Constraint Satisfaction Problems

Chapter 6

Outline

Topics:

- CSP examples
- Backtracking search for CSPs
  - Improving backtracking efficiency
- Problem structure and problem decomposition
- Local search for CSPs

Constraint satisfaction problems (CSPs)

Standard search problem:

- A *state* is a "black box" – can be any data structure that supports goal test, eval, successor

Constraint satisfaction problems (CSPs)

Standard search problem:

- A *state* is a "black box" – can be any data structure that supports goal test, eval, successor

CSP:

- Each state has some structure, given by a set of *variables* and a set of *constraints*.
- The problem is solved when each variable has a value that satisfies the constraints.
- In a CSP, can use *general purpose* algorithms (as opposed to the *problem-specific* heuristics that we've seen in search).

Constraint satisfaction problems (CSPs)

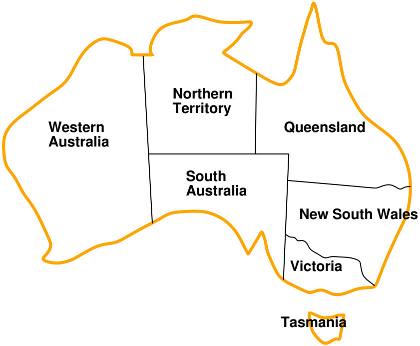
CSP:

- Defined by a set of *variables* $X_1, \ldots, X_n$, and a set of *constraints* $C_1, \ldots, C_m$.
- Each variable X_i has an associated *domain* D_i.
- Each constraint C_i involves some subset of the variables and specifies allowable combinations of values for that subset.
- A *state* is an assignment to some or all of the variable.
- A *solution* is a complete assignment that satisfies all constraints.
  (Sometimes: maximize an *objective function*.)

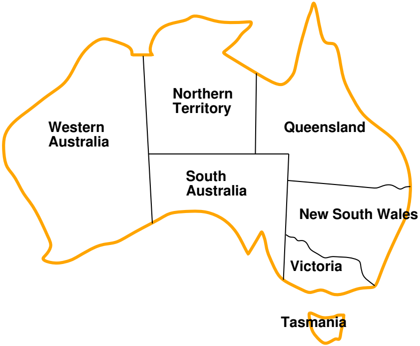
CSPs continued

- This is a simple example of a formal *representation language*
- Allows useful *general-purpose* algorithms with more power than standard search algorithms

Example: Map-Coloring



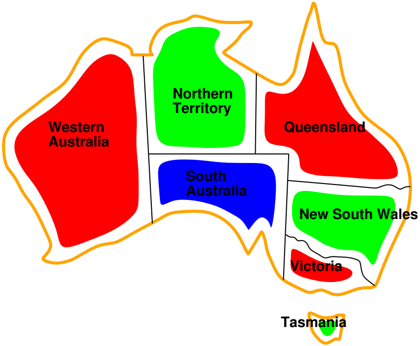
Example: Map-Coloring



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colours

- e.g., $WA \neq NT$ (if the language allows this), or
- $(WA, NT) \in \{(red, green), (red, blue), (green, red), \ldots\}$

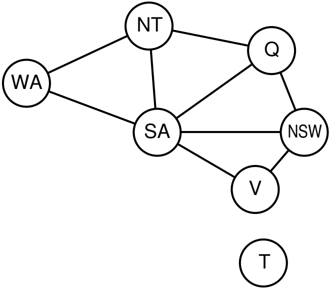
Example: Map-Coloring contd.



Solutions are assignments satisfying all constraints, e.g., $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

- *Binary CSP*: Each constraint relates at most two variables
- *Constraint graph*: Nodes are variables, arcs show constraints



- General-purpose CSP algorithms use the graph structure to speed up search.
  - E.g., Tasmania is an independent subproblem!

