Backtracking

Constraint Satisfaction Problem

Constraint Satisfaction Problem

- Sometimes we can finish the constraint propagation process and still have variables with multiple possible values.
- In that case we have to search for a **solution**.

- For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n. But notice that the branching factor at the top level would be nd because any of d values can be assigned to any of n variables.

- At the next level, the branching factor is (n — 1)d, and so on for n levels.

- So the tree has $n!.d^n$ leaves, even though there are only d^n possible complete assignments!

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

$\Rightarrow \quad b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments
is called backtracking search

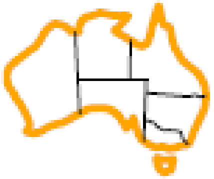
Backtracking search is the basic uninformed algorithm for CSPs

Backtracking search

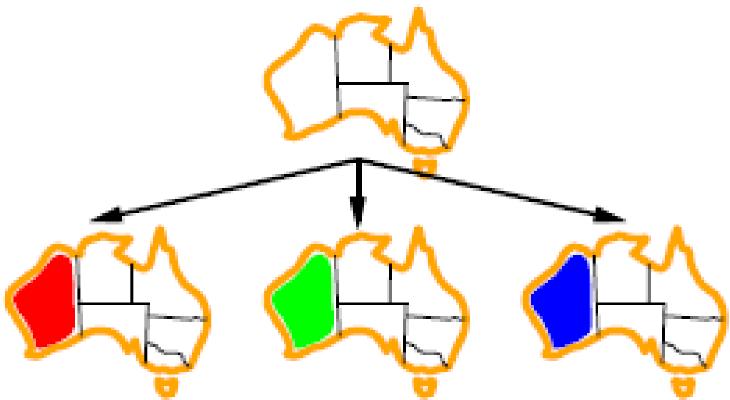
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

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
```

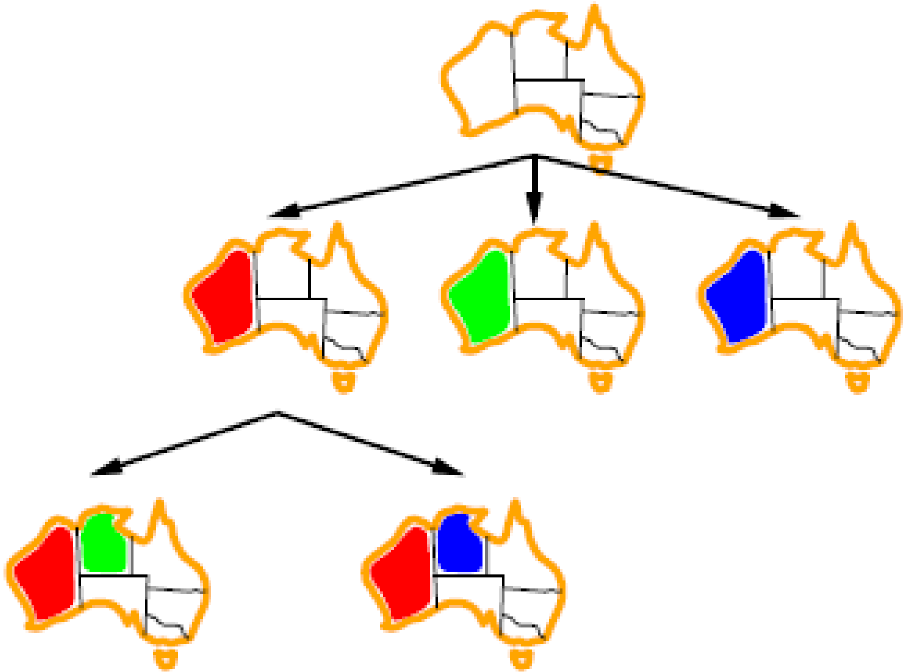
Backtracking example



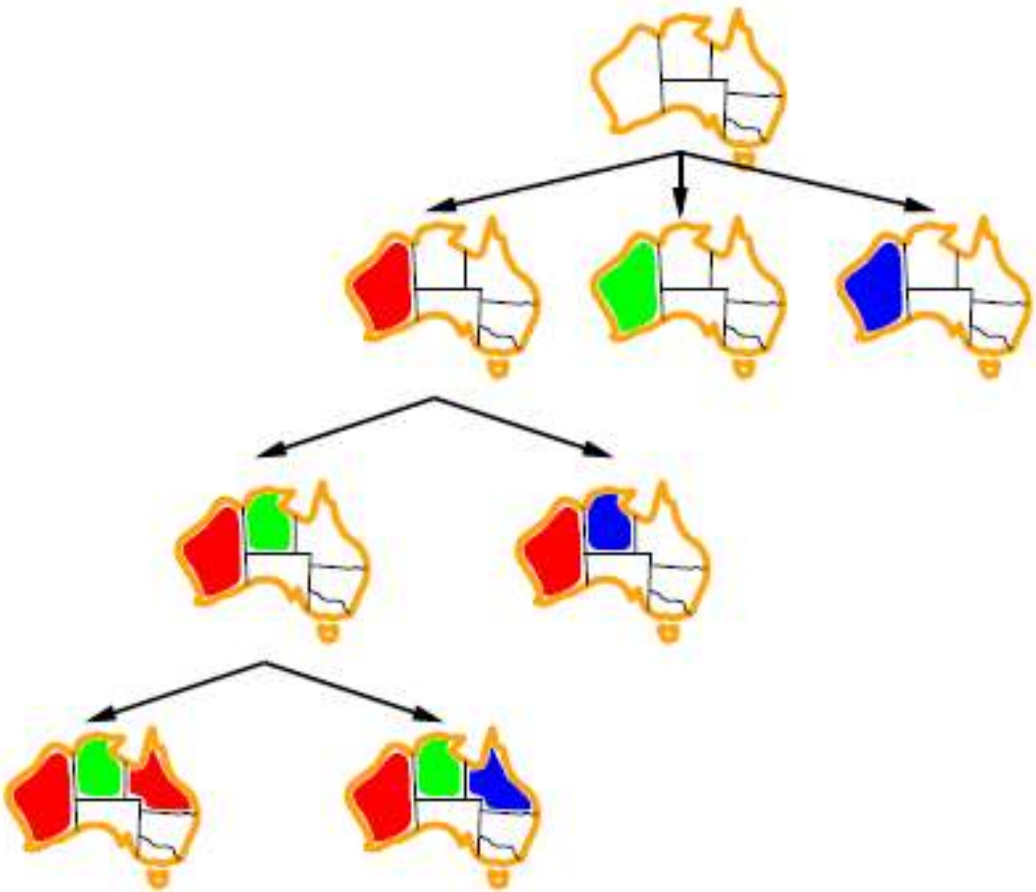
Backtracking example



