Constraint Satisfaction Problems

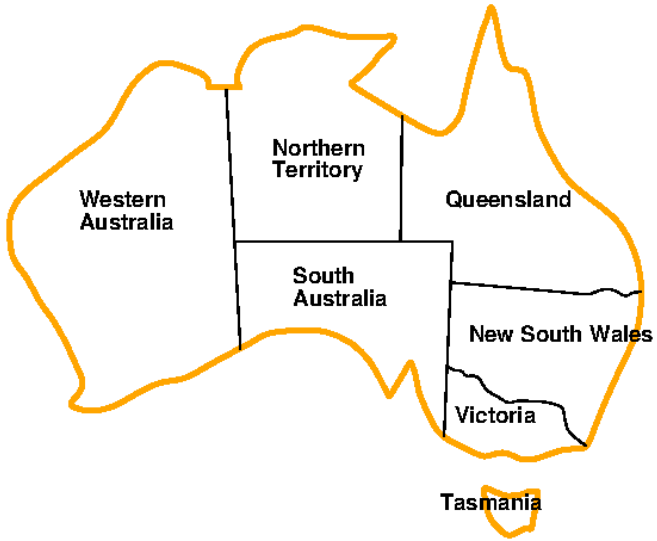
AI 2025/2026

Content

Constraint satisfaction problems (CSPs)

- Are defined by variables X_i with values from domain D_i, and a set of constraints specifying allowable combinations of values for subsets of variables
- An assignment is *consistent* if it does not violate any of the constraints.
  An assignment is *complete* if it includes all variables.
  Solution: a complete assignment of values to variables s.t. all constraints are satisfied.
  - *general-purpose* algorithms, more powerful than standard search algorithms
- MaxCSP: maximize the number of satisfied constraints

Example: Map-Coloring



Variables: WA, NT, Q, NSW, V, SA, T

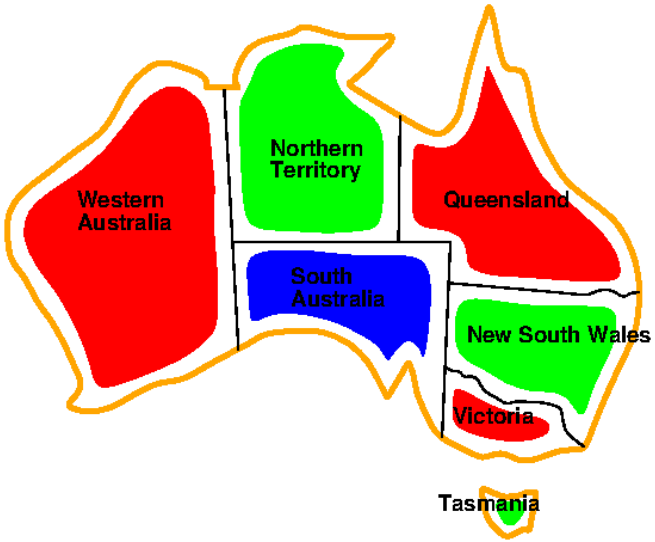
Domains: $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

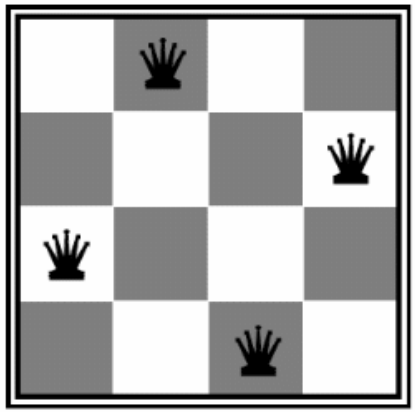
Example: Map-Coloring



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

Example: N-Queens

1st Model:



Variables: X_{ij}

Domains: $\{0, 1\}$

Constraints $\sum_{i,j} X_{ij} = N$

$$\forall i, j, k \quad (x_{ij}, x_{ik}) \in \{(0,0), (0,1), (1,0)\}$$

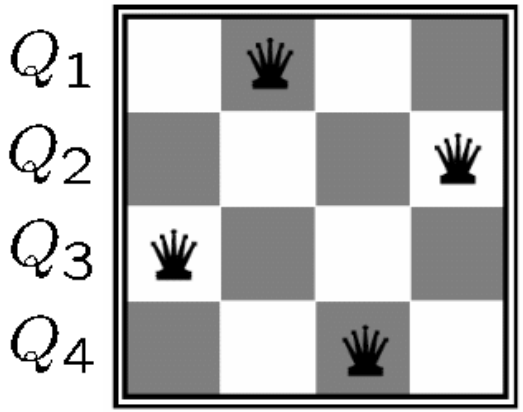
$$\forall i, j, k \quad (x_{ij}, x_{kj}) \in \{(0,0), (0,1), (1,0)\}$$

$$\forall i, j, k \quad (x_{ij}, x_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\forall i, j, k \quad (x_{ij}, x_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

Example: N-Queens

2nd Model:



Variables: Q_k
Domains: $\{1, 2, 3, ..., N\}$
Constraints
Implicit: $\forall i, j \quad non - threatening(Q_i, Q_j)$
Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), ...\}...$

Example: Sudoku

Variables: $x_{ij} \in \{1, ..., 9\} =: N$ (the values in the corresponding cells);
Pairwise inequality constraints: all values in a row, column, subsquare are different

$$x_{ij} \neq x_{ik} \qquad \forall k \neq i, j \in N, (rows)$$
$$x_{ij} \neq x_{kj} \qquad \forall k \neq i, j \in N, (columns)$$
$$x_{i_1 j_1} \neq x_{i_2 j_2} \qquad \forall (i_1, j_1) \neq (i_2, j_2) \in C_{ij}, \forall i, j \in N', (subsquares)$$
$$x_{ij} \in N \qquad \forall i, j \in N$$

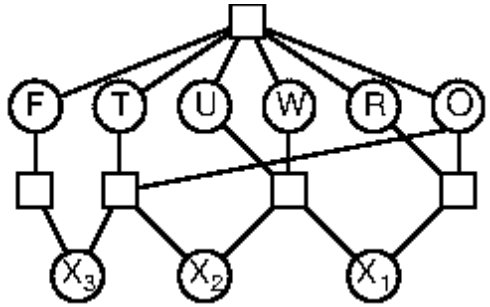
$$C_{ij} = \{(3(i-1) + i', 3(j-1) + j') | (i', j') \in N' x N'\}$$

Use the global `alldifferent` constraint for a more powerful formulation:

$$alldifferent(x_{ij} | j \in N) \ \forall i \in N, (rows)$$
$$alldifferent(x_{ij} | i \in N) \ \forall j \in N, (columns)$$
$$alldifferent(C_{ij}) \ \forall i, j \in N', (subsquares)$$
$$x_{i,j} \in N \ \forall i, j \in N$$

Example: Cryptarithmetic puzzle

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



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:

$alldiff(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

Example: Job-shop scheduling

Scheduling the assembly of a car:

- Tasks: install axles (front, back), affix wheels, tighten nuts for each wheel, affix hubcaps, and inspect the final assembly
  Model each task as a variable, the value is the time the task starts (minutes)
  $X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$

- Constraints:
  - a task must occur before another (a wheel must be installed before the hubcap)
    $T_1 + d_1 \leq T_2$ (T_1 must start before T_2)
  - a task takes a certain amount of time to complete