Varieties of CSPs

Discrete variables, finite domains:

- n vars, domain size $d \implies O(d^n)$ complete assignments
- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

Varieties of CSPs

Discrete variables, finite domains:

- n vars, domain size $d \implies O(d^n)$ complete assignments
- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

Discrete variables, infinite domains:

- integers, strings, etc.
- e.g., job scheduling, variables are start/end days for each job.
  $\Rightarrow$ need a *constraint language*
  e.g., $StartJob_1 + 5 \leq StartJob_3$
- *linear* constraints solvable; *nonlinear* undecidable.

Varieties of CSPs

Discrete variables, finite domains:

- n vars, domain size $d \implies O(d^n)$ complete assignments
- e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

Discrete variables, infinite domains:

- integers, strings, etc.
- e.g., job scheduling, variables are start/end days for each job.
  $\Rightarrow$ need a *constraint language*
  e.g., $StartJob_1 + 5 \leq StartJob_3$
- *linear* constraints solvable; *nonlinear* undecidable.

Continuous variables:

- e.g., start/end times for Hubble Telescope observations.
- linear constraints solvable in poly time by LP methods.

Varieties of Constraints

Unary constraints: Involve a single variable.

- e.g., $SA \neq green$

Varieties of Constraints

Unary constraints: Involve a single variable.

- e.g., $SA \neq green$

Binary constraints: Involve pairs of variables.

- e.g., $SA \neq WA$

Varieties of Constraints

Unary constraints: Involve a single variable.

- e.g., $SA \neq green$

Binary constraints: Involve pairs of variables.

- e.g., $SA \neq WA$

Higher-order constraints: Involve 3 or more variables.

- e.g., sudoku, cryptarithmetic column constraints

Varieties of Constraints

Unary constraints: Involve a single variable.

- e.g., $SA \neq green$

Binary constraints: Involve pairs of variables.

- e.g., $SA \neq WA$

Higher-order constraints: Involve 3 or more variables.

- e.g., sudoku, cryptarithmetic column constraints

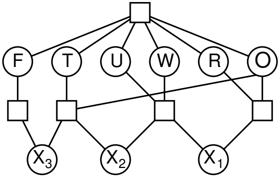
Preferences (soft constraints):

- e.g., *red* is better than *green*
- Often representable by a cost for each variable assignment.

  $\rightarrow$ constrained optimization problems

Higher-Order Example: Cryptarithmetic

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



- *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* (represented by square boxes):
  - $alldiff(F, T, U, W, R, O)$
  - $O + O = R + 10 \cdot X_1$, etc.

Higher-order Constraints

Higher-order constraints can be reduced to binary constraints by introducing new auxiliary variables.

- We're not going to cover this.
  - See Exercise 6.6, 3^{rd} ed. or Exercise 5.11, 2^{nd} ed. for a hint as to how this can be done.
- But as a result of this, we'll just deal with binary constraints.

Real-world CSPs

- Assignment problems
  e.g., who teaches what class
- Timetabling problems
  e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Floorplanning

☞ Notice that many real-world problems involve real-valued variables.

Naive Search Formulation (Incremental)

- We start with the straightforward, dumb approach, then fix it
- Define the state-space:

  States are defined by the values assigned so far.

  Initial state: The empty assignment, $\emptyset$
  Successor function: Assign a value to an unassigned variable that does not conflict with current assignment.
  - Fail if no legal assignments (not fixable!)

  Goal test: The current assignment is complete

Naive Search Formulation (Incremental)

Notes:

1. This can be used for all CSPs!
2. Every solution appears at depth n with n variables
   - use depth-first search
3. Path is irrelevant
4. $b = (n - \ell)d$ at depth ℓ where domain size for all variables is d.
   - there are $n!d^n$ leaves, even though there are only d^n complete assignments!

