustration ON

or options:

Medium speed (20 reactions/sec)

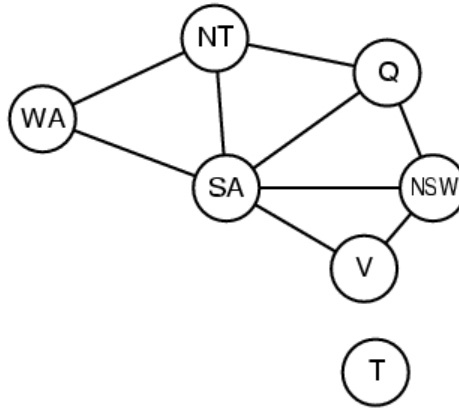
Variable catalyst rule

Stopped. #tests = 2002 #reactions = 100 MOD = 1.0 Time = 5.374



Constraint Graph

- **Binary CSP:** each constraint relates two variables
- **Constraint Graph:** nodes are variables, arcs are constraints



Varieties of CSPs

■ Discrete variables

- Finite domains:
 - n variables, domain size $d \Rightarrow O(d^n)$ complete assignments
 - e.g., Boolean CSPs, including Boolean Satisfiability (NP-complete)
- Infinite domains:
 - integers, strings, etc.
 - e.g., job scheduling, variables are start/end days for each job
 - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

■ Continuous variables

- e.g., start/end times for Hubble Space Telescope observations
- Linear constraints solvable in polynomial time by linear programming

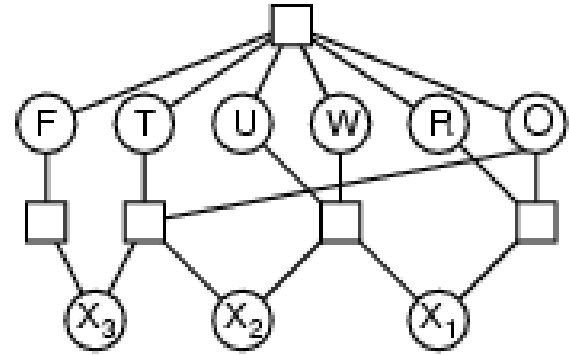
Constraint Types and Varieties

- **Unary** constraints involve a single variable,
 - e.g., $SA \neq \text{green}$
- **Binary** constraints involve pairs of variables,
 - e.g., $SA \neq WA$
- **Higher-order (sometimes called global)** constraints involve 3 or more variables,
 - 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:
 - $\text{Alldiff}(F, T, U, W, R, O)$
 - $O + O = R + 10 \cdot X_1$
 - $X_1 + W + W = U + 10 \cdot X_2$
 - $X_2 + T + T = O + 10 \cdot X_3$
 - $X_3 = F, T \neq 0, F \neq 0$



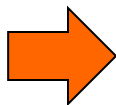
Constraint Hypergraph
 Circles: nodes for variable
 Squares: hypernodes for n-ary
 constraints

Real-world CSPs

- Assignment problems
 - e.g., who teaches what class
- Timetabling problems
 - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

Example: Sudoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 2 | 6 | | 7 | | 1 |
| 6 | 8 | | | 7 | | | 9 | |
| 1 | 9 | | | | 4 | 5 | | |
| 8 | 2 | | 1 | | | | 4 | |
| | | 4 | 6 | | 2 | 9 | | |
| | 5 | | | | 3 | | 2 | 8 |
| | | 9 | 3 | | | | 7 | 4 |
| | 4 | | | 5 | | | 3 | 6 |
| 7 | | 3 | | 1 | 8 | | | |



?