Backtracking example



Backtracking example



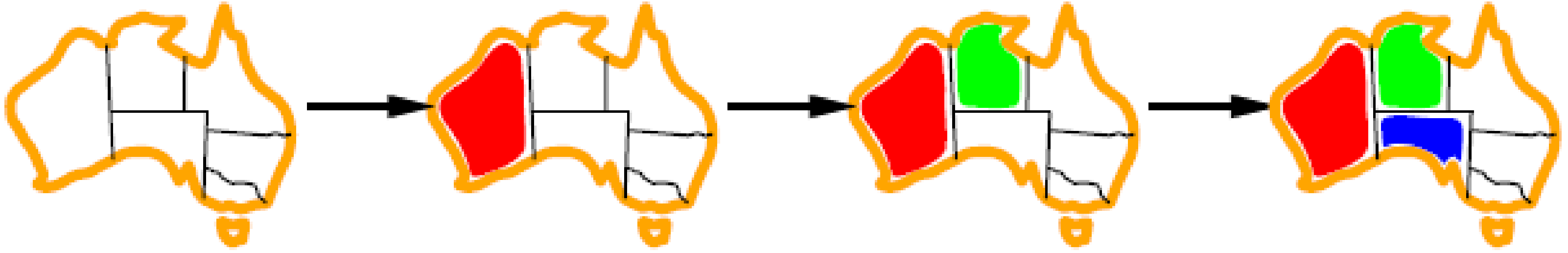
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

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

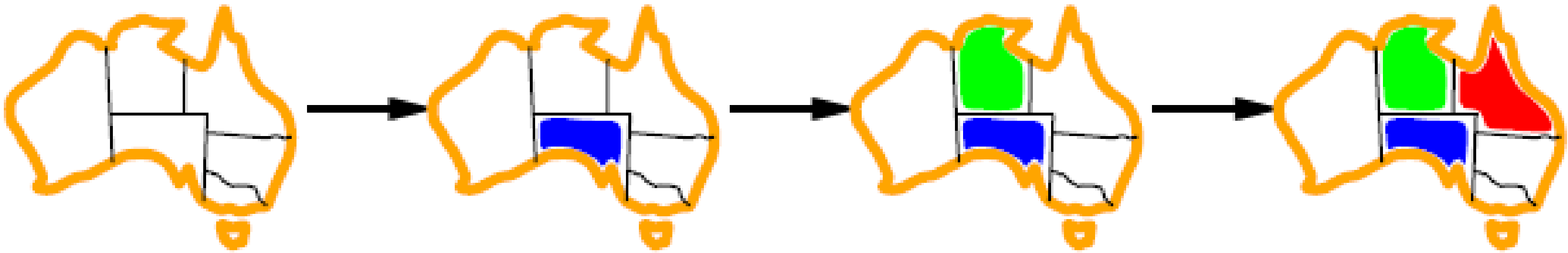
Minimum remaining values (MRV):
choose the variable with the fewest legal values



Degree heuristic

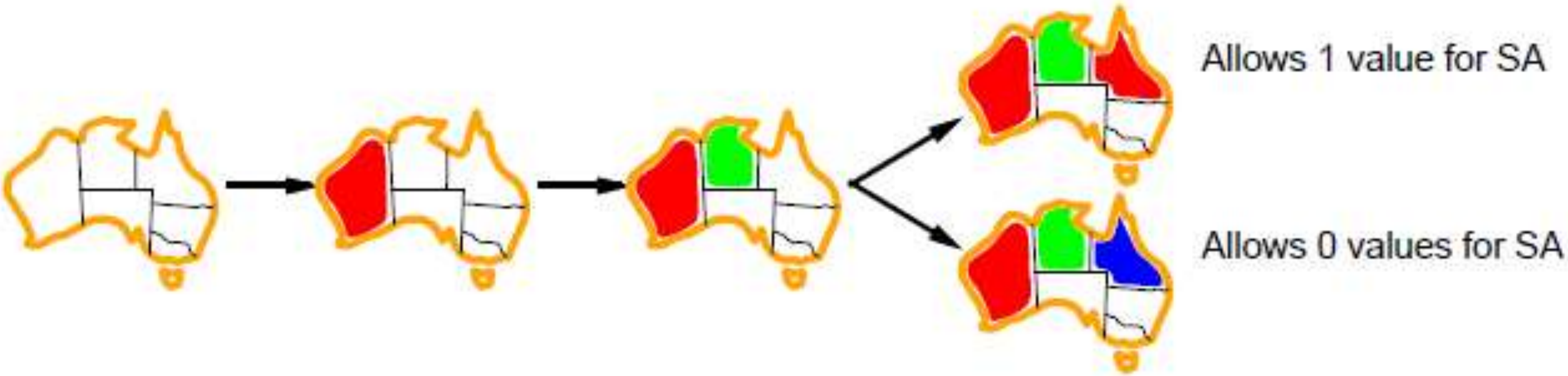
Tie-breaker among MRV variables

Degree heuristic:
    choose the variable with the most constraints on remaining variables



Least constraining value

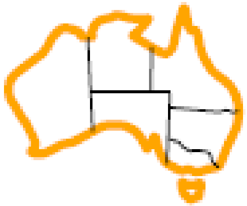
Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables

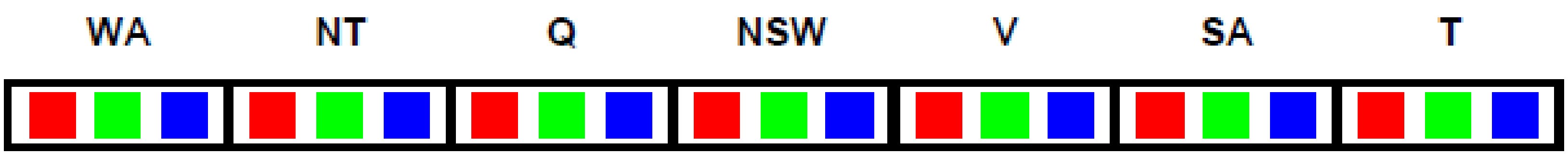


Combining these heuristics makes 1000 queens feasible

Forward checking

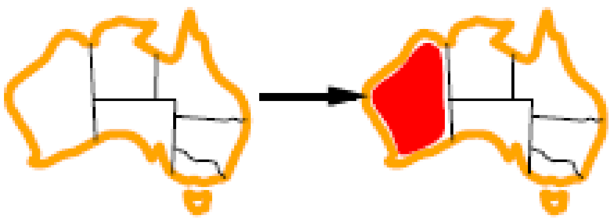
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

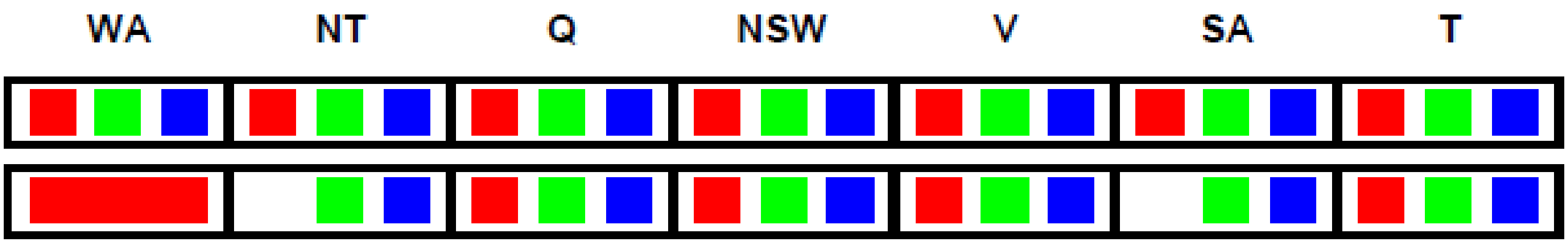




Forward checking

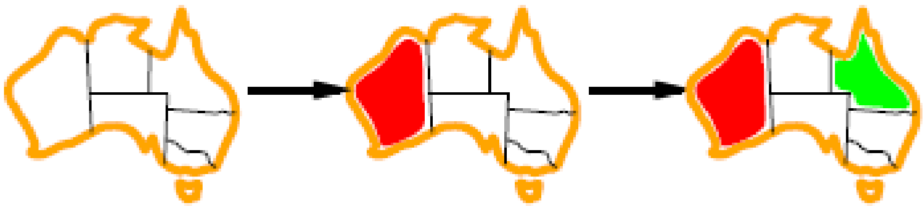
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

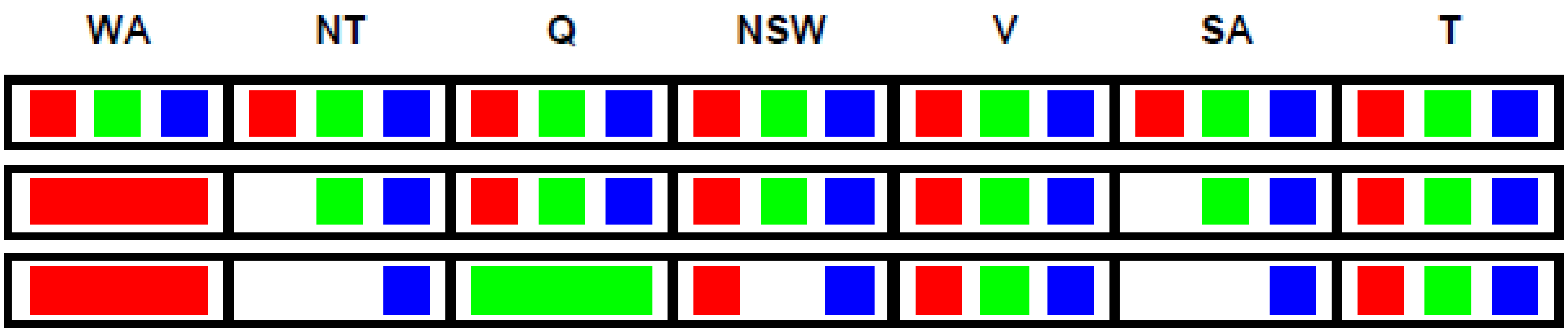




Forward checking

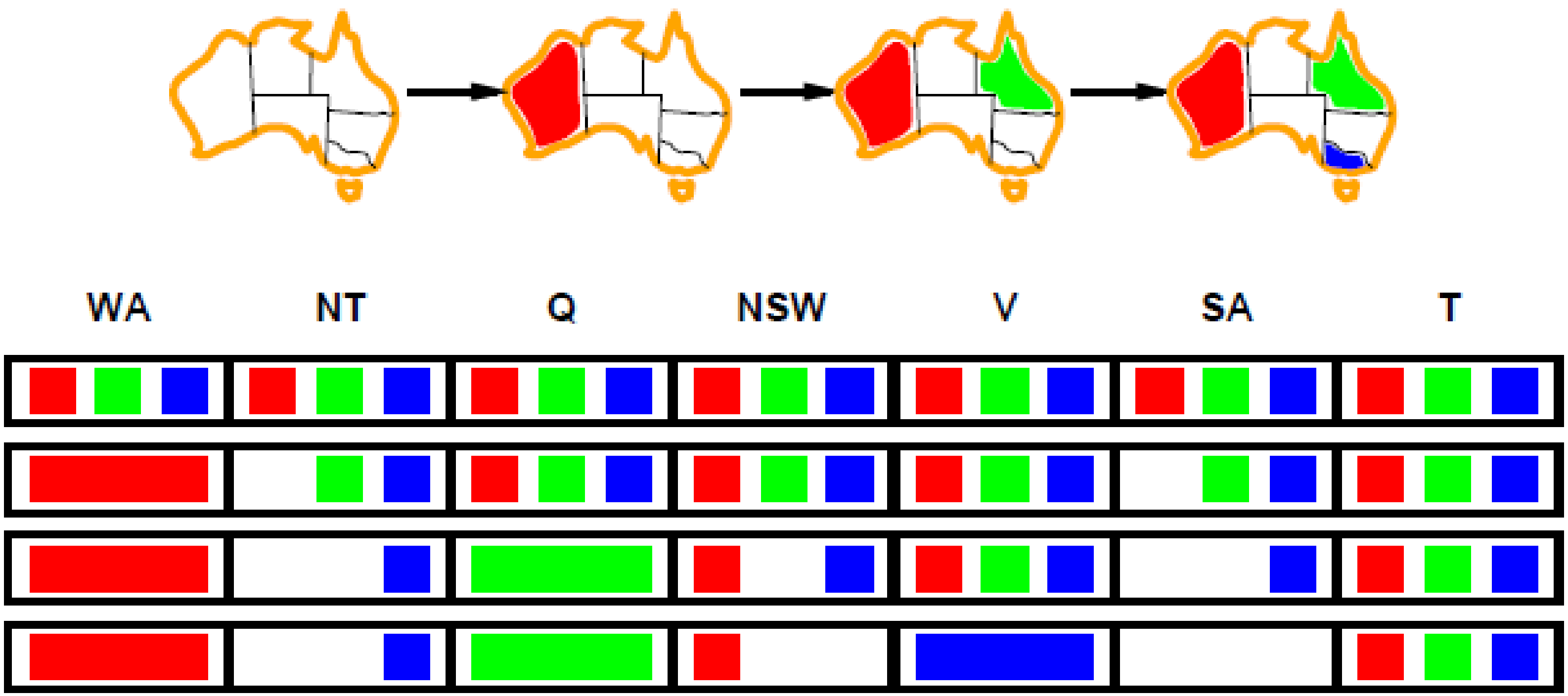
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





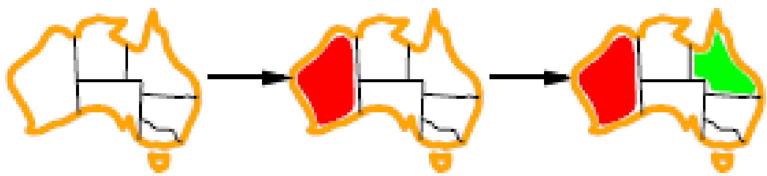