Backtracking Search

- Problem with the naive formulation:
  - It ignores that variable assignments are *commutative*
  - i.e. [$WA = red$ then $NT = green$]
    same as
    [$NT = green$ then $WA = red$]
- So just consider assignments to a single variable at each node
  - Obtain d^n leaves
- Depth-first search for CSPs with single-variable assignments is called *backtracking* search
  - I.e. try assigning values of successive variables, and backtrack when a variable has no legal values to assign.
  - ☞ Backtracking search is the basic uninformed algorithm for CSPs
  - Can solve n-queens for $n \approx 25$

Backtracking search

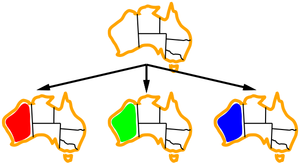
Function Backtracking-Search(csp) returns solution/failure
  return Recursive-Backtracking($\{\}$, csp)

Backtracking search

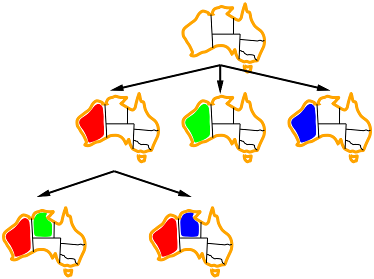
```
Function Backtracking-Search(csp) returns solution/failure
   return Recursive-Backtracking({ }, csp)

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 ≠ failure then
            return result
         remove {var = value} from assignment
   return failure
```

Backtracking example

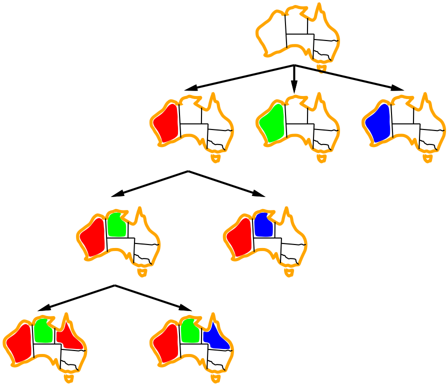
Backtracking example



Backtracking example



Backtracking example

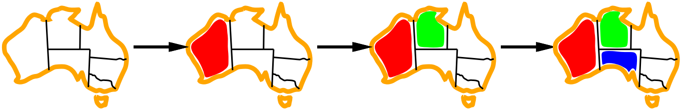


Improving backtracking efficiency

- In Chapter 3 we looked at improving performance of uninformed searches by considering domain-specific information.
- For CSPs, *general-purpose (uninformed)* heuristics can give huge gains in speed.
- Consider the following questions:
  1. Which variable should be assigned next?
  2. In what order should its values be tried?
  3. Can we detect inevitable failure early?
  4. Can we take advantage of problem structure?

Minimum remaining values

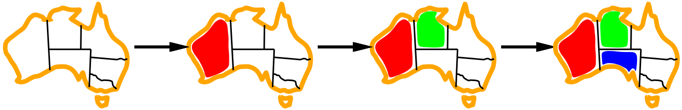
- *Minimum remaining values (MRV)*: Choose the variable with the fewest legal values



- Thus we choose the variable that seems most likely to fail.

Minimum remaining values

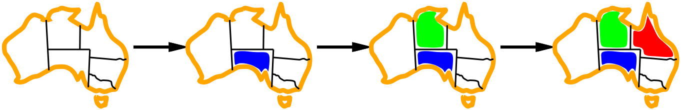
- *Minimum remaining values (MRV)*: Choose the variable with the fewest legal values



- Thus we choose the variable that seems most likely to fail.
- Can save an exponential amount of time. (why?)

Degree heuristic

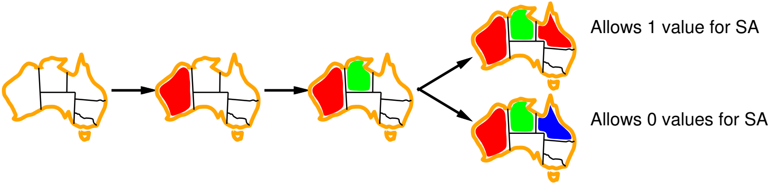
- Tie-breaker among MRV variables
- *Degree heuristic*: Choose the variable with the most constraints on other unassigned variables



- In this case, begin with SA, since it is involved with the greatest number of constraints with unassigned variables.
  - I.e. Deg(SA) $= 5$; all others have degree ≤ 3.