Example: Job-shop scheduling

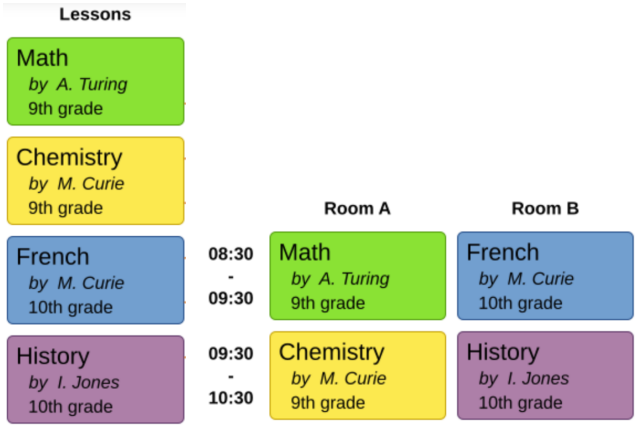
Scheduling the assembly of a car. Constraints:

- The axles have to be in place before the wheels are put on
  $Axle_F + 10 \leq Wheel_{RF}$
- After affixing the wheel, then tighten the nuts, and attach the hubcap
  $Wheel_{RF} + 1 \leq Nuts_{RF}$; $Nuts_{RF} + 2 \leq Cap_{RF}$
- To put the axle in place, share one tool ($Axle_F$, $Axle_B$ must not overlap)
  $(Axle_F + 10 \leq Axle_B)$ **or** $(Axle_B + 10 \leq Axle_F)$
- Inspection comes last and takes 3 minutes
  $X + d_X \leq Inspect$
- The assembly must be done in 30 minutes
  $D = \{1, 2, 3, ..., 27\}$

Example: timetable scheduling

Scheduling the timetable for teachers and students

- Teachers, courses, classes, classrooms, time intervals
  A teacher teaches certain courses
  A course is given to certain classes
- Constraints:
  - two courses of the same class cannot be scheduled at the same time
  - a teacher prefers some time intervals, etc.



Example: timetable scheduling

Sets: $t \in T$ time, $r \in R$ rooms, $c \in C$ courses, $s \in S$ students, $(c_i, c_j) \in SA$: c_i, c_j cannot overlap (the same teacher), z_s the courses assigned to group s

- Variabiles: $x_c = t$ course c is scheduled at t
  $y_c = r$ course c takes place in r
- Constraints:
  - $x_c = t$, for $c \in C$, ...
  - two courses cannot be held at the same time (same teacher)
    If $x_{c_1} = t_1$ *and* $(c_1, c_2) \in SA$ *then* $x_{c_2} \neq t_2$ *if overlap*(t_1, t_2)
  - two courses assigned to the same group of students cannot take place at the same time
    If $z_s = c_1$ *and* $z_s = c_2$ *and* $x_{c_1} = t_1$ *then* $x_{c_2} \neq t_2$ *if overlap*(t_1, t_2)
  - two courses cannot be scheduled at the same time in the same room
    If $(x_{c_1} = t_1)(x_{c_2} = t_2)(y_{c_1} = r_1)$ *then* $y_{c_2} \neq r_1$ *if overlap*(t_1, t_2)

Real-world CSPs

- Staff scheduling/nurse rostering problems
- Transportation scheduling, Factory scheduling, Floorplanning, etc.
- Meeting scheduling
- Assignment problems (who teaches, at what class)
- Hardware configuration, etc.

Notice that many real-world problems involve real-valued variables
`https://www.csplib.org/Problems/`

Varieties of CSPs

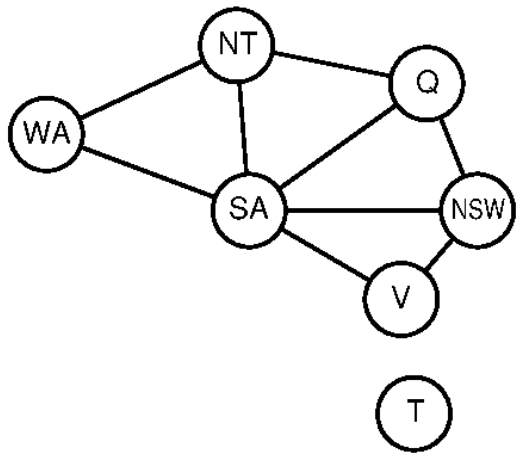
- Discrete variables (finite domains: size $d \implies O(d^n)$ complete assignments and infinite domains), continuous variables

- Constraints
  - Unary: involve a single variable
    ex: $SA \neq green$
  - Binary: involve pairs of variables
    ex: $SA \neq WA$
    Binary CSP: each constraint relates at most two variables
  - Higher-order: involve 3 or more variables
    ex: 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

Constraint graph

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!)

Content

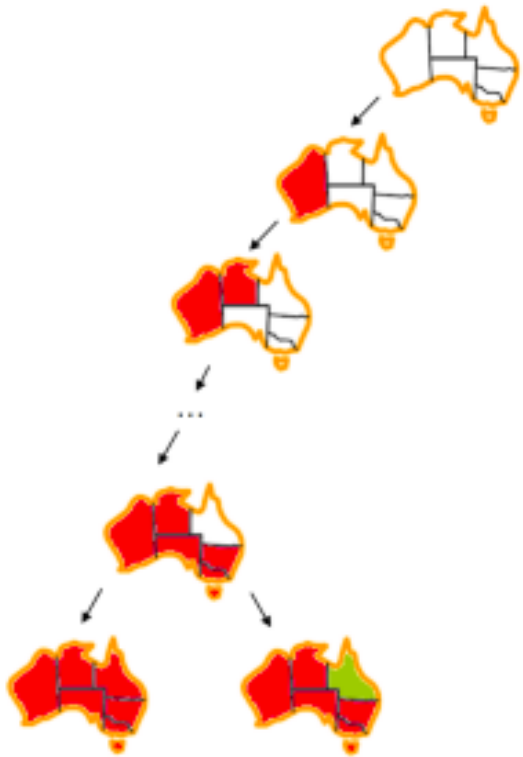
Standard search formulation (incremental)

The straightforward approach:
States are defined by the values assigned so far.

- Initial state: the empty assignment, $\emptyset$
- Successor function: assign a value to an unassigned variable
- Goal test: the current assignment is complete

Standard formulation (incremental search)

- Solutions are at depth n (n variables assigned)
  $\Longrightarrow$ use Depth-first search
- The branching factor is $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves, d^n possible assignments



Backtracking search

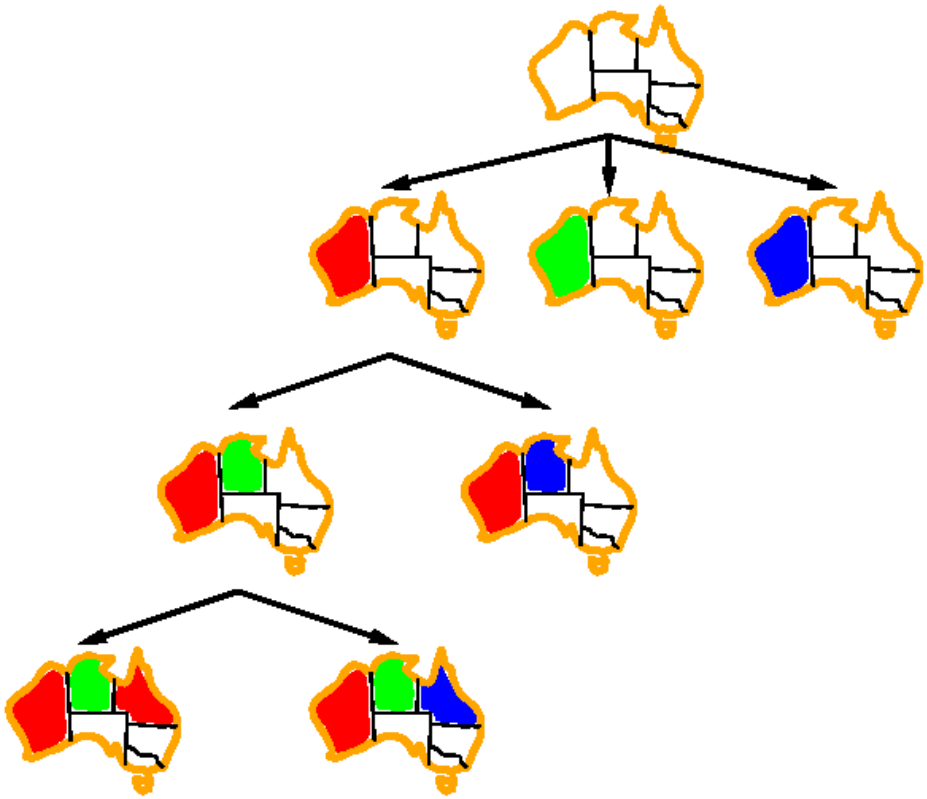
Variable assignments are commutative, i.e.,
$[WA = red \text{ then } NT = green]$ same as $[NT = green \text{ then } WA = red]$

At each node, consider the assignment to a single variable
(does not conflict with the current assignment)
$\Longrightarrow$ $b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments: backtracking search. Backtracking is the basic uninformed algorithm for CSPs.