Forward checking

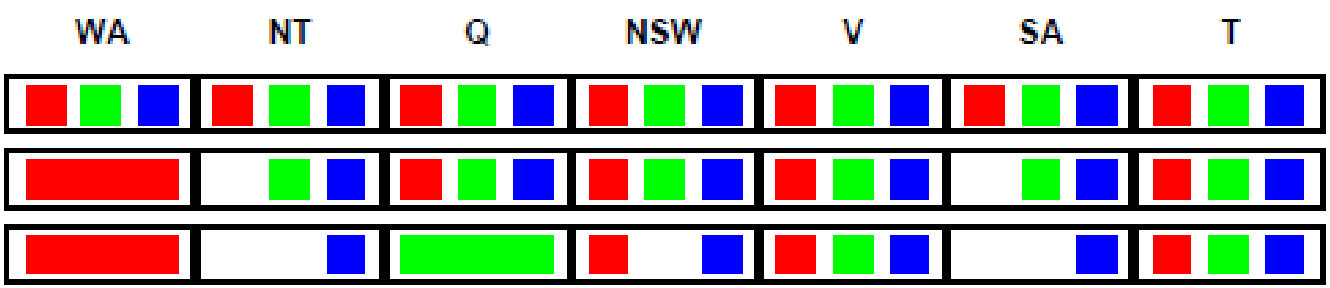
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Constraint propagation

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





NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc-consistentency

- A variable in a CSP is arc-consistent (edge-consistent) if every value in its domain satisfies the variable's binary constraints.

- Xi is arc-consistent with respect to another variable Xj; if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj).

- A graph is arc-consistent if every variable is arcconsistent with every other variable.