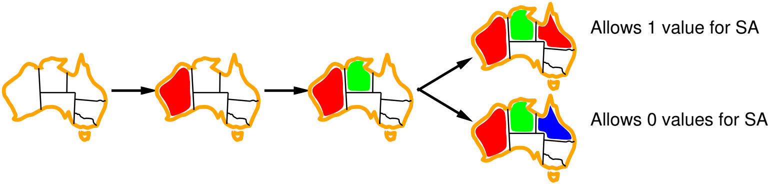
Least constraining value

- Given a variable, have to decide which value to assign.
- Here: Choose the *least constraining value*:
  - i.e. the one that rules out the fewest values in the remaining variables



Least constraining value

- Given a variable, have to decide which value to assign.
- Here: Choose the *least constraining value*:
  - i.e. the one that rules out the fewest values in the remaining variables



- Combining these heuristics makes 1000 queens feasible

Beyond Simple Search

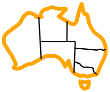
- So far, we have looked at backtracking search, and ways to speed it up.
- It turns out, additional efficiency can be gained by carrying out further processing at a state.
- We'll look at:
    - Forward checking
    - Constraint propagation: Arc consistency

Forward Checking

- *Idea*:

  Keep track of remaining legal values for unassigned variables

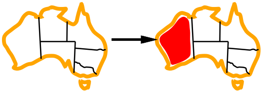
  - Terminate search when any variable has no legal values

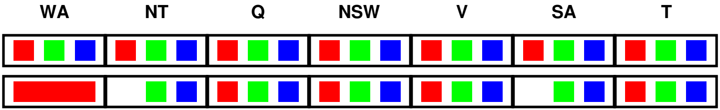


- *Idea*:

  Keep track of remaining legal values for unassigned variables

  - Terminate search when any variable has no legal values



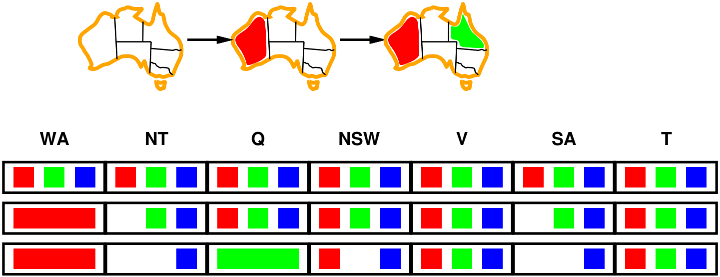


Forward checking

- *Idea*:

  Keep track of remaining legal values for unassigned variables

  - Terminate search when any variable has no legal values

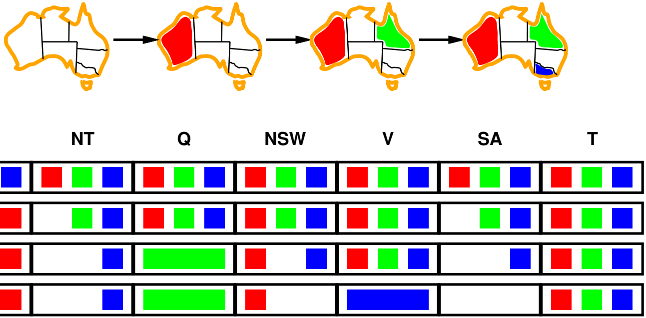


Forward checking

- *Idea*:

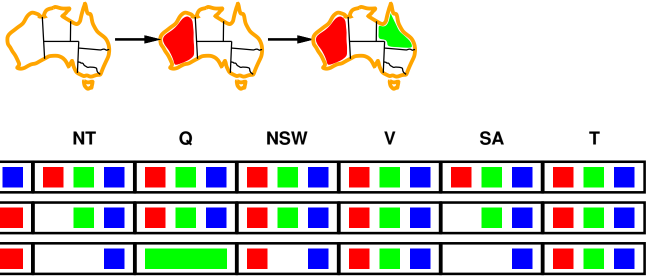
  Keep track of remaining legal values for unassigned variables