Backtracking example

When a node is expanded, each subsequent state is checked for consistency before being added.



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 = DFS + variable-ordering + fail-on-violation

Can solve n-queens for $n \approx 25$.

Content

Improving backtracking efficiency

1. Which variable should be assigned next?
2. In what order should the values be verified?
3. Can the inevitable failure be detected earlier?
4. Can we take advantage of the problem structure?

1. Minimum remaining values

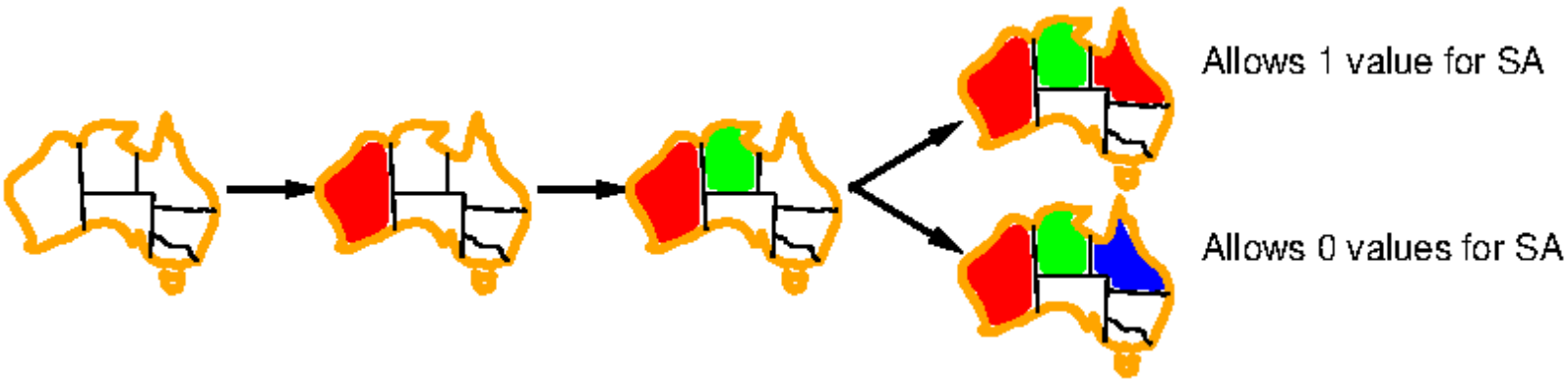
Minimum remaining values (MRV): choose the variable with the fewest legal values (most constrained variable)

"Fail-first" heuristic

2. Least constraining value

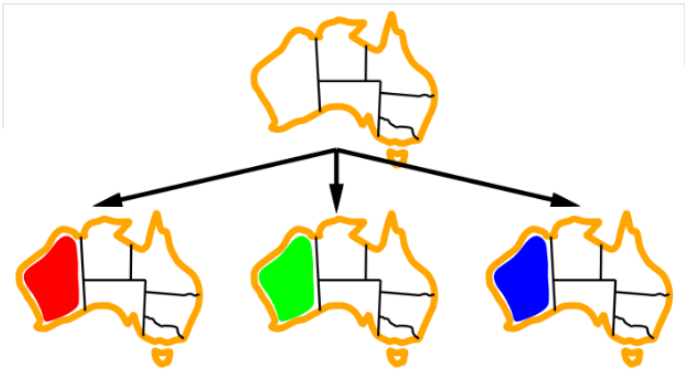
Least constraining value: choose the least constraining value
(the one that rules out the fewest values in the remaining variables)

Example: what value should we choose for Q?



Combining these heuristics makes 1000-queens feasible.

Questionnaire



Question!

Variable order WA, NT, Q, NSW, V, T, SA. Tightest upper bound on naïve backtracking search space size?

(A): 145 (B): 382

(C): 433 (D): 3^7

Questionnaire

Question!

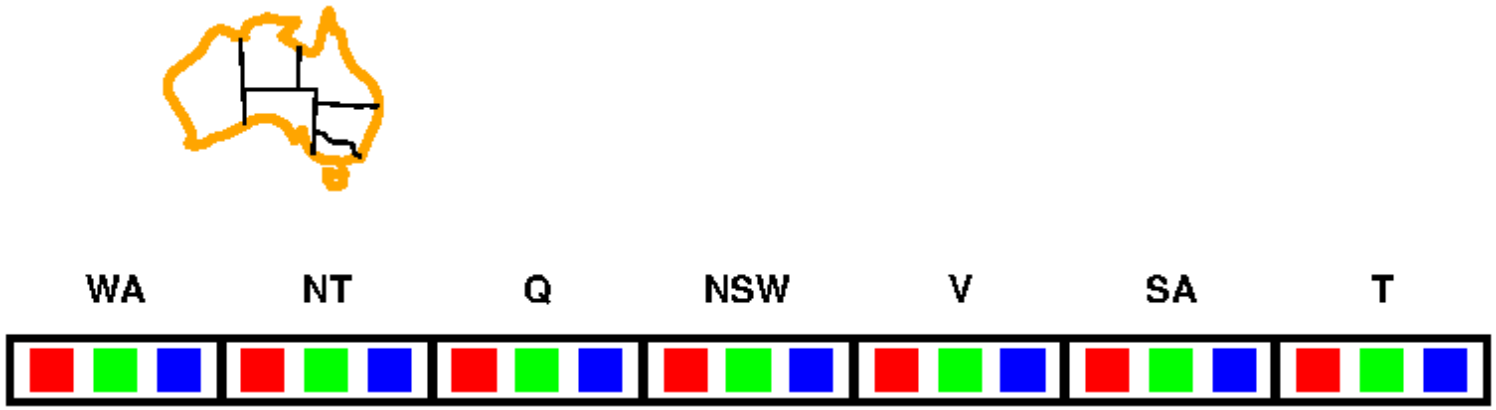
Variable order SA, NT, Q, NSW, V, WA, T. Tightest upper bound on naïve backtracking search space size?

(A): 52 (B): 145

(C): 382 (D): 433

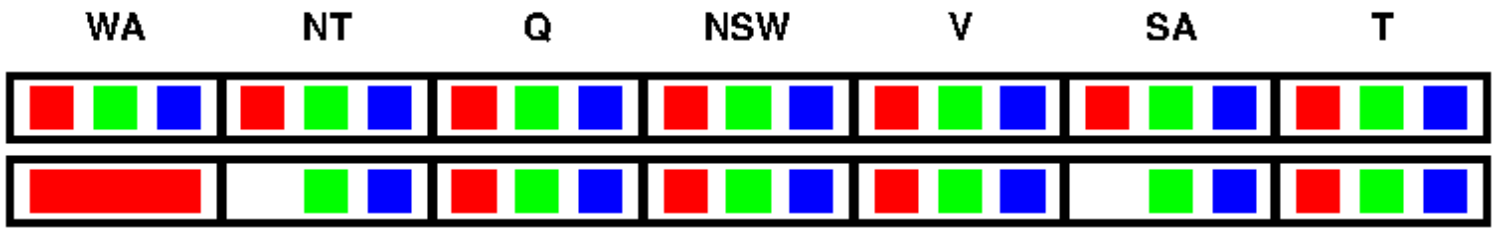
3a. Forward checking

Idea: update the domain of unassigned variables (when a node is expanded, eliminate values from unassigned variables which are connected with that node)



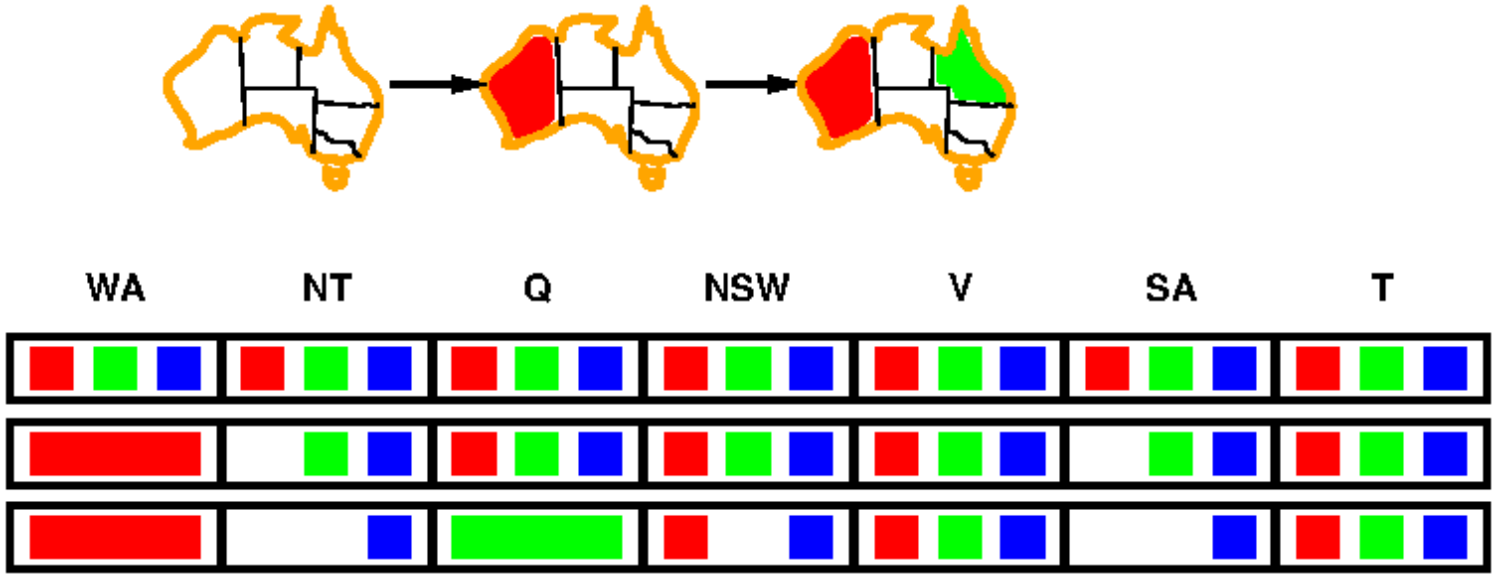
Forward checking

Idea: update the domain of unassigned variables (when a node is expanded, eliminate values from unassigned variables which are connected with that node)



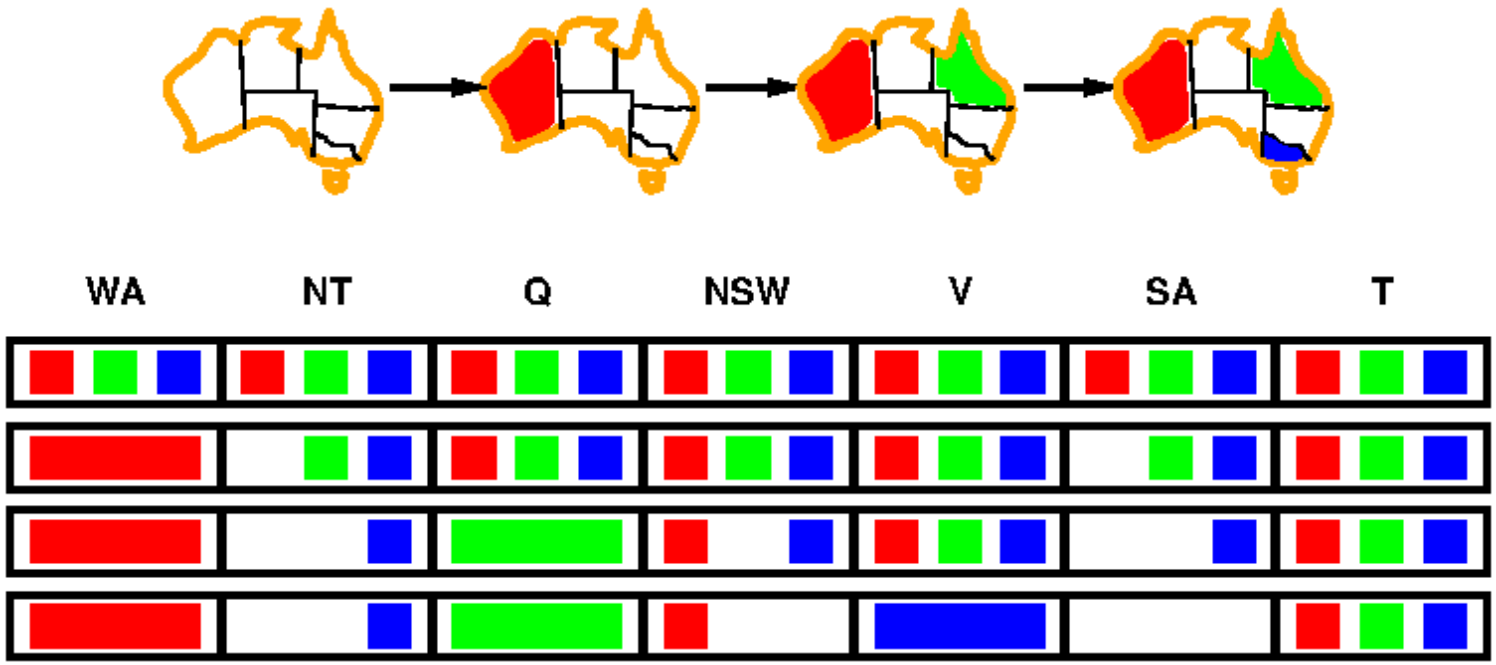
Forward checking

Idea: update the domain of unassigned variables (when a node is expanded, eliminate values from unassigned variables which are connected with that node)



Forward checking

Idea: update the domain of unassigned variables (when a node is expanded, eliminate values from unassigned variables which are connected with that node)



Forward checking

procedure SELECTVALUE-FORWARD-CHECKING

while D'_i is not empty
select an arbitrary element $a \in D'_i$, and remove a from D'_i
empty-domain $\leftarrow$ *false*
for all k, $i < k \leq n$
for all values b in D'_k
if not CONSISTENT$(\vec{a}_{i-1}, x_i{=}a, x_k{=}b)$
remove b from D'_k
end for
if D'_k is empty ($x_i = a$ leads to a dead-end)
empty-domain $\leftarrow$ *true*
if *empty-domain* (don't select a)
reset each D'_k, $i < k \leq n$ to value before a was selected
else
return a
end while
return null (no consistent value)
end procedure

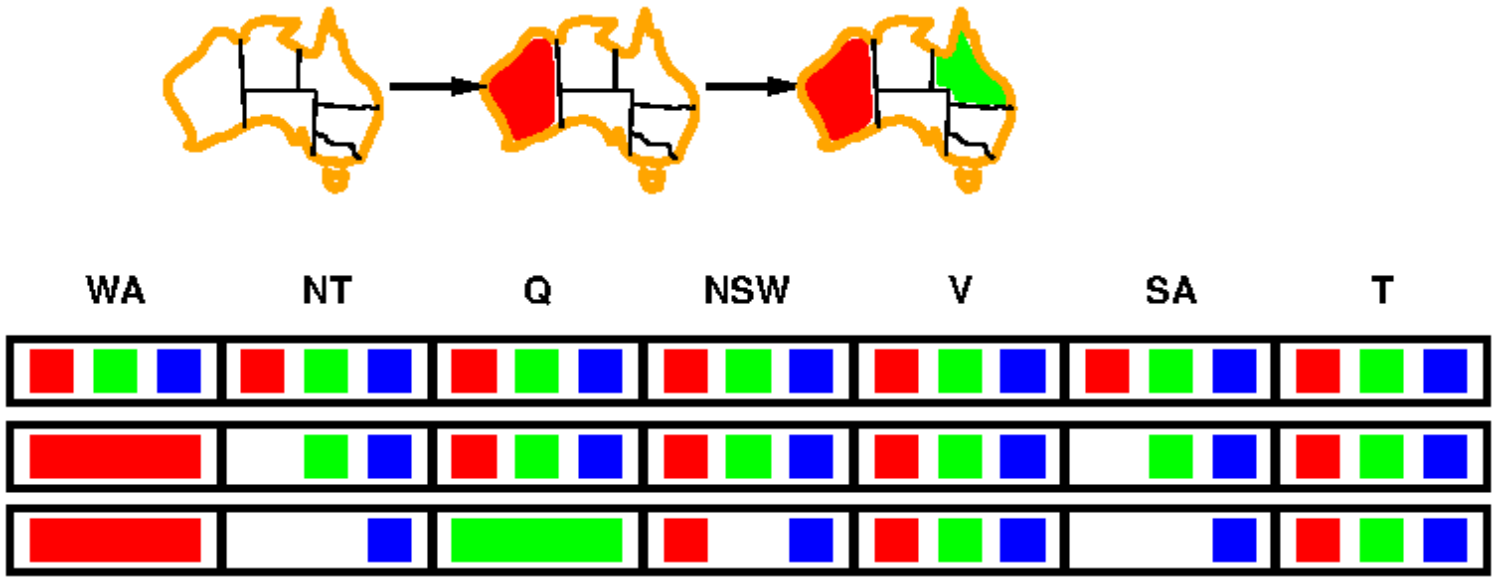
Generalized look ahead

procedure GENERALIZED-LOOKAHEAD
Input: A constraint network $P = (X, D, C)$
Output: Either a solution, or notification that the network is inconsistent.

```
D'_i ← D_i for 1 ≤ i ≤ n          (copy all domains)
i ← 1                            (initialize variable counter)
while 1 ≤ i ≤ n
   instantiate x_i ← SELECTVALUE-XXX
   if x_i is null                (no value was returned)
      i ← i − 1                  (backtrack)
      reset each D'_k, k > i, to its value before x_i was last instantiated
   else
      i ← i + 1                  (step forward)
end while
if i = 0
   return "inconsistent"
else
   return instantiated values of {x_1, ..., x_n}
end procedure
```

3b. Constraint propagation

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

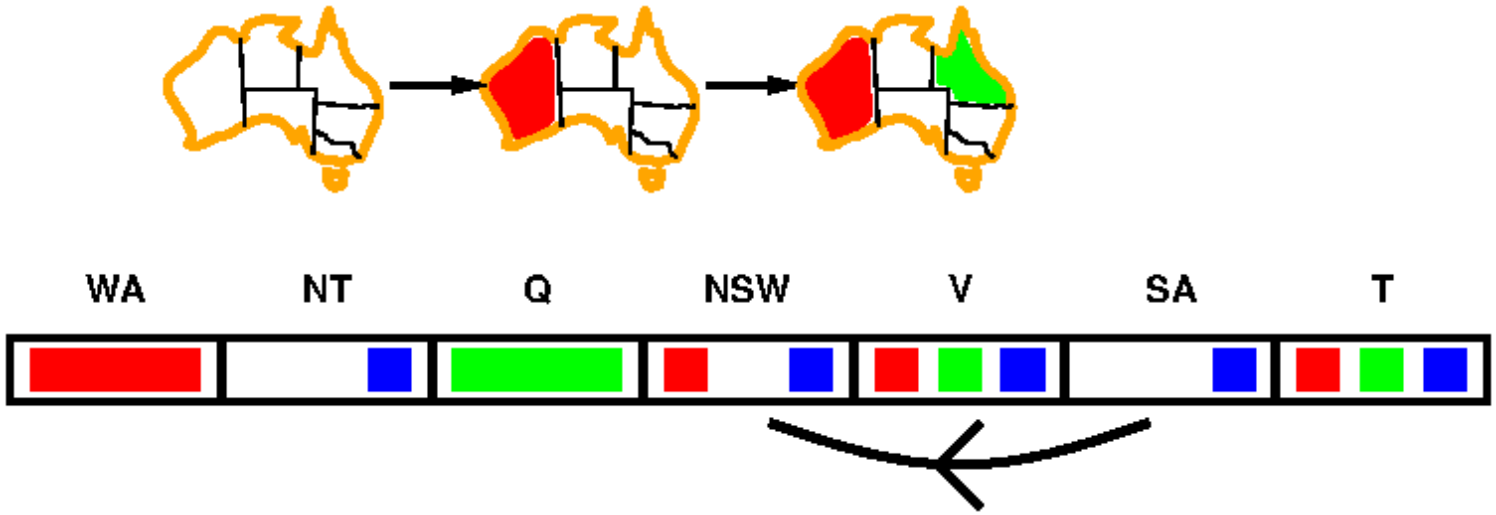


NT and *SA* cannot both be blue!
There is no propagation between unassigned variables!