Example

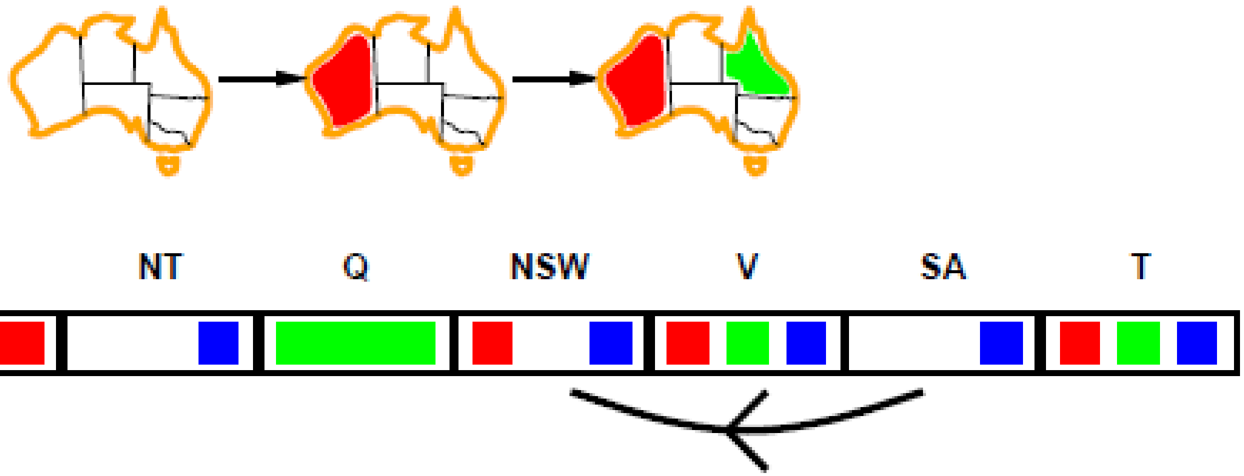
- Consider the constraint $Y = X^2$
  - where the domain of both X and Y is the set of decimal digits.
- We can write this constraint explicitly as
  $((X,Y),\{(0,0),(1,1),(2,4),(3,9)\})$
- To make X arc-consistent with respect to Y,
  - we reduce X's domain to {0, 1,2,3}.
- If we also make Y arc-consistent with respect to X,
  - then Y's domain becomes {0, 1,4,9}, and the whole CSP is arc-consistent.

Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

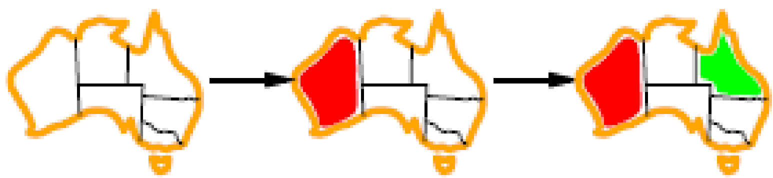
for every value x of X there is some allowed y

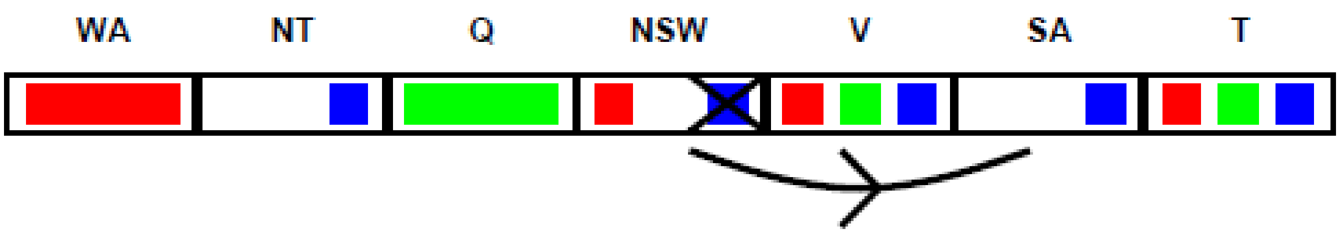


Arc consistency

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



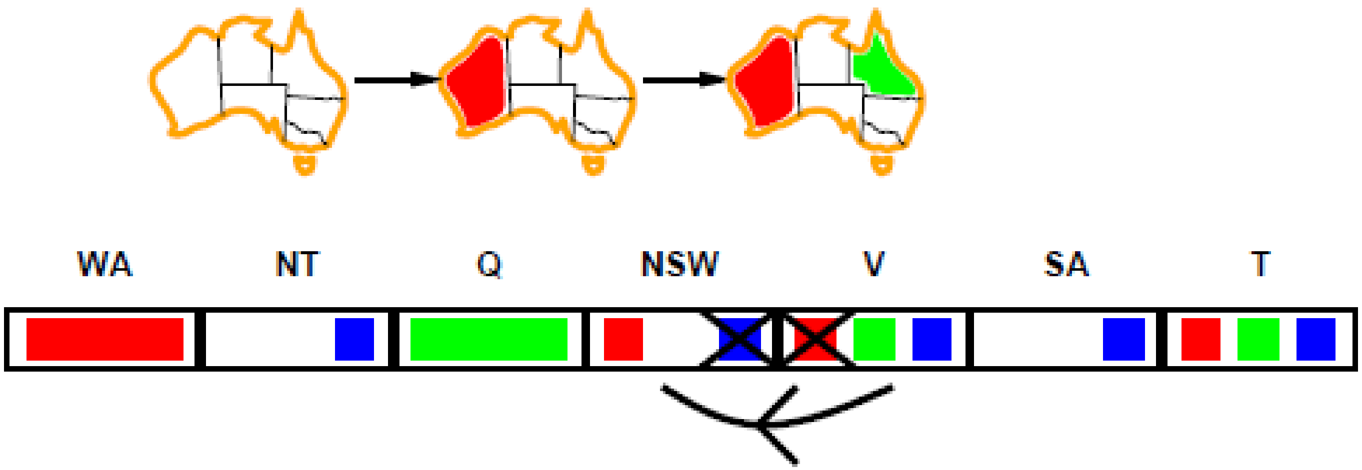


Arc consistency

Simplest form of propagation makes each arc consistent

$X \to Y$ is consistent iff

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

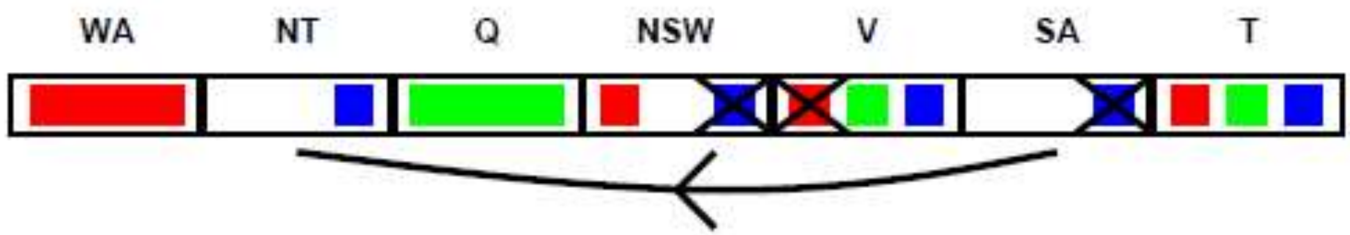


If X loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
   for **every** value x of X there is **some** allowed y



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

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

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

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

---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

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

AC-3 algorithm

We can represent the AC-3 algorithm in 3 steps:

1. Get all the constraints and turn each one into two arcs. For example:
   $A > B$ becomes $A > B$ and $B < A$.
2. Add all the arcs to a queue.