  - Terminate search when any variable has no legal values



Constraint propagation

- Forward checking propagates information from assigned to unassigned variables.
    - Doesn't provide early detection for all failures.
- E.g., second step in the previous example:



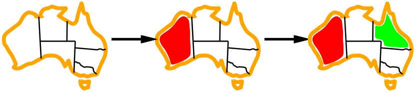
- *NT* and *SA* cannot both be blue!
    - *Constraint propagation* repeatedly enforces constraints locally

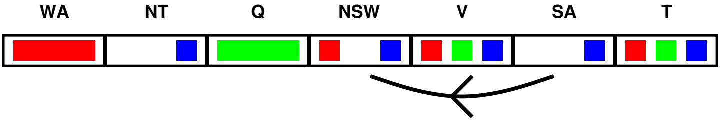
Constraint Propagation (cont'd)

- Constraint propagation involves propagating the implications of a constraint on one variable onto other variables.
  - Must be *fast*
  - I.e. it's no good reducing the amount of search if we spend a whole lot of time propagating constraints.

Arc Consistency

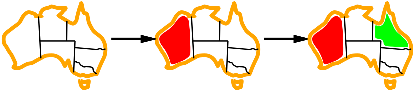
- Simplest form of propagation, makes each arc *consistent*
- $X \rightarrow Y$ is consistent iff
  for *every* value x of X there is *some* allowed y of Y.

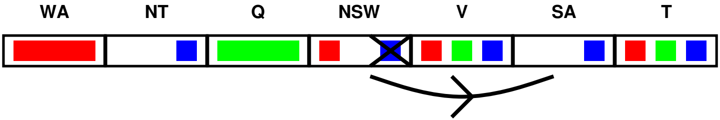




Arc Consistency

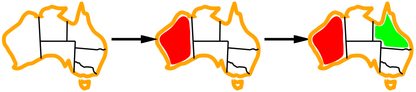
- Simplest form of propagation, makes each arc *consistent*.
- $X \rightarrow Y$ is consistent iff
  for *every* value x of X there is *some* allowed y.

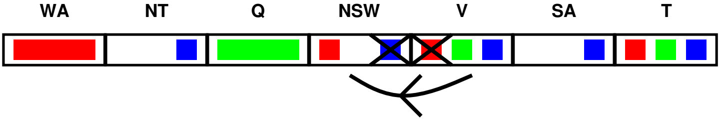




Arc Consistency

- 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.



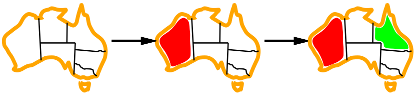


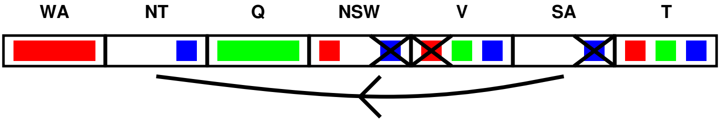
- If X loses a value, neighbors of X need to be rechecked.

Arc Consistency

- Simplest form of propagation, makes each arc *consistent*.
- $X \to Y$ 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 removed?
  removed? $\leftarrow$false
  for each x in Domain$[X_i]$ do
    if no $y \in$ Domain$[X_j]$ allows (x,y) to satisfy the X_i, X_j constraint
      then delete x from Domain$[X_i]$; removed? $\leftarrow$true
  return removed?

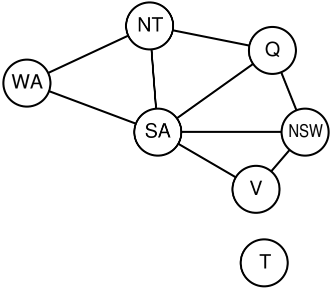
Arc Consistency Algorithm

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 removed?
  removed? $\leftarrow$false
  for each x in Domain$[X_i]$ do
    if no $y \in$ Domain$[X_j]$ allows (x,y) to satisfy the X_i, X_j constraint
      then delete x from Domain$[X_i]$; removed? $\leftarrow$true
  return removed?

☞ $O(n^2 d^3)$, can reduce 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 $(n/c) \times d^c$, *linear* in n

Problem Structure contd.

- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $(n/c) \times d^c$, *linear* in n
- E.g., $n = 80$, $d = 2$, $c = 20$, and 10 million nodes/sec

Problem Structure contd.

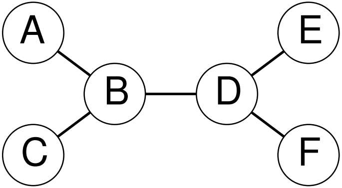
- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $(n/c) \times d^c$, *linear* in n
- E.g., $n = 80$, $d = 2$, $c = 20$, and 10 million nodes/sec
    - $2^{80} = 4$ billion years
    - $4 \cdot 2^{20} = 0.4$ seconds

Problem Structure contd.

- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $(n/c) \times d^c$, *linear* in n
- E.g., $n = 80$, $d = 2$, $c = 20$, and 10 million nodes/sec
  - $2^{80} = 4$ billion years
  - $4 \cdot 2^{20} = 0.4$ seconds

☞ So a good heuristic is to assign values to variables so as to break a problem into independent subproblems.

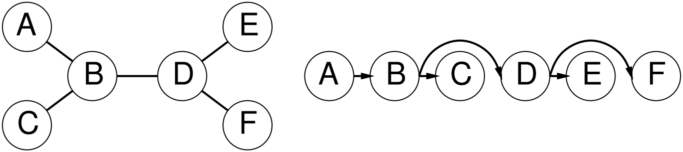
Tree-structured CSPs



- *Theorem*: If the constraint graph is a tree, the CSP can be solved in $O(n\,d^2)$ time
- Compare to general CSPs: 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



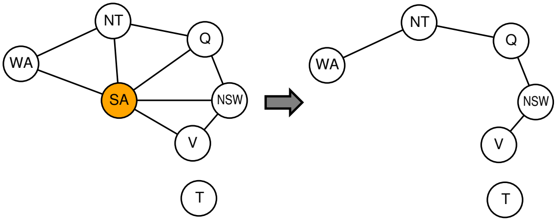
2. For j from n down to 2, apply

   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: Cutset Conditioning

- *Conditioning*: Instantiate a variable, prune its neighbors' domains



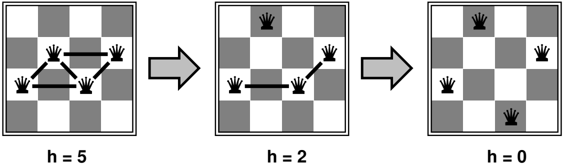
- *Cycle cutset*: Instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size $c \implies$ runtime $O(d^c \cdot (n-c)d^2)$
  ☞ Very fast for small c

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

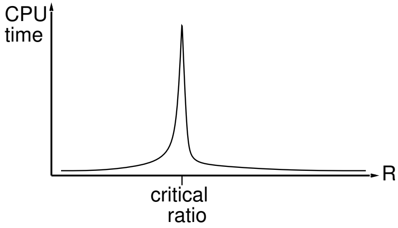
States: 4 queens in 4 columns ($4^4 = 256$ states)
Operators: move queen in column
Goal test: no attacks
Evaluation: $h(n)$ = number of attacks



Performance of Min-Conflicts

- 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

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



- Good example: Propositional satisfiability

Summary

- CSPs are a special kind of problem:
  - States are defined by values of a fixed set of variables.
  - Goal test defined by *constraints* on variable values.
- Backtracking = depth-1^{st} search with one variable assigned per node.
- Var. ordering and value selection heuristics help a great deal.
- 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.