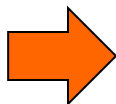
Variables: each square (81 variables)

Domains: [1 .. 9]

Constraints: each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9

Example: Sudoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 2 | 6 | | 7 | | 1 |
| 6 | 8 | | | 7 | | | 9 | |
| 1 | 9 | | | | 4 | 5 | | |
| 8 | 2 | | 1 | | | | 4 | |
| | | 4 | 6 | | 2 | 9 | | |
| | 5 | | | | 3 | | 2 | 8 |
| | | 9 | 3 | | | | 7 | 4 |
| | 4 | | | 5 | | | 3 | 6 |
| 7 | | 3 | | 1 | 8 | | | |



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 5 | 2 | 6 | 9 | 7 | 8 | 1 |
| 6 | 8 | 2 | 5 | 7 | 1 | 4 | 9 | 3 |
| 1 | 9 | 7 | 8 | 3 | 4 | 5 | 6 | 2 |
| 8 | 2 | 6 | 1 | 9 | 5 | 3 | 4 | 7 |
| 3 | 7 | 4 | 6 | 8 | 2 | 9 | 1 | 5 |
| 9 | 5 | 1 | 7 | 4 | 3 | 6 | 2 | 8 |
| 5 | 1 | 9 | 3 | 2 | 6 | 8 | 7 | 4 |
| 2 | 4 | 8 | 9 | 5 | 7 | 1 | 3 | 6 |
| 7 | 6 | 3 | 4 | 1 | 8 | 2 | 5 | 9 |

Variables: each square (81 variables)

Domains: [1 .. 9]

Constraints: each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9

Solving CSP Using the “Standard” Search Approach

Let's start with the straightforward approach, then improve it

States are defined by the values assigned so far

- **Initial state:** the empty assignment $\{ \}$
 - **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment
 - fail if no legal assignments
 - **Goal test:** the current assignment is complete
-
1. This is the same for all CSPs
 2. Every solution appears at depth n with n variables □ use depth-first search
 3. Path is irrelevant, so can be discarded
 4. $b = (n - \ell)d$ at depth ℓ , hence $n! \cdot d^n$ leaves

Backtracking Search

- Variable assignments are **commutative**, i.e.,
[WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assignments to a **single variable** at each node
 - $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called **backtracking** search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n -queens for $n \approx 25$

Backtracking search

(note: textbook has a slightly more complex version)

```
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
```

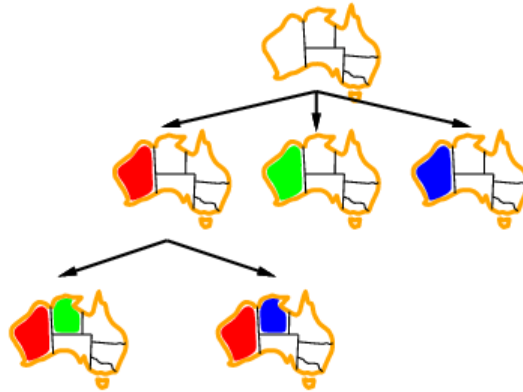
Backtracking example



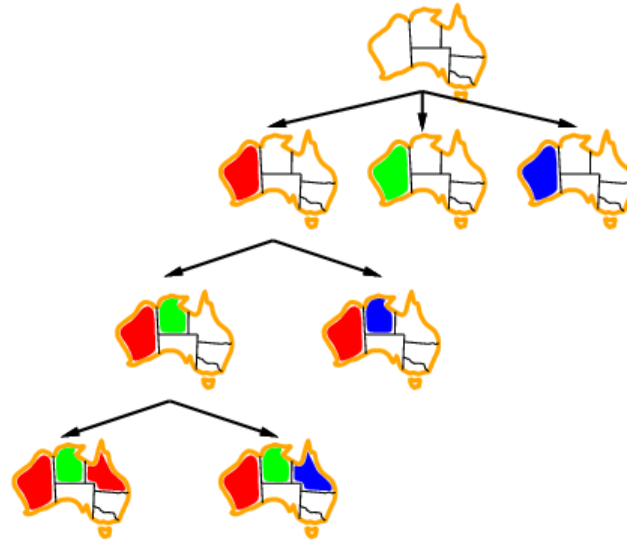
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

- **General-purpose** methods can give huge gains in speed (like using heuristics in informed search):
 - Which variable should be assigned next?
 - In what order should its values be tried?
 - Can we detect inevitable failure early?

Selecting the Most Constrained Variable

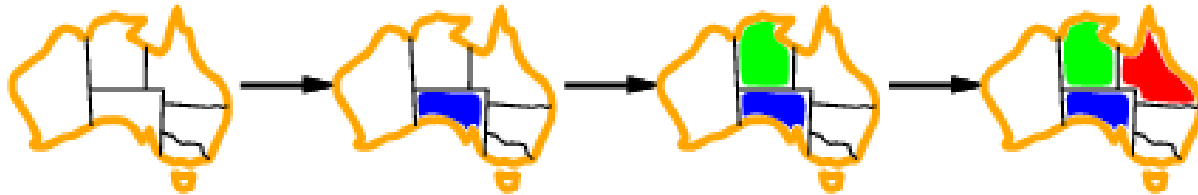
- Most constrained variable:
choose the variable with the fewest legal values



- a.k.a. **minimum remaining values (MRV)** heuristic

Most constraining variable

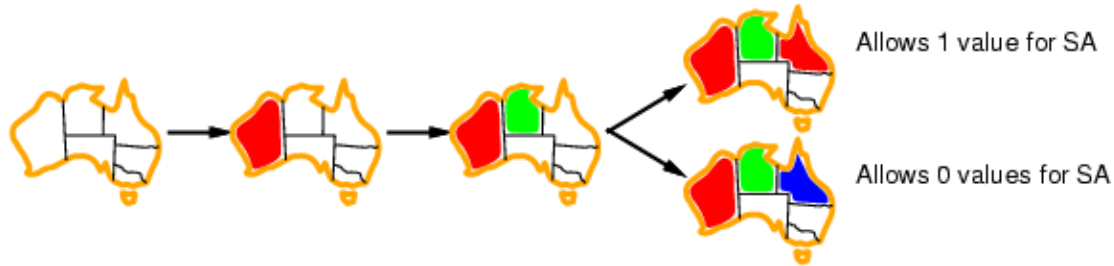
- Tie-breaker among most constrained variables
- Most constraining variable:
 - choose the variable with the most constraints on remaining variables



- also known as the **degree heuristic**

Least constraining value

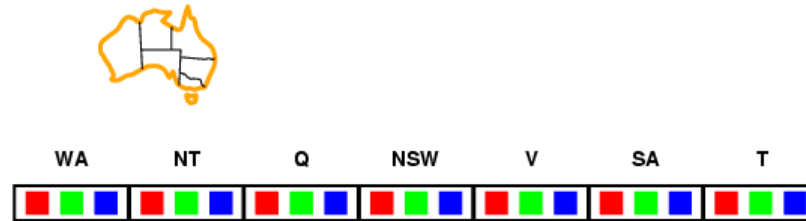
- Given a variable, choose the least constraining value:
 - the one that rules out the fewest values in the remaining variables



- Combining these heuristics makes 1000 queens feasible

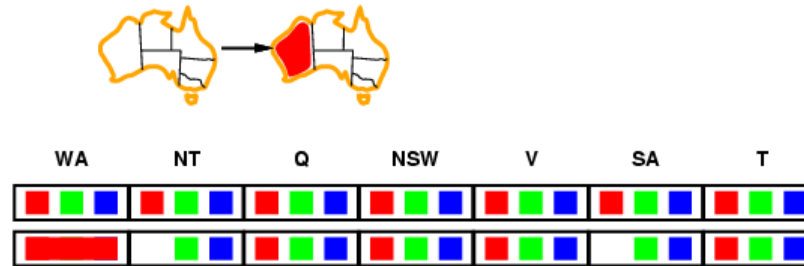
Forward Checking (to detect failures as early as possible)

- **Idea:**
 - Keep track of remaining legal values for unassigned variables (inference step)
 - Terminate search when any variable has no legal values



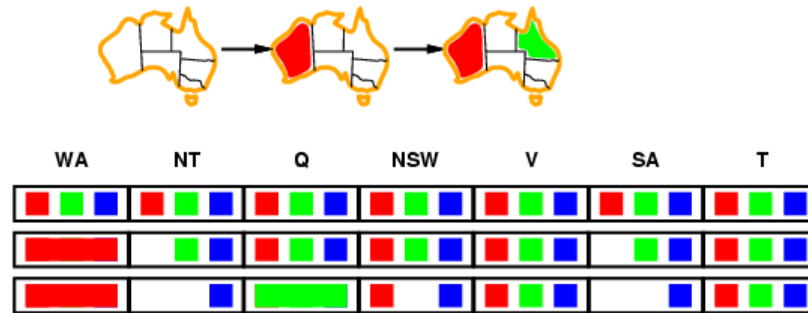
Forward checking

- **Idea:**
 - Keep track of remaining legal values for unassigned variables (inference step)
 - Terminate search when any variable has no legal values



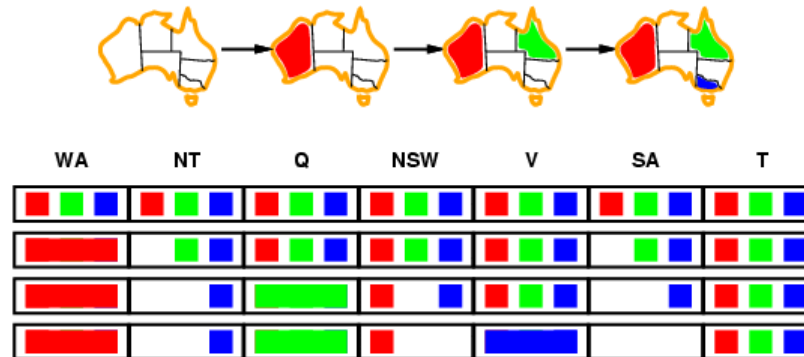
Forward checking

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



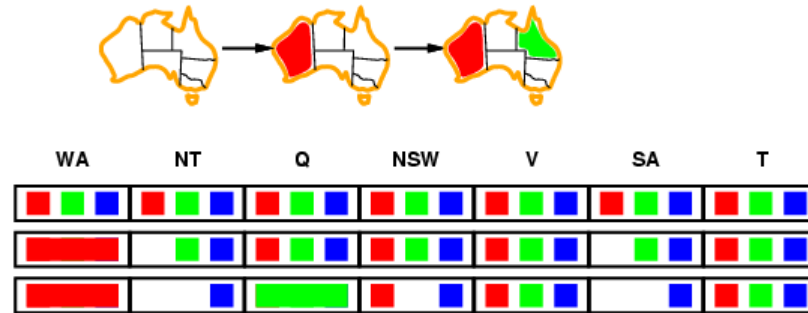
Forward checking

- **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

Node and Arc Consistency

A single variable is **node-consistent** if all the values in its domain satisfy the variable's unary constraints

A variable is **arc-consistent** if every value in its domain satisfies the binary constraints

- i.e., X_i arc-consistent with X_j if for every value in D_i there exists a value in D_j that satisfies the binary constraints on arc (X_i, X_j)

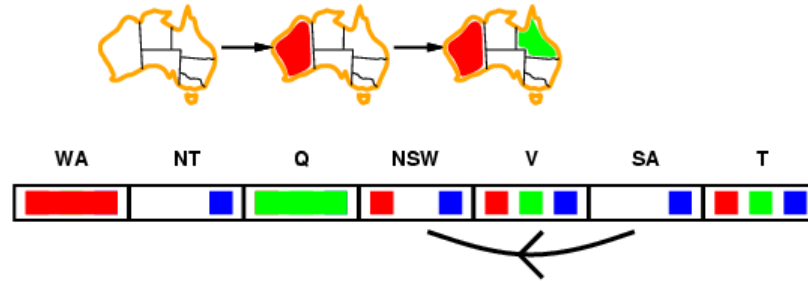
A **network is arc-consistent** if every variable is arc-consistent with every other variable.

Arc-consistency algorithms: reduce domains of some variables to achieve network arc-consistency.

Arc Consistency

- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$ is consistent iff

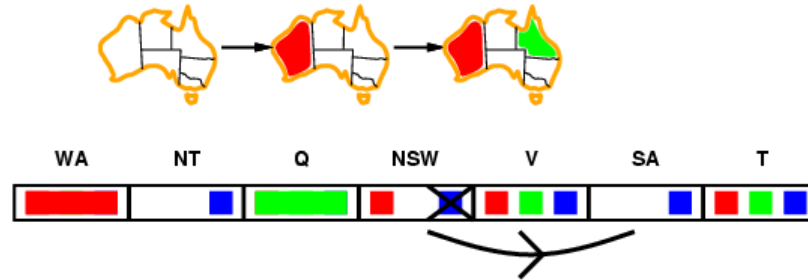
for **every** value x of X there is **some** allowed y



Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$ is consistent iff

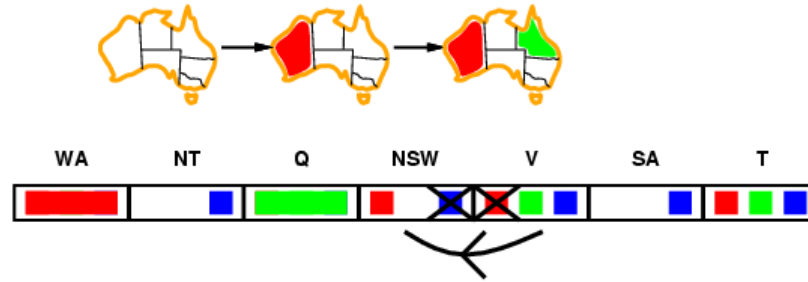
for **every** value x of X there is **some** allowed y



Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ is consistent iff

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

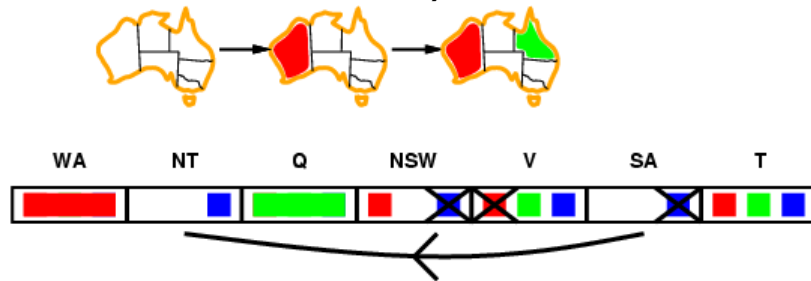


- If X loses a value, neighbors of X need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \square Y$ 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
- After running AC-3, either every arc is arc-consistent or some variable has empty domain, indicating the CSP cannot be solved.
- Can be run as a preprocessor or after each assignment

Arc Consistency Algorithm: AC-3

- Start with a queue that contains all arcs
- Pop one arc (X_i, X_j) and make X_i arc-consistent with respect to X_j
 - If D_i was not changed, continue to next arc,
 - Otherwise, D_i was **revised** (domain was reduced), so need to check all arcs connected to X_i again: add all connected arcs (X_k, X_i) to the queue. (this is because the reduction in D_i may yield further reductions in D_k)
 - If D_i is revised to empty, then the CSP problem has no solution.

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: ? (n variables, d values)

Arc Consistency algorithm AC-3

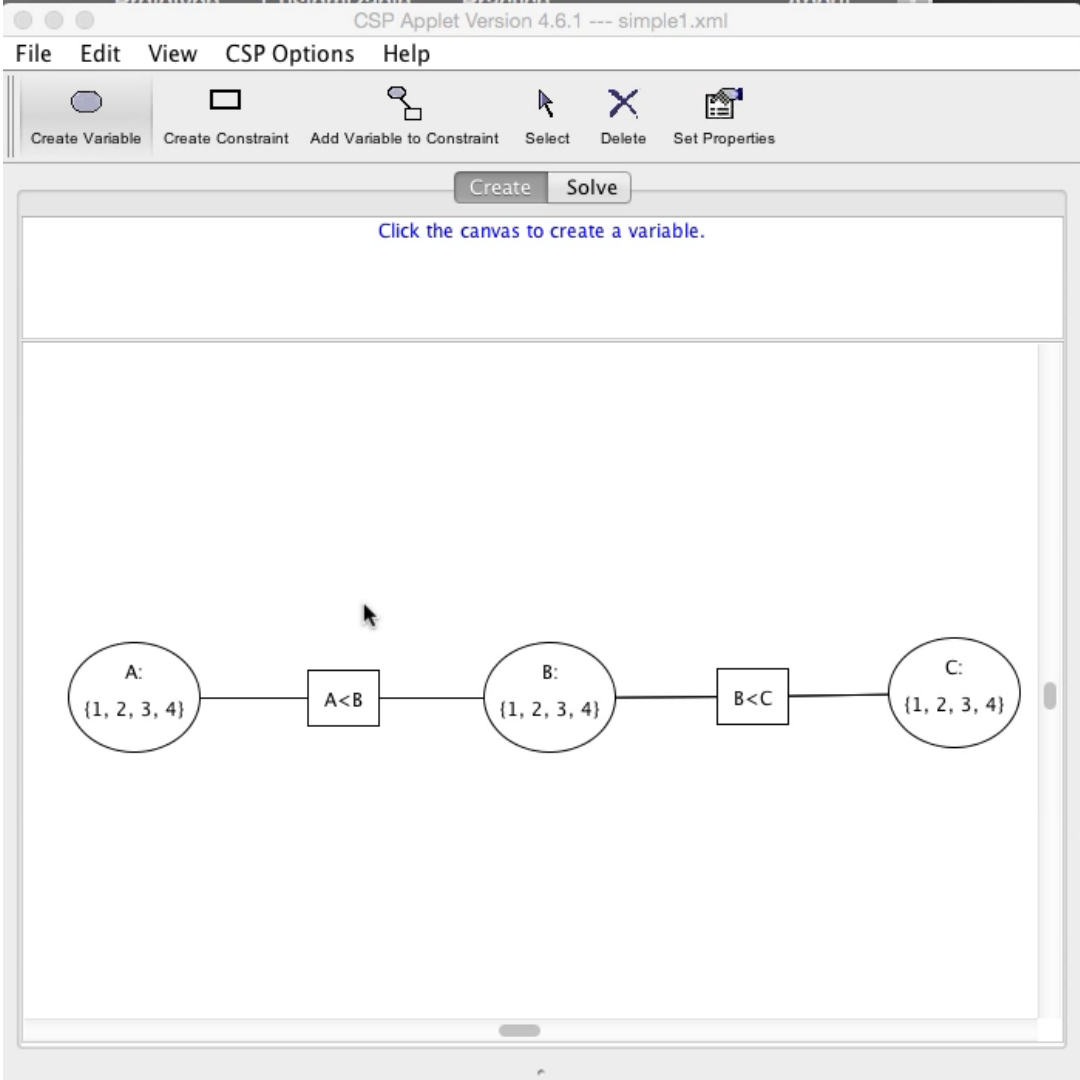
```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp

  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: $O(n^2d^3)$ (n variables, d values)
- (each arc can be queued only d times, n^2 arcs (at most), checking one arc is $O(d^2)$)

Demo

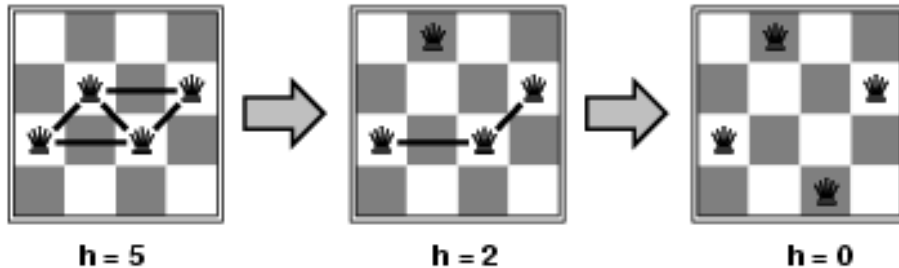


Another Approach: Local Search for CSPs

- 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., hill-climb with $h(n)$ = total number of violated constraints

Example: 4-Queens

- **States:** 4 queens in 4 columns ($4^4 = 256$ states)
- **Actions:** move queen in column
- **Goal test:** no attacks
- **Evaluation:** $h(n)$ = number of attacks



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

Min-Conflicts Algorithm

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

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Example: N-Queens

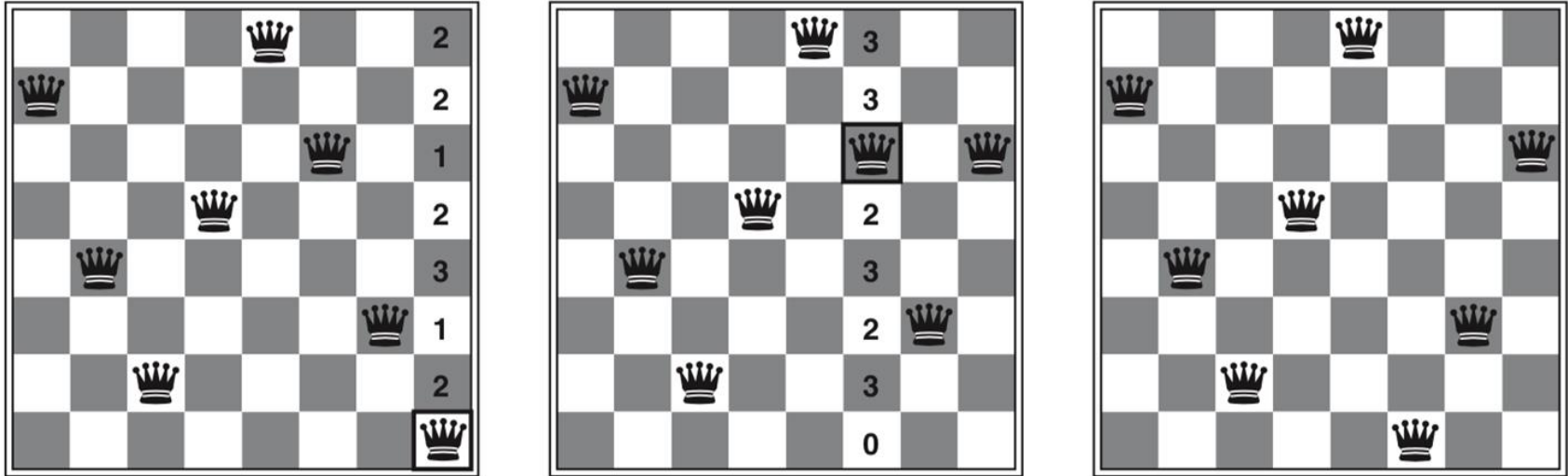


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Summary

- CSPs are a special kind of search 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
- Iterative min-conflicts is usually effective in practice