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 of Y

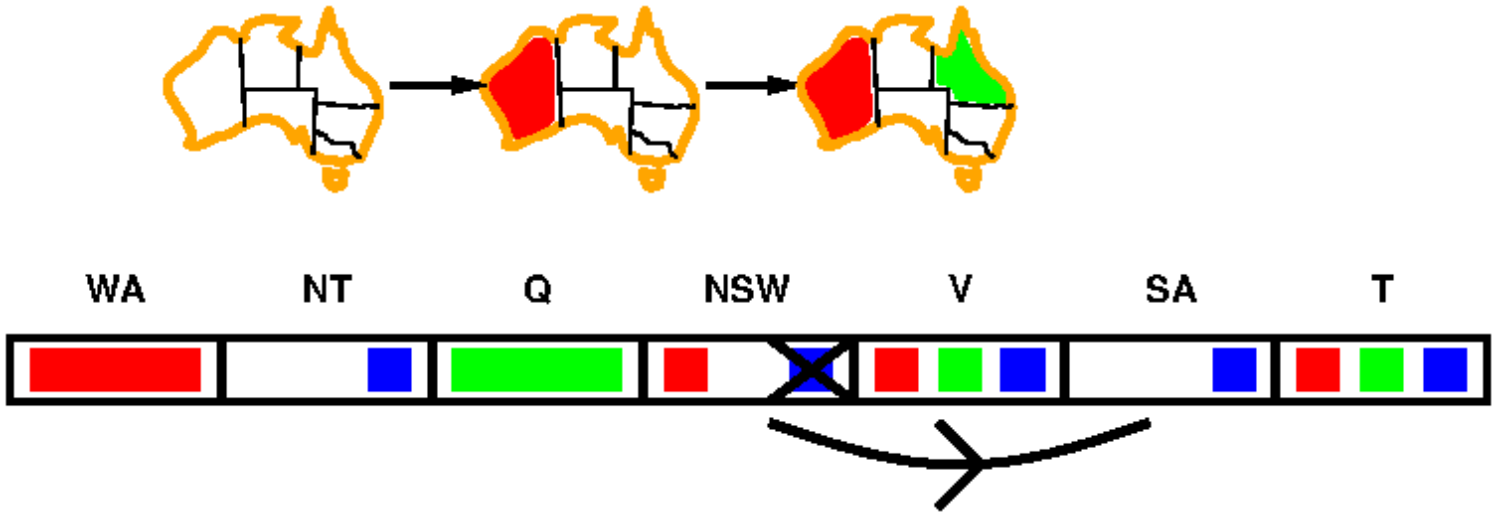


Arc consistency

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 of Y

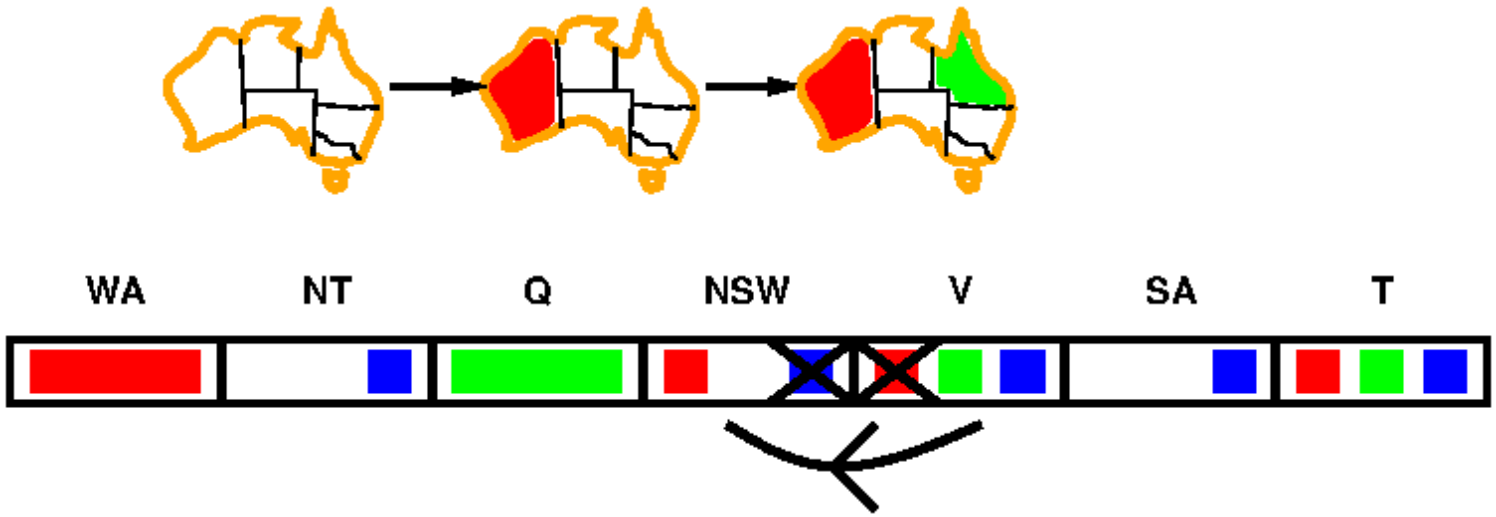


Arc consistency

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 of Y

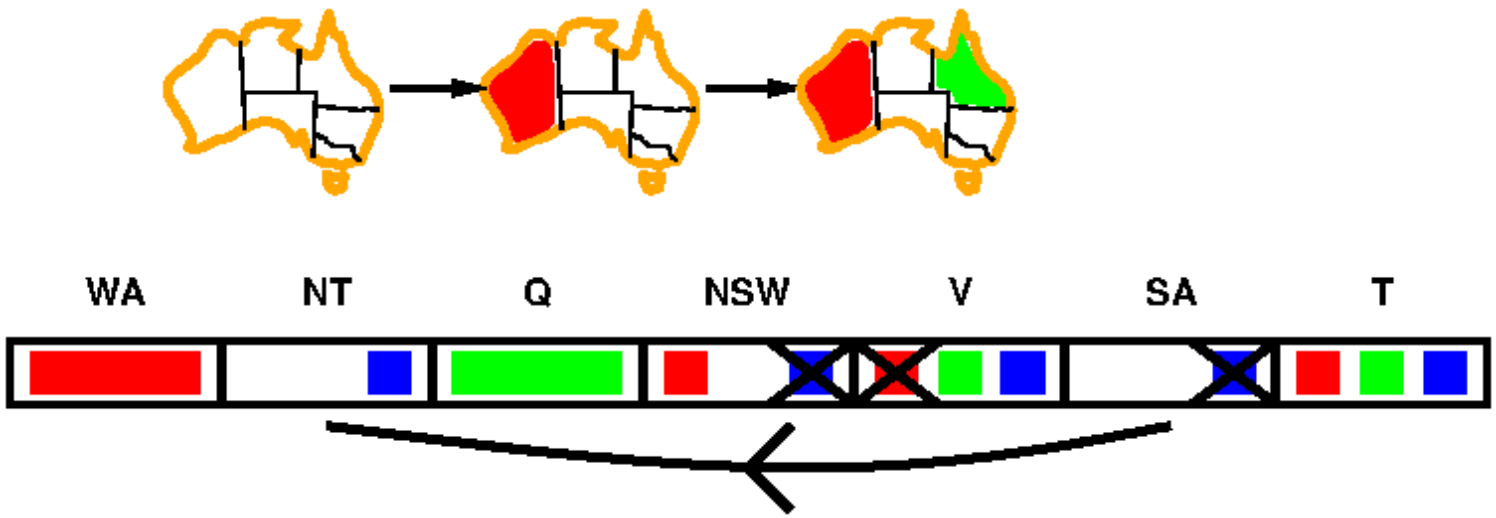


If X loses a value, neighbors of X need to be rechecked.

Arc consistency

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 preprocessing step 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 $O(n^2d^2)$

Constraint propagation

- 3-consistent (*Path consistency*), k-consistency
  k-consistency: any consistent assignment of $k-1$ variables can be extended to an instantiation of k variables

  If a CSP with n variables is n-consistent, then there is no need for backtracking.

- A large usage of constraint propagation techniques implies an increase in CPU time
  - a tradeoff between pruning and searching; if pruning takes longer than searching, it is not worth it

3c. Conflict-Directed Backjumping (CBJ)

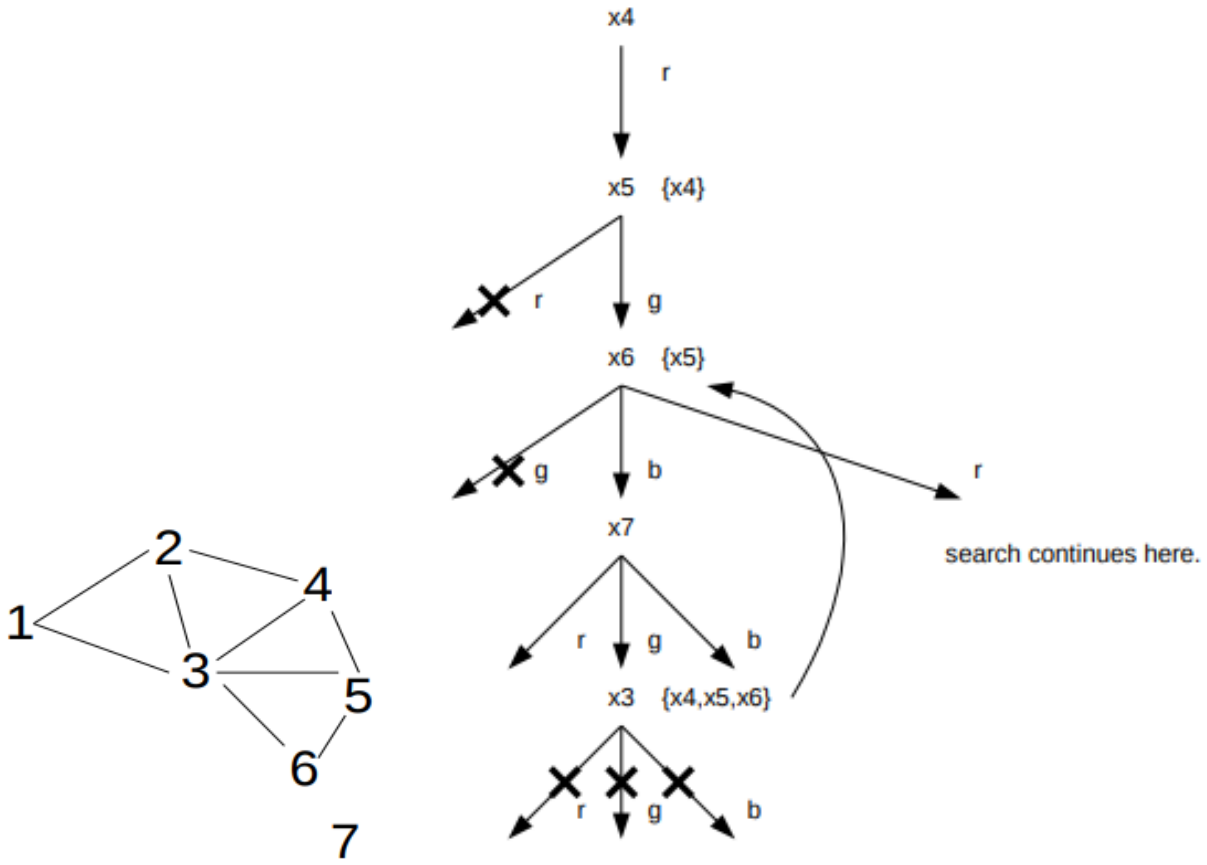
- Backtracking: backtracks to the first point where a variable can be assigned a new value
  - it backs up ONE level in the search tree at a time
- When we hit a dead end due to an inconsistency, we can try and deduce the reason for the problem
  - rather than backing up one level in the search tree, we can try to go directly to one of the variables that caused the problem

Conflict-Directed Backjumping

- Idea: Maintain a CONFLICT SET for every variable (updated as we assign values to variables)
- Assume we are setting X_i. The CONFLICT SET for X_i is the set of PREVIOUSLY ASSIGNED VARIABLES connected to X_i by a constraint.
- When no assignment is found for the current variable X_i, BACKJUMP to the deepest X_k in the conflict set of X_i
- Update the conflict set of X_k
  $CONFLICT\_SET(X_k) =$
  $CONFLICT\_SET(X_k) \cup CONFLICT\_SET(X_i) \setminus X_k$

Conflict-Directed Backjumping

Example: map coloring



X_7 is not in the conflict set of X_3, so back up to the closest variable from the conflict set (X_6)

Conflict-Directed Backjumping

procedure CONFLICT-DIRECTED-BACKJUMPING
Input: A constraint network $\mathcal{R} = (X, D, C)$.
Output: Either a solution, or a decision that the network is inconsistent.

$i \leftarrow 1$ (initialize variable counter)
$D'_i \leftarrow D_i$ (copy domain)
$J_i \leftarrow \emptyset$ (initialize conflict set)
while $1 \leq i \leq n$
  instantiate $x_i \leftarrow$ SELECTVALUE-CBJ
  if x_i is null (no value was returned)
    $iprev \leftarrow i$
    $i \leftarrow$ index of last variable in J_i (backjump)
    $J_i \leftarrow J_i \cup J_{iprev} - \{x_i\}$ (merge conflict sets)
  else
    $i \leftarrow i + 1$ (step forward)
    $D'_i \leftarrow D_i$ (reset mutable domain)
    $J_i \leftarrow \emptyset$ (reset conflict set)
end while
if $i = 0$
  return "inconsistent"
else
  return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

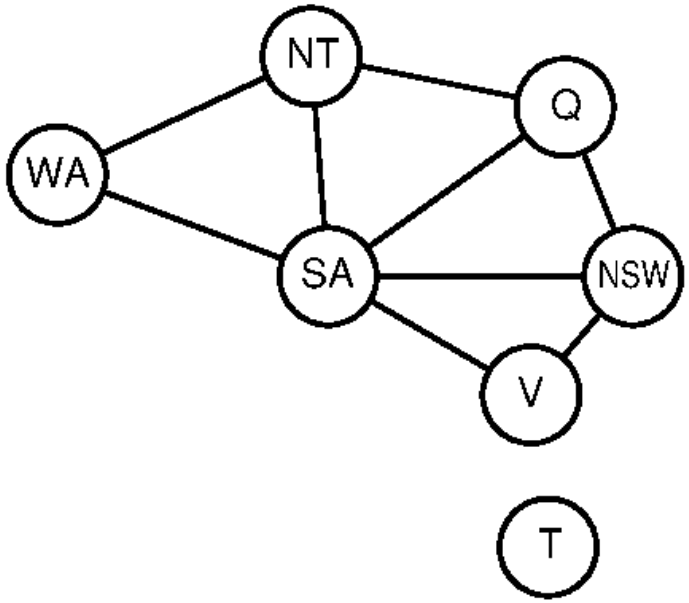
Conflict-Directed Backjumping

subprocedure SELECTVALUE-CBJ

while D'_i is not empty
select an arbitrary element $a \in D'_i$, and remove a from D'_i
consistent $\leftarrow$ *true*
$k \leftarrow 1$
while $k < i$ and *consistent*
if CONSISTENT$(\vec{a}_k, x_i = a)$
$k \leftarrow k + 1$
else
let R_S be the earliest constraint causing the conflict
add the variables in R_S's scope S, but not x_i, to J_i
consistent $\leftarrow$ *false*
end while
if *consistent*
return a
end while
return null (no consistent value)
end procedure

Content

Problem structure



Tasmania and mainland are independent subproblems, identifiable as connected components of the constraint graph

Problem structure

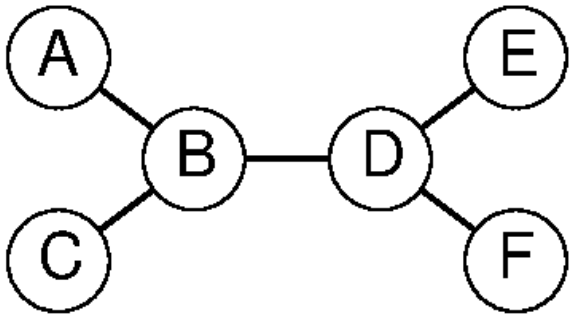
Suppose each subproblem has c variables, out of n total.

Worst-case solution cost is $n/c \cdot d^c$, *linear* in n.

Example: $n = 80$, $d = 2$, $c = 20$
$2^{80} = 4$ billion years at 10 million nodes/sec vs.
$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

Rare cases.

Tree-structured CSPs



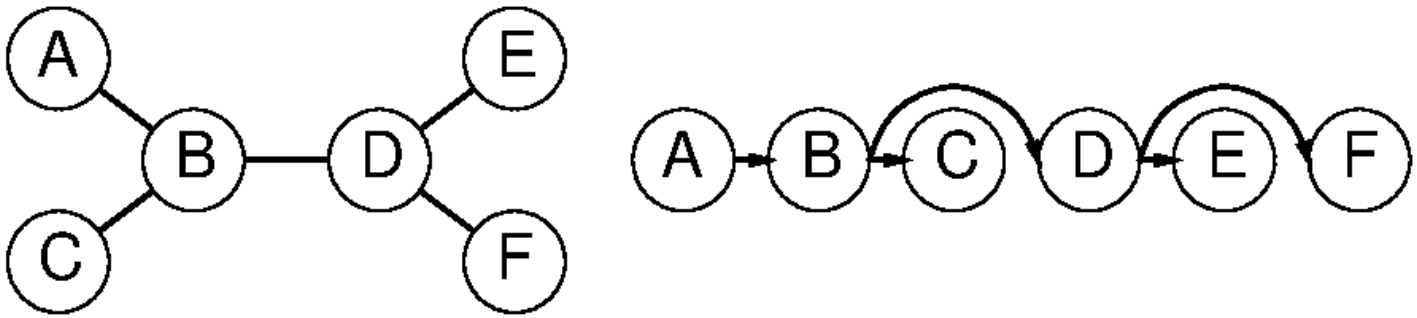
Theorem: if the constraint graph has no cycles, the CSP can be solved in $O(n\,d^2)$ time.

(Compare to general CSPs, where worst-case time is $O(d^n)$)

Algorithm for tree-structured CSPs

A CSP problem is *directed arc-consistent* for an ordering of variables $X_1, X_2, ..., X_n$ iff each X_i is arc-consistent with $X_j, \forall j > i$.

Method:
1. Choose a variable as root, order variables from root to leaves s.t. every node's parent precedes it in the ordering

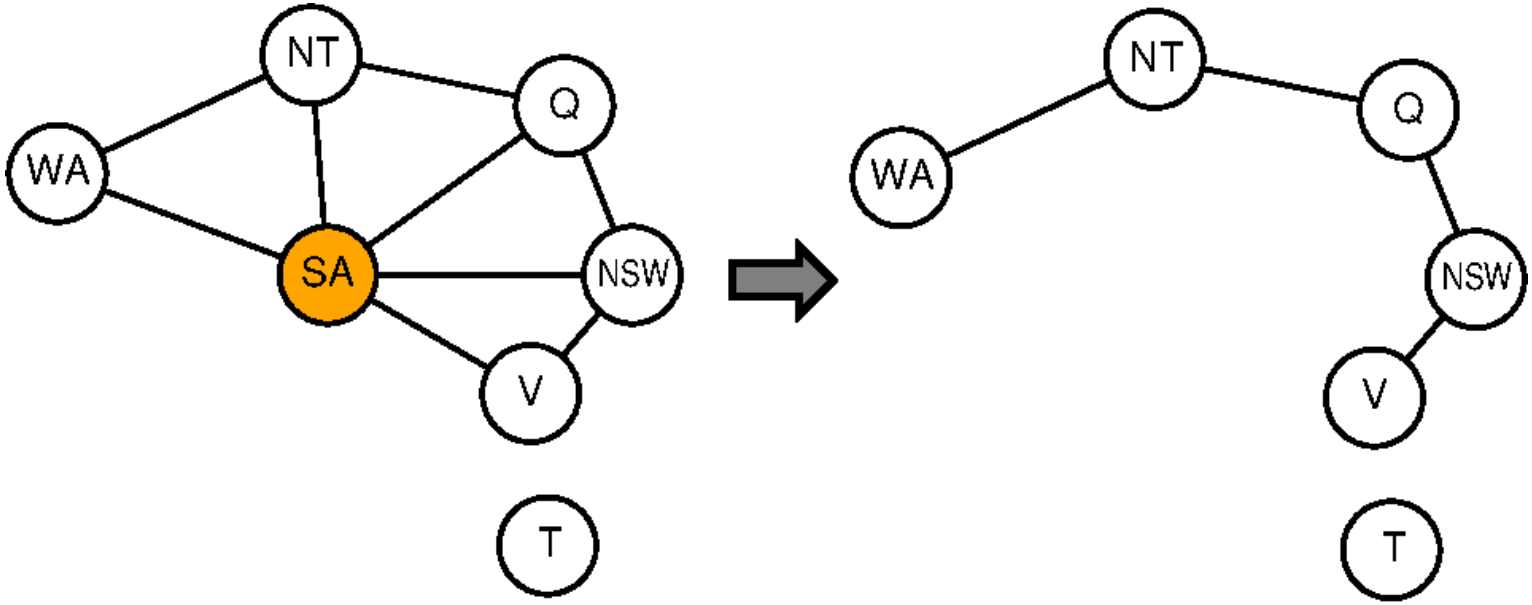


2. For j from n down to 2, apply *Make-Arc-Consistent*$(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j (consistent with $Parent(X_j)$)

Nearly tree-structured CSPs

Cutset conditioning



- Choose a subset of variables S s.t. the constraint graph is a tree after deleting S (S *cutset*)
- For each possible assignment of variables from S, delete from the domain of other variables values that are inconsistent with the assignment; return the two solutions.

Runtime: $O(d^c \cdot (n-c)d^2)$, c cutset size (very fast for small c)

Content

Iterative algorithms for CSPs

Hill-climbing, *simulated annealing* typically work with "complete" states (all variables assigned)

To apply to CSPs:
- allow states with unsatisfied constraints
- operators *assign* new values for vars

Variable selection: randomly select any conflicted variable

Value selection by min-conflicts heuristic:
- choose the value that violates the fewest constraints
- min $h(s)$ = total number of violated constraints

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

  $current \leftarrow$ 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*

Example: 4-Queens as a CSP

Assume one queen in each column.

Variables: Q_1, Q_2, Q_3, Q_4

Domains: $D_i = \{1, 2, 3, 4\}$

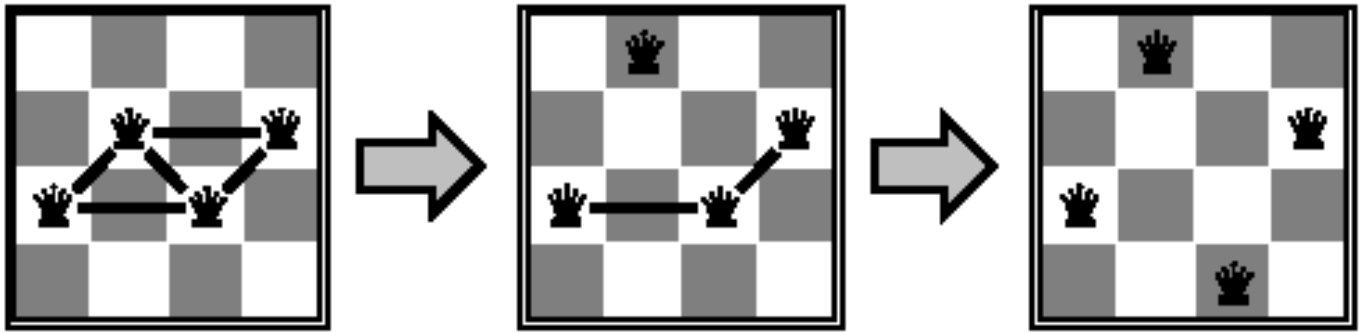
Constraints:

$Q_i \neq Q_j$ (cannot be in the same row)

$|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

Example: 4-Queens

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



Comparison of algorithms

The average number of consistency checks needed to solve the problem

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Summary

- Constraint Satisfaction Problems
  states: assignments of variables
  constraints on variables
- Backtracking = Depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics
- *Forward checking* prevents assignments that guarantee later failure. Constraint propagation (*Arc consistency*) does additional work to constrain values and detect inconsistencies.
- The constraint graph can be used to analyse the problem structure. Tree-structured CSPs can be solved in linear time.
- Iterative *Min-conflicts* is usually effective in practice.

Bibliography

- S. Russell, P. Norvig. Artificial Intelligence: A Modern Approach (3rd Edition). Prentice-Hall, Englewood Cliffs, NJ, 2010 (6. Constraint Satisfaction Problems)
- R. Bartak. Constraint Programming
- Solvers: CP Optimizer, OR-Tools