3. Repeat until the queue is empty:
   3.1. Take the first arc $(\boldsymbol{x}, \boldsymbol{y})$, off the queue (**dequeue**).
   3.2. For **every** value in the $\boldsymbol{x}$ domain, there must be some value of the $\boldsymbol{y}$ domain.
   3.3. Make $\boldsymbol{x}$ arc consistent with $\boldsymbol{y}$. To do so, remove values from $\boldsymbol{x}$ domain for which there is no possible corresponding value for $\boldsymbol{y}$ domain.
   3.4. If the $\boldsymbol{x}$ domain has changed, add all arcs of the form $(\boldsymbol{k}, \boldsymbol{x})$ to the queue (**enqueue**). Here $\boldsymbol{k}$ is another variable different from $\boldsymbol{y}$ that has a relation to $\boldsymbol{x}$.

Example

- Let’s take this example, where we have three variables A, B, and C and the constraints: $A > B$ and $B = C$.

- A={1,2,3}
- B={1,2,3}
- C={1,2,3}

- **Step 1: Generate All Arcs**

A>B

B<A

B=C

C=B

- **Step 2: Create the Queue**

Arcs

Queue:

A>B
B<A
B=C
C=B

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Queue:

A>B
B<A
B=C
C=B

A={1,2,3}
B={1,2,3}
C={1,2,3}

Arcs

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Queue:

~~A>B~~
B<A
B=C
C=B

A={2,3}
B={1,2,3}
C={1,2,3}

Arcs

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Arcs

Queue:

~~A>B~~
B<A
B=C
C=B

A={2,3}
B={1,2,3}
C={1,2,3}

Arcs
A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Arcs

Queue:

Queue:
~~A>B~~
~~B<A~~
B=C
C=B

A={2,3}
B={1,2}
C={1,2,3}

Arcs
A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Queue:

~~**A>B**~~
~~**B<A**~~
B=C
C=B
A>B

A={2,3}
B={1,2}
C={1,2,3}

Arcs

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Queue:

~~A>B~~
~~B<A~~
~~B=C~~
C=B

A={2,3}
B={1,2}
C={1,2,3}

Arcs

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Arcs

Queue:

~~A>B~~
~~B<A~~
~~B=C~~
C=B

A={2,3}
B={1,2}
C={1,2,3}

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Queue:

~~A>B~~
~~B<A~~
~~B=C~~
~~C=B~~

A={2,3}
B={1,2}
C={1,2}

Arcs

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Arcs

Queue:

~~A>B~~
~~B<A~~
~~B=C~~
~~C=B~~

A={2,3}
B={1,2}
C={1,2}

A>B
B<A
B=C
C=B

- **Step 3: Iterate Over the Queue**
-

Arcs

Queue:

~~A>B~~
~~B<A~~
~~B=C~~
~~C=B~~

A={2,3}
B={1,2}
C={1,2}

A>B
B<A
B=C
C=B

- **Step 4: Stop if no more arc in the queue**
-

Queue:

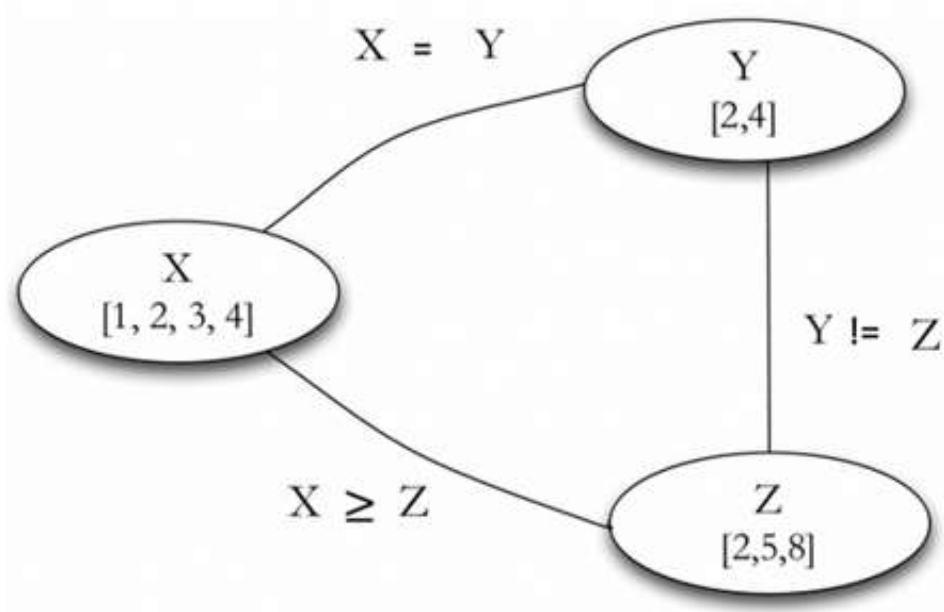
~~A>B~~
~~B<A~~
~~B=C~~
~~C=B~~

A={2,3}
B={1,2}
C={1,2}

Arcs

A>B
B<A
B=C
C=B

Solve the following constraint graph using AC-3 Algorithm.

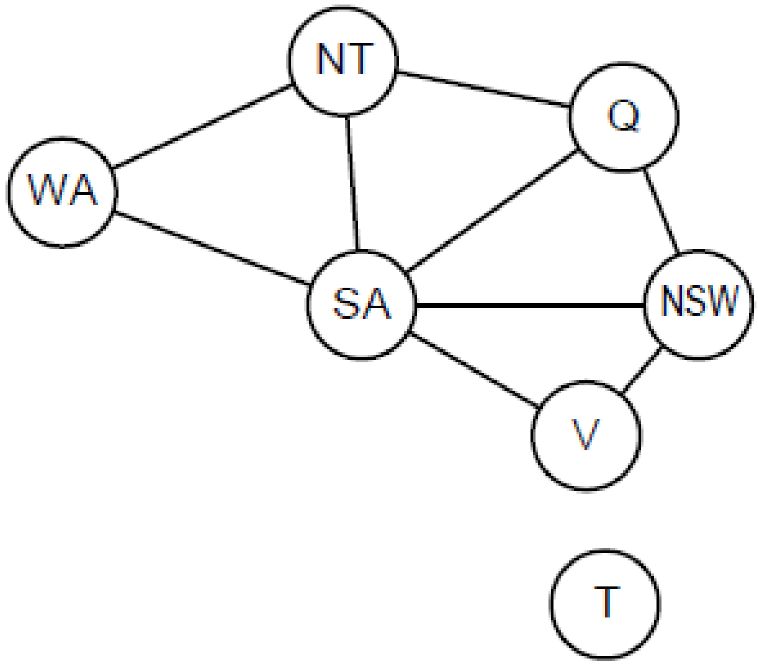


Variables: {X, Y, Z}
Domain values are given inside the node. for e.g. X={1,2,3,4}
Constraints are given at the edges of the graph. For e.g. {Y!=Z}

Note:

1.In the First step, you have to specify all the variables and constraints and write all the arcs in the initial state of the queue using LL fashion in alphabetical order (like dictionary-based) only.

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

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

Summary

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

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

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

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

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice