UNIVERSITÉ
JEAN MONNET
SAINT-ÉTIENNE

Introduction to AI

(Part 4: CLP Constraint Logic Programming)

Constraint Logic Programming

- --- built on top of Prolog's foundations
- Developed by Jaffar and Lassez at Monash University in Melbourne, Australia (1992)
- Includes domain-specific constraint solvers to augment the logical deduction algorithm
- Different domains are targeted with different specialized solvers
  - CLP(FD), for finite domains
  - CLP(R), for real number

Prolog example

```
solution(X,Y,Z) :- p(X),p(Y),p(Z),test(X,Y,Z).
p(11).
p(3).
p(7).
p(16).
p(15).
p(14).
test(X,Y,Z) :- Y is X+1,Z is Y+1.

solution(X,Y,Z)?
X=14; Y=15; Z=16 ?
no
```

Efficiency?

- one (and only) solution to the previous Prolog program?
  - “
    solution and a total of Y steps to exhaust the search space.”
  - You can also count the number of failure nodes

The problem is…

- Prolog has an extremely limited knowledge of mathematics
  - It leads to a big search space over only six possible integer values!
- It checks to see if the formulae hold, but it doesn't think about them as mathematical formulae nor does it manipulate them as math

Speeding up the earlier example: reordering conjuncts

```
solution(X,Y,Z) :- test(X,Y,Z),p(X),p(Y),p(Z).
p(11).
p(3).
p(7).
p(16).
p(15).
p(14).
test(X,Y,Z) :- Y is X+1,Z is Y+1.

solution(X,Y,Z)?
```

This fails, since X is uninstantiated in test
NB: +evaluable)

CLP

- CLP essentially merges logic programming with constraint solving
- Constraint solving is much in the spirit of logic programming, allowing a two-way flow of computation
  - But the domains are not limited to relations
  - Borning's Thinglab (http://www.thinglab.com.au/) is a classic example of a system based on constraint solving
    - "here's a polygon in which I always want the opposite sides to be parallel to each other."
    - "keep point M as the midpoint of the line defined by points A and B."

Solvers

- Underneath any constraint-based system is a constraint solver that takes equations and solves them (preferably quickly)
- The constraint satisfaction algorithms used depend on the domain over which the constraints are defined
  - For reals, common algorithms include gauss and simplex methods
- To become truly facile at CLP for a given domain one has to become knowledgeable about the solvers

CLP does “more”

- The reason CLP can do “more” than logic programming is that the elements have semantic meaning
  - in CLP(R), they are real numbers
  - In logic programming they were just strings to which you associated some meaning
- That is, CLP can, in general, manipulate symbolic expressions, too
- To do this, CLP(R) has to understand numbers, equations, arithmetic, etc.

A CLP(R) example

```
p(X,Y,Z) :- Z = X + Y.
p(3,4,Z)?
Z=7

p(X,4,7)?
X=3

p(X,Y,7).
X = -Y + 7 // instead of returning
           //multiple answers
```

The example in CLP(R): replace is with =

```
solution(X,Y,Z) :- test(X,Y,Z),p(X),p(Y),p(Z).
p(11).
p(3).
p(7).
p(16).
p(15).
p(14).
test(X,Y,Z) :- Y = X+1,Z = Y+1.

solution(X,Y,Z)?
X=14;Y=15;Z=16;
NO
•
```

Furthermore

```
solution(X,Y,Z) :-
  test(X,Y,Z),p(X),p(Y),p(Z).

test(X,Y,Z) :- Y = X+1,Z = Y+1.

solution(A,B,C)?
B = C - 1
A = C - 2
```

Fibonacci: Prolog vs. CLP(R)

```
fib(0,0).
fib(1,1).
fib(N,F) :-
  N > 1, N1 is N-1, N2 is
  N-2,
      fib(N1,F1),
      fib(N2,F2),
  F is F1 + F2.

fib(10,L)?
fib(N,55)?
  // instantiation error
```

```
fib(0,0).
fib(1,1).
fib(N,F1 + F2) :-
  N > 1,
      fib(N-1,F1),
      fib(N-2,F2).

fib(10,L)?
fib(N,55)?
fib(X,X)?   //0,1,5
```

Outline

1. **Constraint Satisfaction Problems (CSP)**
2. Solving CSP
   - Backtracking search for CSPs
   - Improving the search (forward checking, arc/ node consistency)
3. Writing a constraint solver in Prolog
4. CLP(FD) with Gnu-Prolog

Constraint satisfaction problems (CSPs)

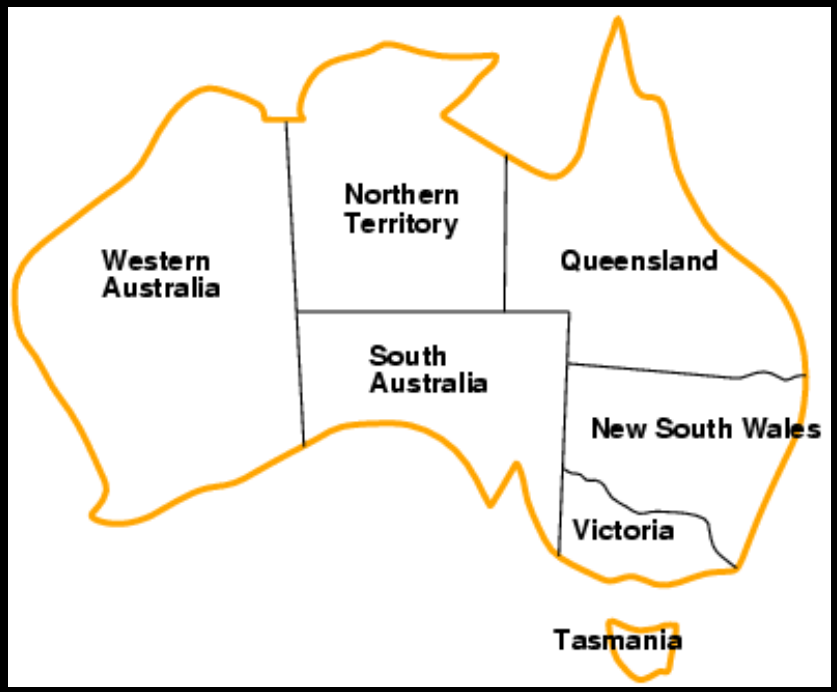
- Standard search problem:
  - is a "black box“ – any data structure that supports successor function, heuristic function, and goal test
- CSP:
  - is defined by 1) variables $_i$ with values from 2) domain D_i
  - is a set of 3) constraints specifying allowable combinations of values for subsets of variables
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

CSP Solution

- An *evaluation* of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains.

- An evaluation is *consistent* if it does not violate any of the constraints. An evaluation is *complete* if it includes all variables. An evaluation is a *solution* if it is consistent and complete; such an evaluation is said to *solve* the CSP.

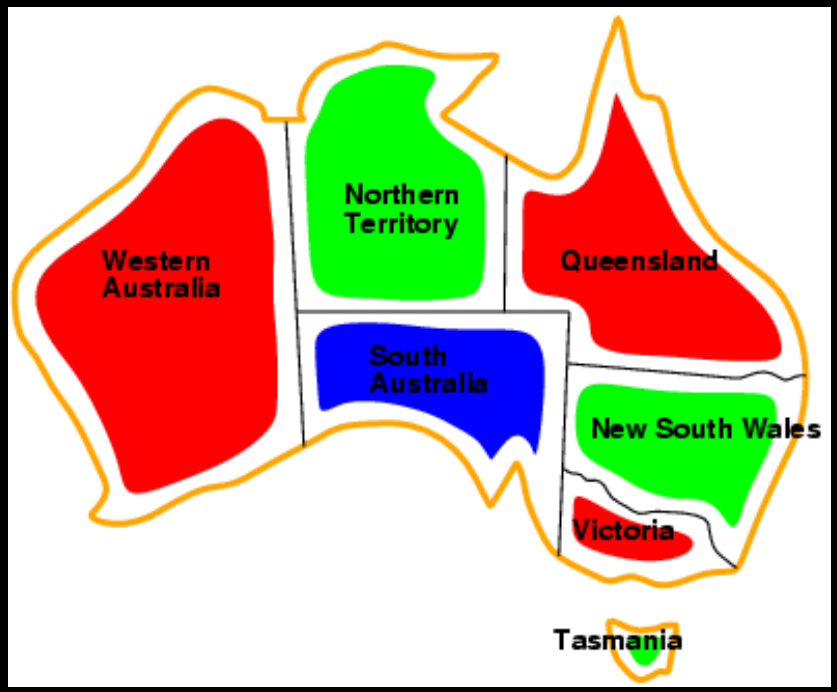
Example: Map-Coloring



- *WA, NT, Q, NSW, V, SA, T*
- D_i
-

- e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue), (blue,red),(blue,green)}

Example: Map-Coloring



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

Varieties of CSPs

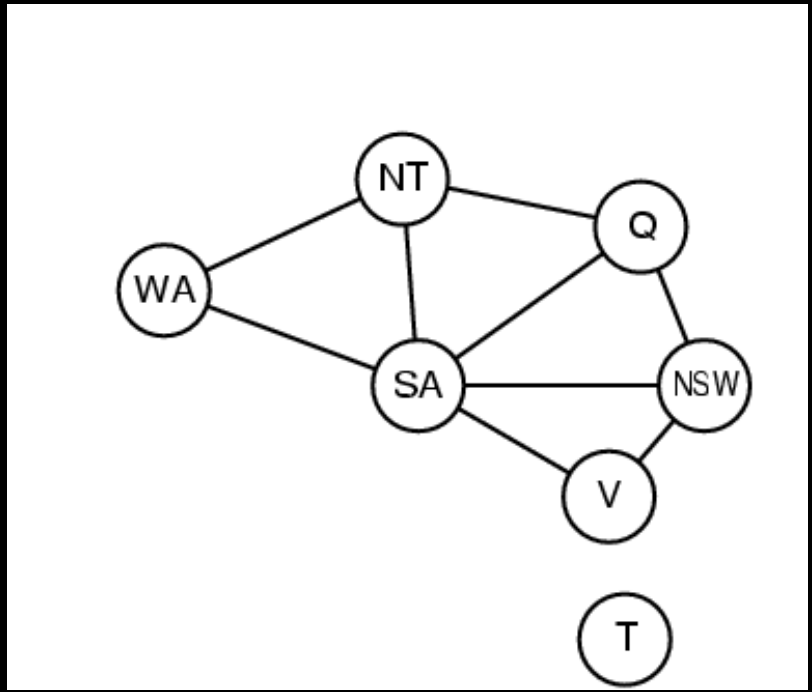
- Discrete variables
  - finite domains:
    - → d^n)
    - incl.~Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Varieties of constraints

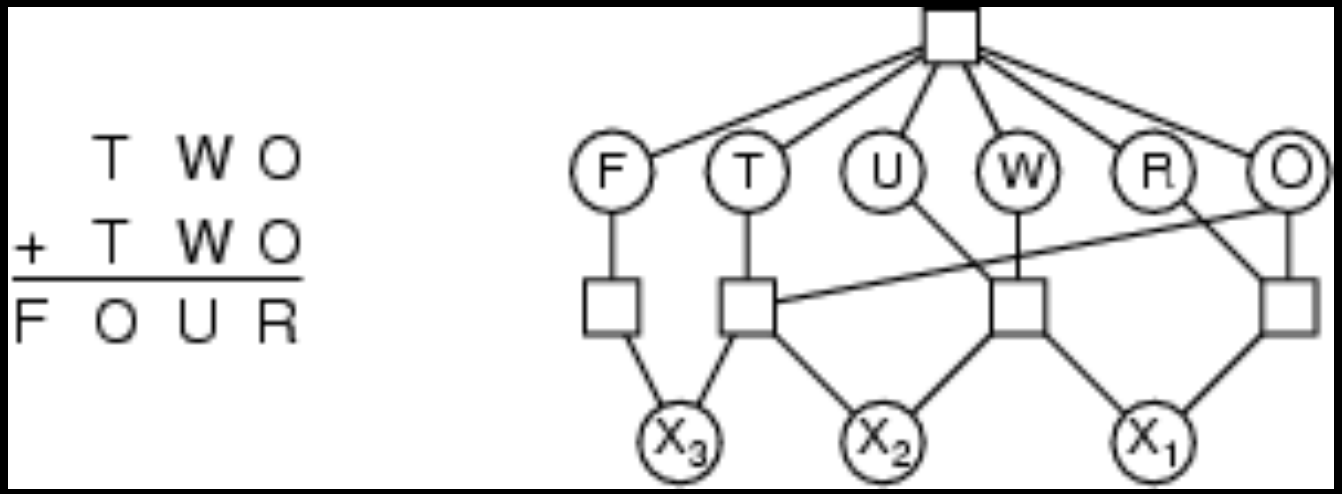
- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

Constraint graph

-
- Constraint graph: nodes are variables, arcs are constraints



Example: Cryptarithmetic



- Variables:
- Domains:
- Constraints:

Money changing problem

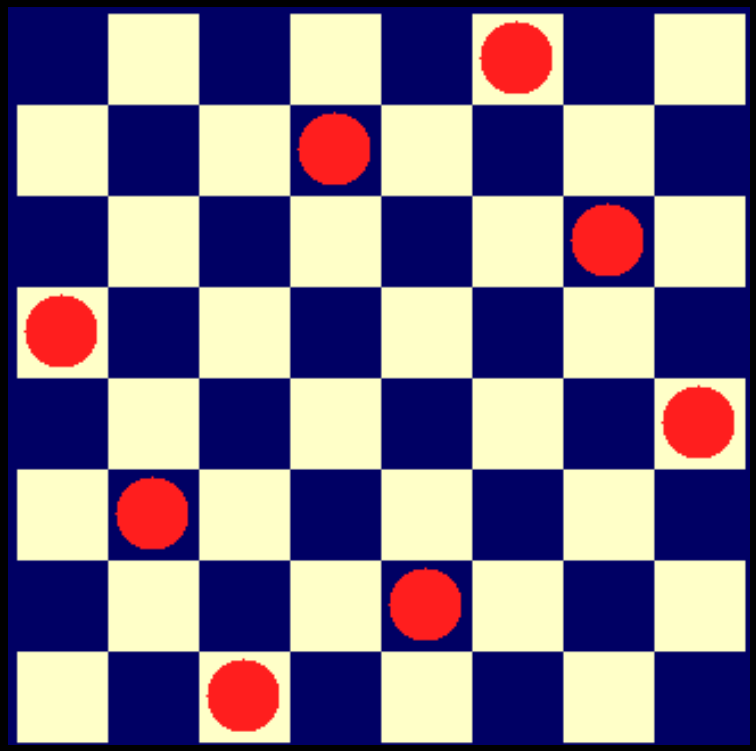
You want to buy a drink at a vending machine. You insert T euro cents. You select a drink which cost P euro cents (P and T are multiple of 10). How much change should the machine give you back if you know that its stock contains:

- _
- _
- _
- _
- _

1. Model this problem as a CSP
2. How can you state that you want has few coins as possible back?

Ex: N-Queens Problem

- The problem is to put N queens on a board of NxN such that no queen attacks any other queen.



- A queen moves vertically, horizontally and diagonally

Modeling N-Queens Problem

- The queens problem can be modeled (for example) via the following CSP
  - Variables=$\{Q_1,Q_2,Q_3,Q_4,\ldots,Q_N\}$.
  - Domain={1,2,3,…,N} represents the column in which the variables can be.
  - Constraints
    - Queens not on the same row: already taken care off by the good modeling of the variables.
    - Queens not on the same column: $Q_i \neq Q_j$
    - Queens not on the same diagonal: $|Q_i-Q_j| \neq |i-j|$

Ex: Magic square problem

- A magic square of size N is an NxN square filled with the numbers from 1 to N^2 such that the sums of each row, each column and the two main diagonals are equal.
- Example:

Ex: Uzbekian Puzzle

- An uzbekian sales man met five traders who live in five different cities. The five traders are:
  - {Abdulhamid, Kurban,Ruza, Sharaf, Usman}

The five cities are :

  - {Bukhara, Fergana, Kokand, Samarkand, Tashkent}

Find the order in which he visited the cities given the following information:

  - He met Ruza before Sharaf after visiting Samarkand,
  - He reached Fergana after visiting Samarkand followed by other two cities,
  - The third trader he met was Tashkent,
  - Immediately after his visit to Bukhara, he met Abdulhamid
  - He reached Kokand after visiting the city of Kurban followed by other two cities;

Modeling Uzbekian Puzzle

- The uzbekian puzzle can formulated within the CSP framework as follows:
  - Variables: order in which he visited each city and met each trader
  - Domain:1..5
  - constraints:
    - He met Ruza before Sharaf after visiting Samarkand,
    - He reached Fergana after visiting Samarkand followed by other two cities,
    - The third trader he met was Tashkent,
    - Immediately after his visit to Bukhara, he met Abdulhamid
    - He reached Kokand after visiting the city of Kurban followed by other two cities;

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

SOLVING CSP
(NOT NECESSARILY IN LOGIC)

Rqs

- CSP on finite domain only
- What we will **not** see:
  - Solving optimization pb under constraints (see « Machine Learning 2 »)
    - ex money changing pb
    - Branch and bound
  - Solving CSP over infinite domains
    - Ex: simplex (OR part)
    - Intervalle propagation
  - Incomplete algorithms
    - Find a good solution (not necessarily the best)
    - Local search, taboo search, simulated annealing, ant algorithms…

Standard search formulation (Generate and Test)

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

States are defined by the values assigned so far

- : the empty assignment { }
- : assign a value to an unassigned variable that does not conflict with current assignment
  → fail if no legal assignments

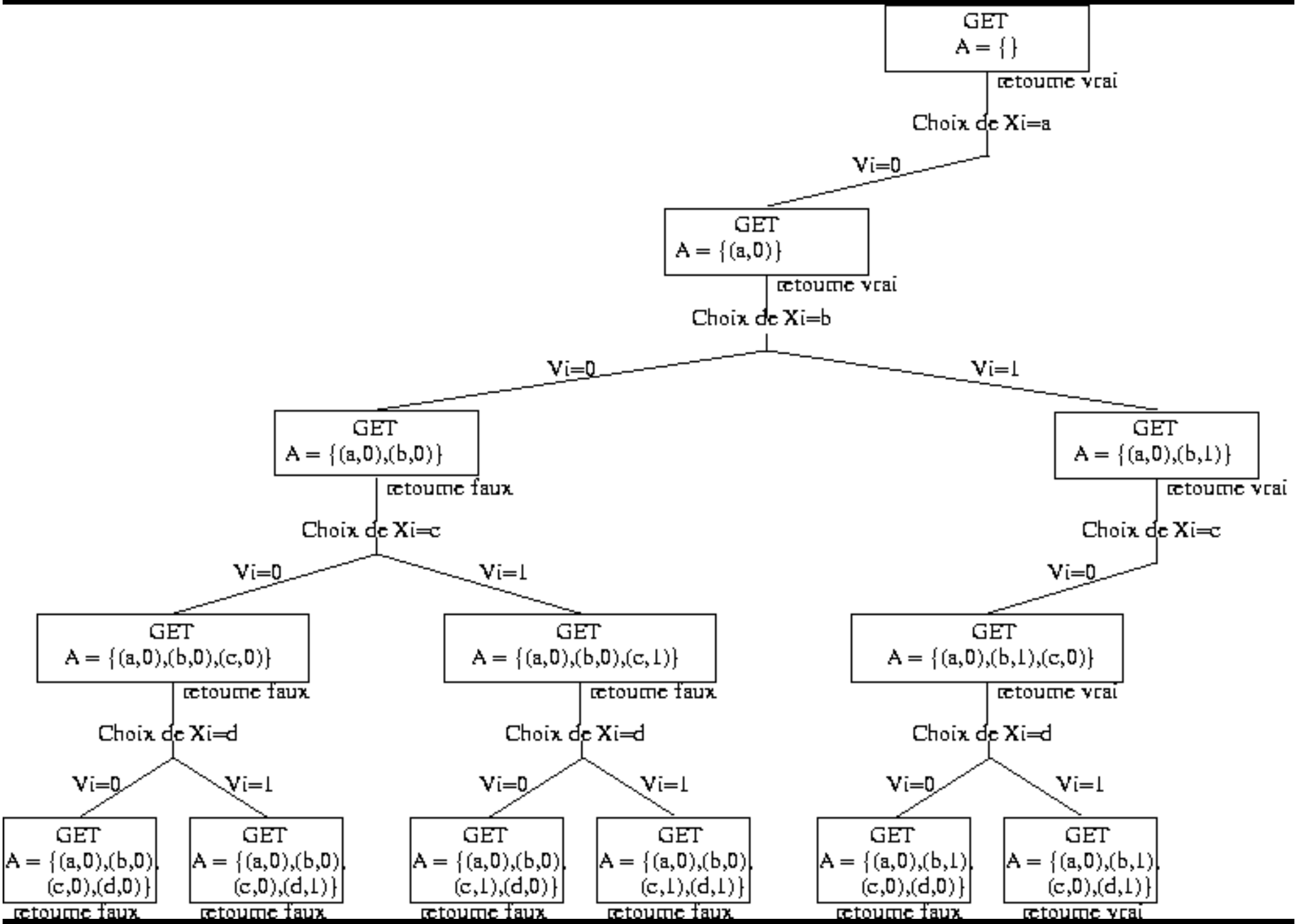
- : the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth *n* with *n* variables
3. b = (n - i)d at depth *i*, hence $n!*d^n$ leaves
   – b = branching factor d: number of element in the domain

Example

- $X = \{a,b,c,d\}$
- $D(a) = D(b) = D(c) = D(d) = \{0,1\}$
- $C = \{ a \neq b, c \neq d, a+c < b \}$

- A is the current solution (not consistent nor complete at the beginning)


GET
A = {}
retourne vrai
Choix de Xi=a
Vi=0
GET
A = {(a,0)}
retourne vrai
Choix de Xi=b
Vi=0
Vi=1
GET
A = {(a,0),(b,0)}
retourne faux
GET
A = {(a,0),(b,1)}
retourne vrai
Choix de Xi=c
Choix de Xi=c
Vi=0
Vi=1
Vi=0
GET
A = {(a,0),(b,0),(c,0)}
retourne faux
GET
A = {(a,0),(b,0),(c,1)}
retourne faux
GET
A = {(a,0),(b,1),(c,0)}
retourne vrai
Choix de Xi=d
Choix de Xi=d
Choix de Xi=d
Vi=0
Vi=1
Vi=0
Vi=1
Vi=0
Vi=1
GET
A = {(a,0),(b,0),(c,0),(d,0)}
retourne faux
GET
A = {(a,0),(b,0),(c,0),(d,1)}
retourne faux
GET
A = {(a,0),(b,0),(c,1),(d,0)}
retourne faux
GET
A = {(a,0),(b,0),(c,1),(d,1)}
retourne faux
GET
A = {(a,0),(b,1),(c,0),(d,0)}
retourne faux
GET
A = {(a,0),(b,1),(c,0),(d,1)}
retourne vrai

Pb with naive generate-and-test

- Search space is often too big to test everything in a reasonable time
  - (here product of the number of values of each variable domain)
- Only develop consistent partial Assignment
  - backtracking
- Reduce the domain values while searching
- Introduce heuristics

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

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

Ex: N-Queens first idea

We associate 2 variables Li (line) and Ci (column) to the queen i

- Variables :
- Domain :
- Constraints:

%queens must be on different lines

Clig = {Li ≠ Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}

%queens must be on different columns

Ccol = {Ci ≠ Cj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}

%queens must be on different diagonals (going up)

Cdu = {Ci+Li≠Cj+Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}

%queens must be on different diagonals (going down)

Cdd = {Ci-Li≠Cj-Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}

+ALLDiff({L1,L2,L3,L4}) and *Ccol = ALLDiff({C1,C2,C3,C4})*.

Ex: N queens second idea

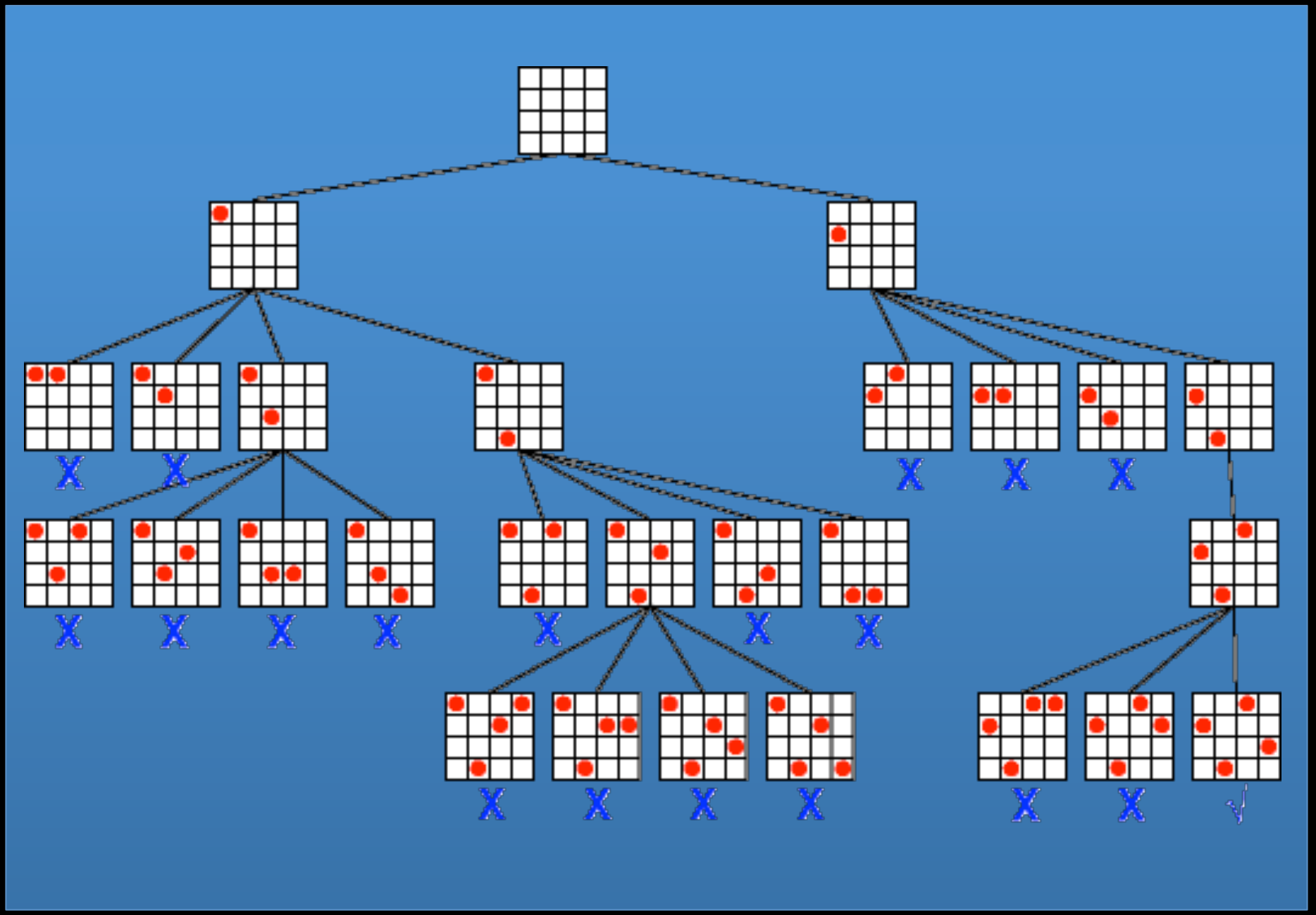
We suppose that we know that there is only one queen per column

- Variables : $X = \{X_1, X_2, X_3, X_4\}$
- Domain : $D(X_1) = D(X_2) = D(X_3) = D(X_4) = \{1,2,3,4\}$
- Constraints : $C =$

$\{X_i \neq X_j \,/\, i \text{ element\_of } \{1,2,3,4\}, j \text{ element\_of } \{1,2,3,4\} \text{ and } i \neq j\} \cup$

$\{X_i + i \neq X_j + j \,/\, i \text{ element\_of } \{1,2,3,4\}, j \text{ element\_of } \{1,2,3,4\} \text{ and } i \neq j\} \cup$

$\{X_i - i \neq X_j - j \,/\, i \text{ element\_of } \{1,2,3,4\}, j \text{ element\_of } \{1,2,3,4\} \text{ and } i \neq j\}$



Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

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

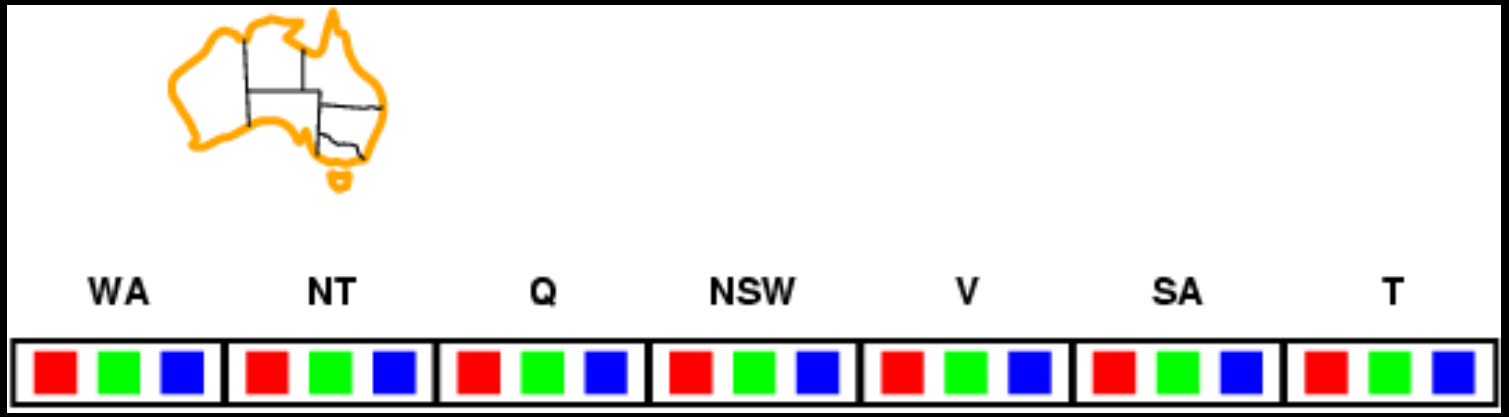
Least constraining value

- 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

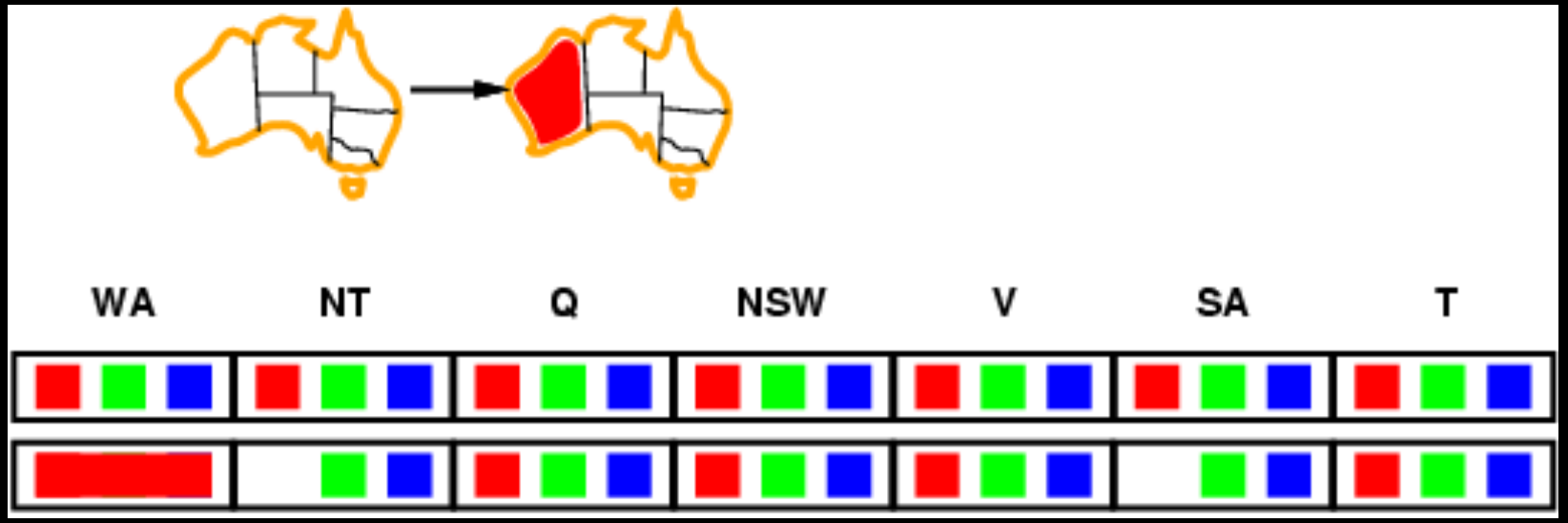
Forward checking

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



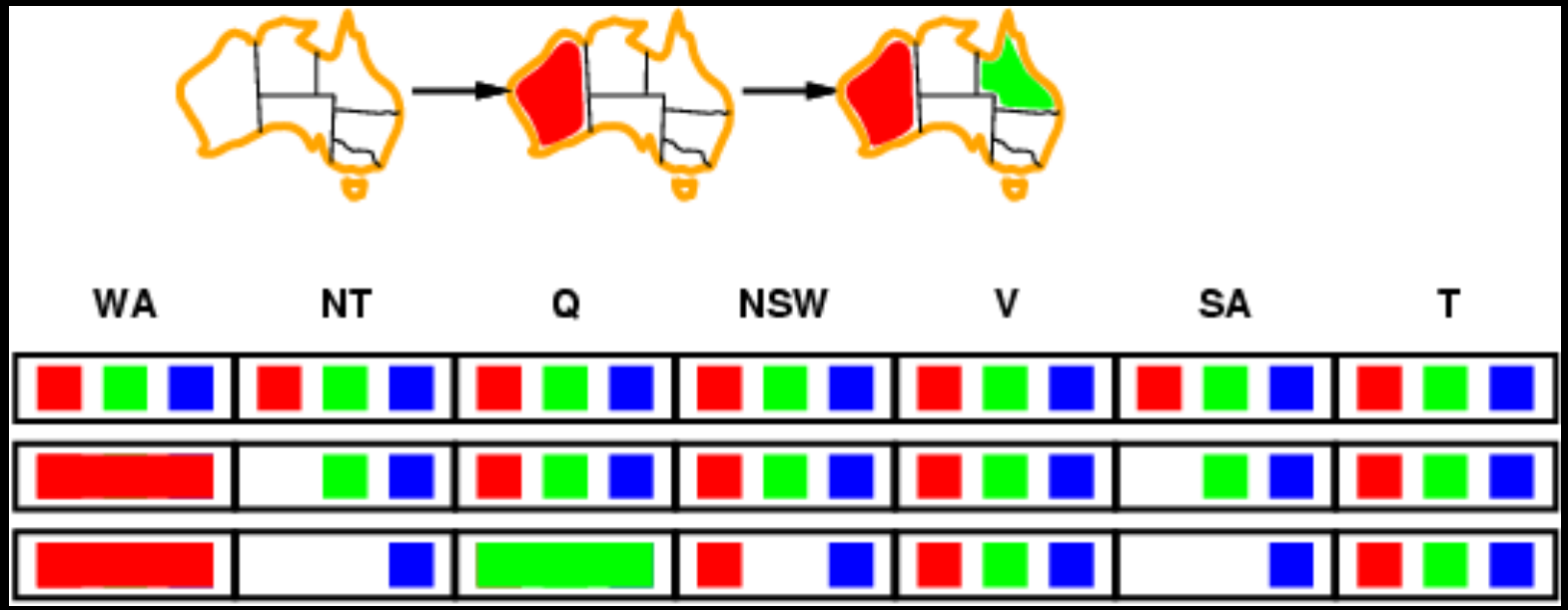
Forward checking

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



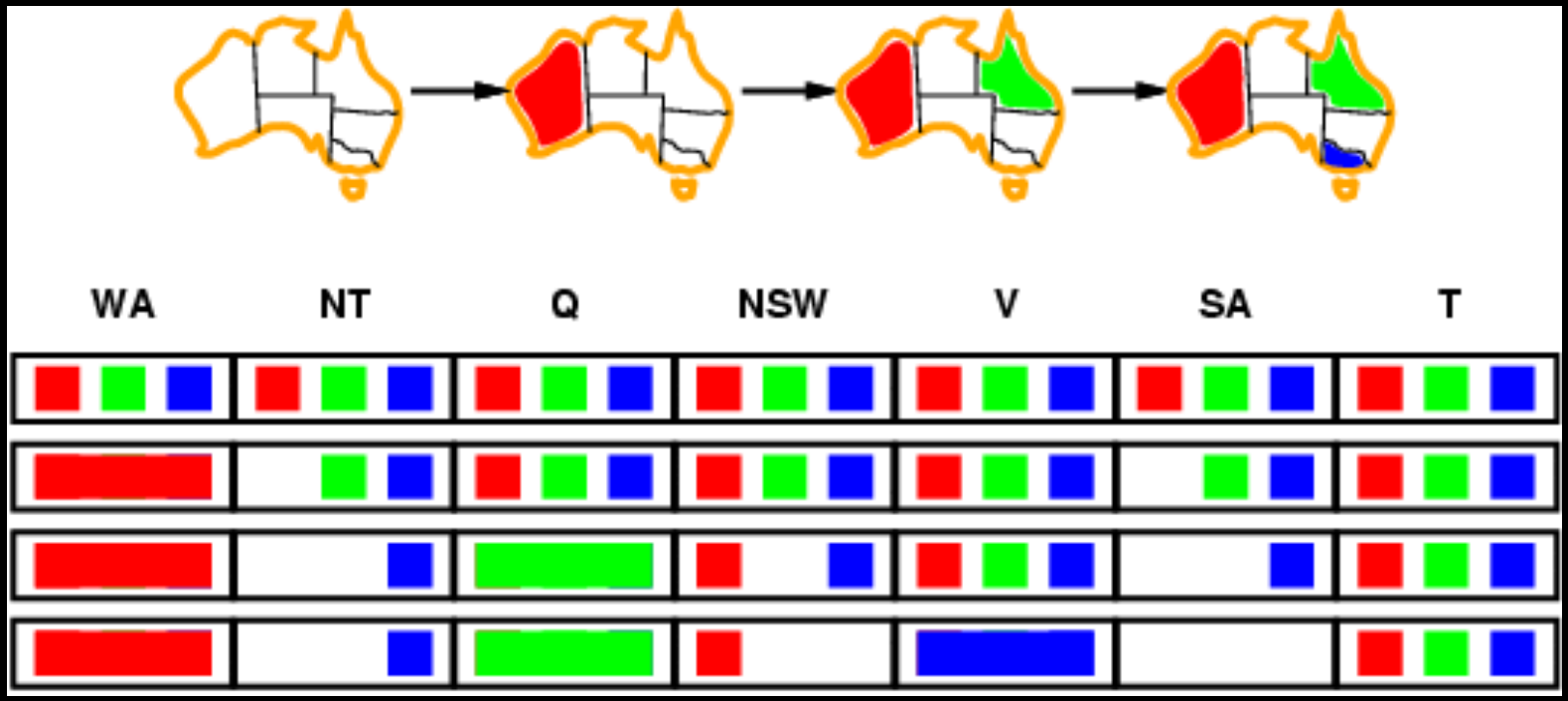
Forward checking

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



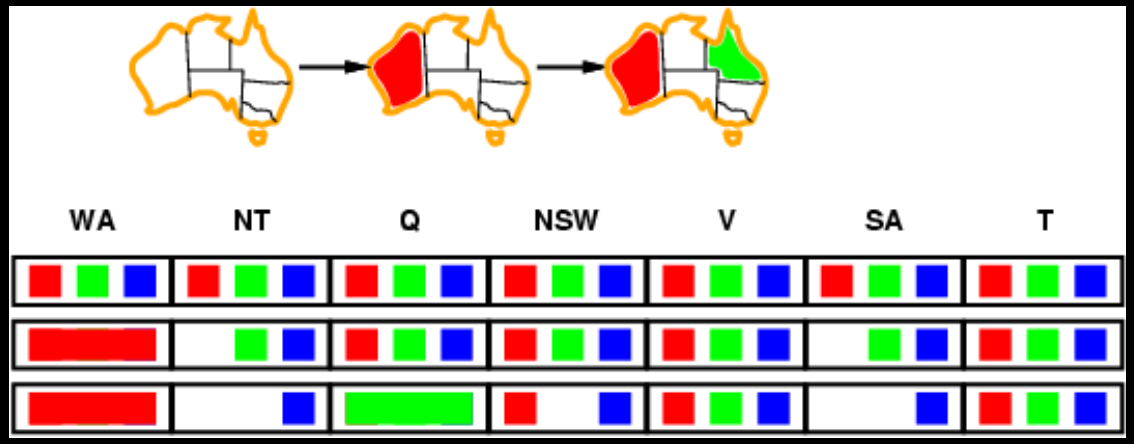
Forward checking

- :
  - 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:



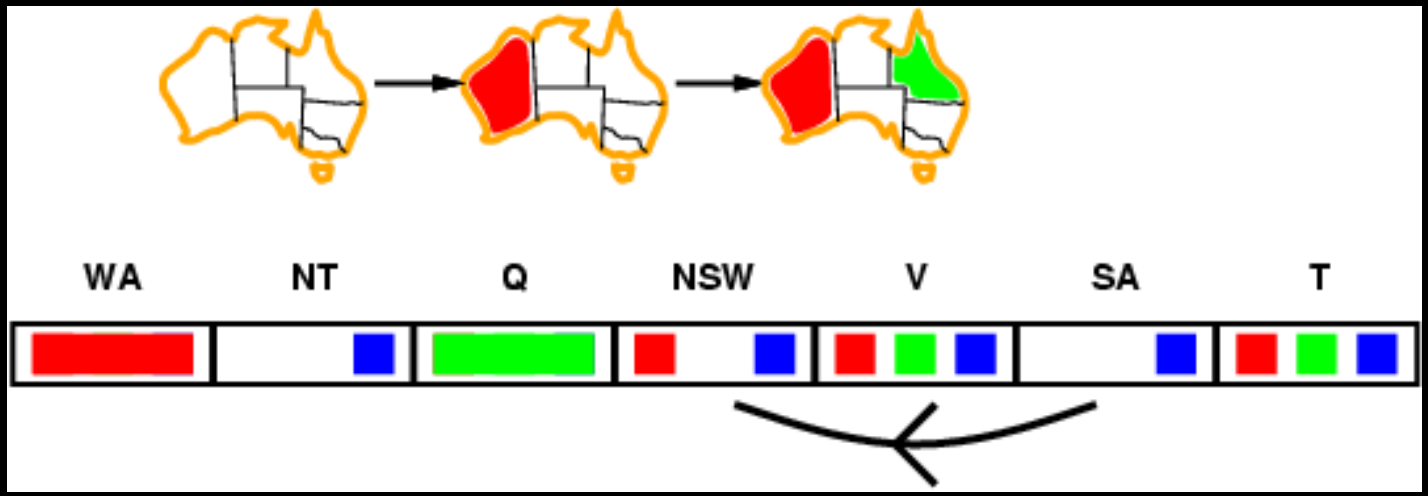
- NT and SA cannot both be blue!

- Constraint propagation repeatedly enforces constraints locally

Arc consistency

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

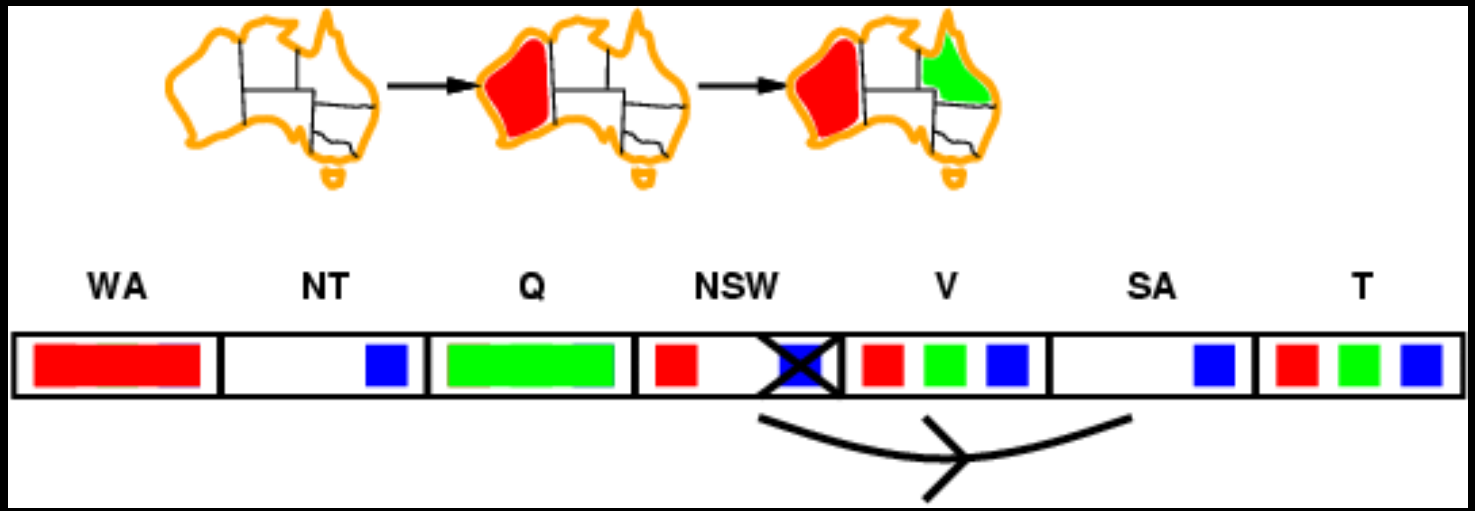
  for every value x of X there is some allowed y



Arc consistency

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

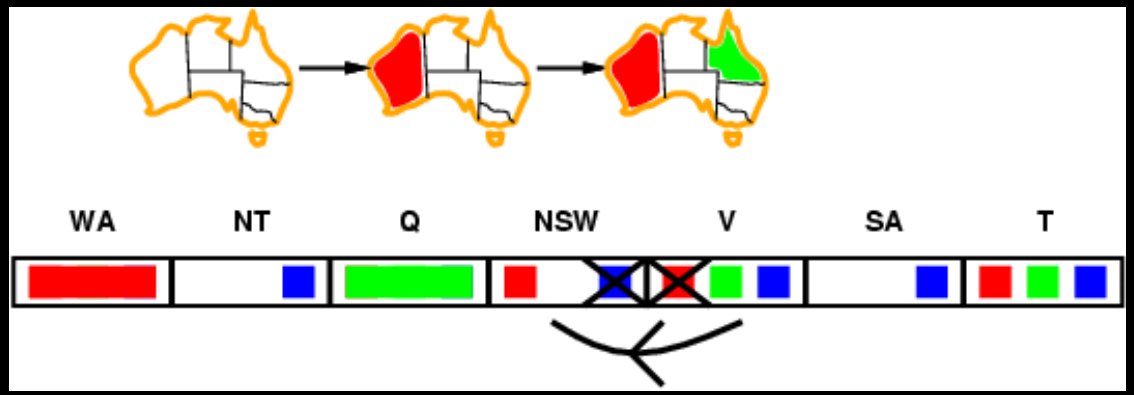
for every value *x* of *X* there is some allowed *y*



Arc consistency

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

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

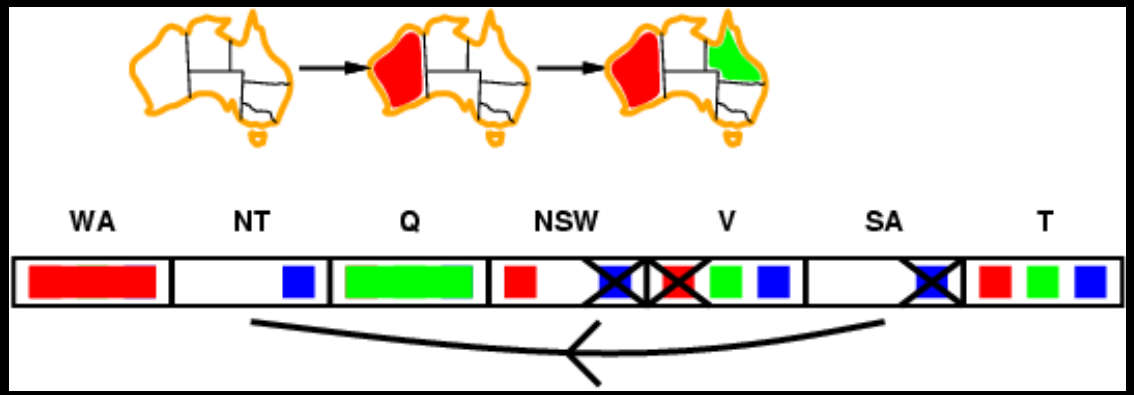


- If *X* loses a value, neighbors of *X* need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *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 AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., 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) ← REMOVE-FIRST(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

---

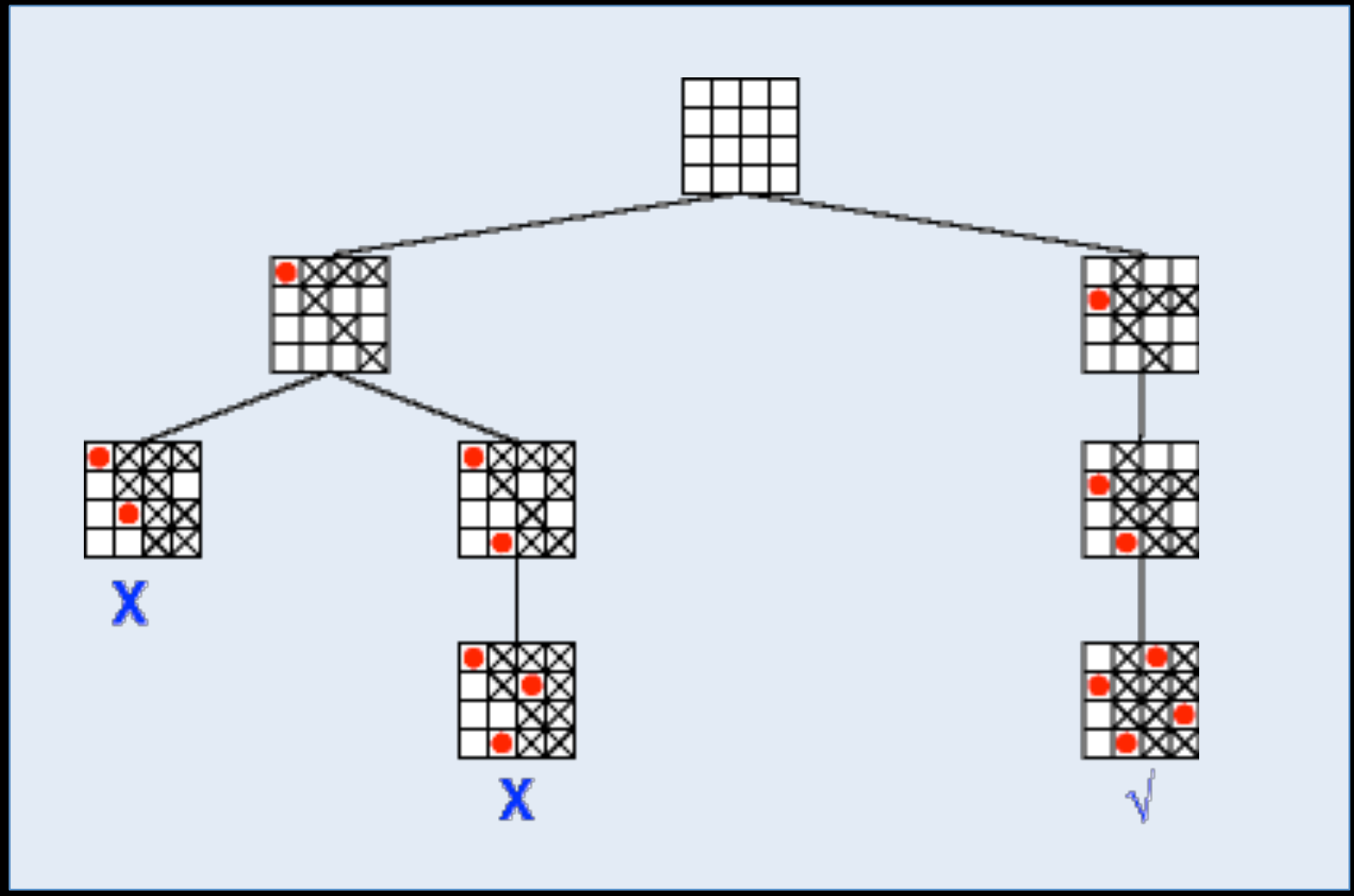
function RM-INCONSISTENT-VALUES( X_i, X_j) returns true iff remove a value
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint(X_i, X_j)
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

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

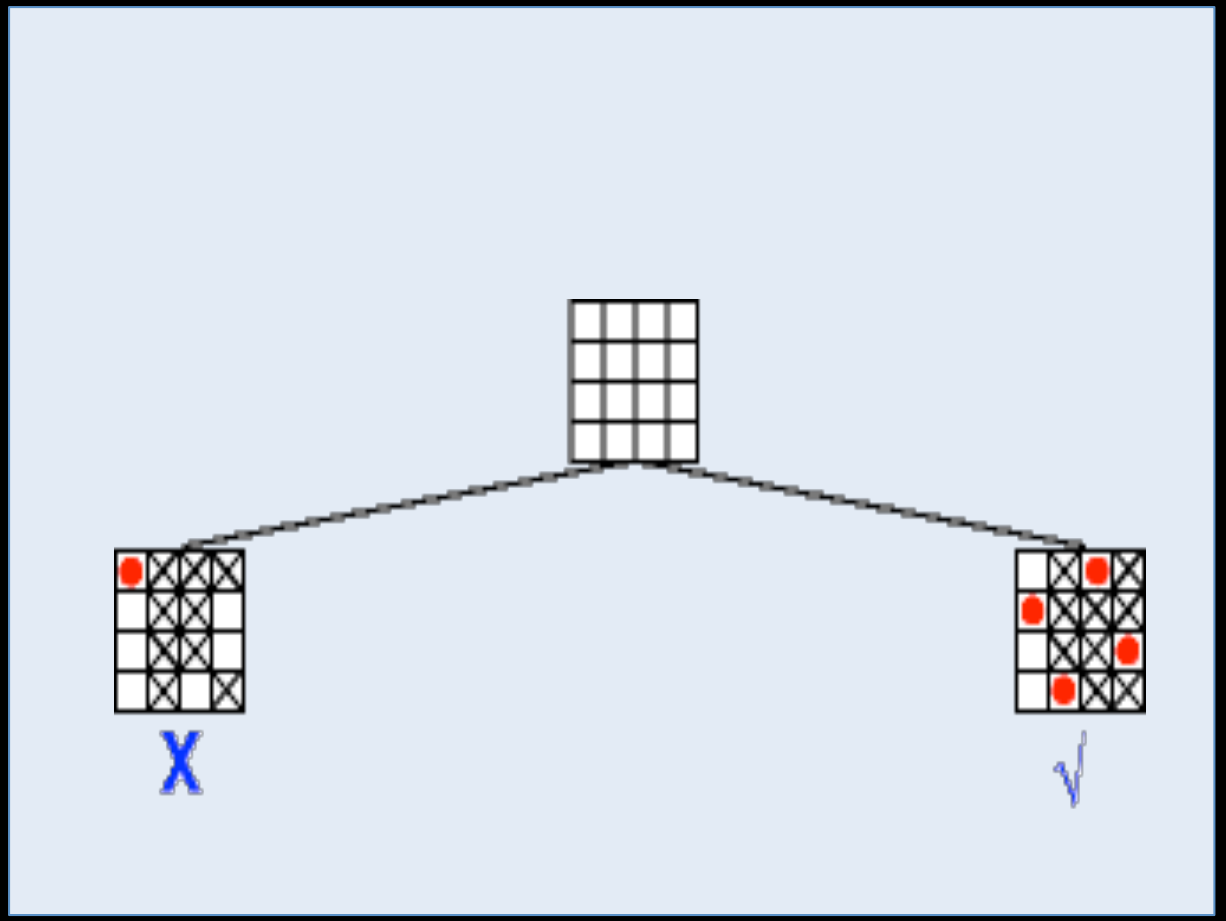
Node consistency ?

- Node consistency requires that every unary constraint on a variable is satisfied by all values in the domain of the variable, and vice versa.
  - can be enforced by reducing the domain of each variable to the values that satisfy all unary constraints on that variable. As a result, unary constraints can be neglected and assumed incorporated into the domains.
- Ex: given V a variable with a domain {1,2,3,4} and a constraint V =< 3, node consistency would restrict the domain to {1,2,3} and the constraint could then be discarded.

N queens only with node consistency



N queens with Arc consistency



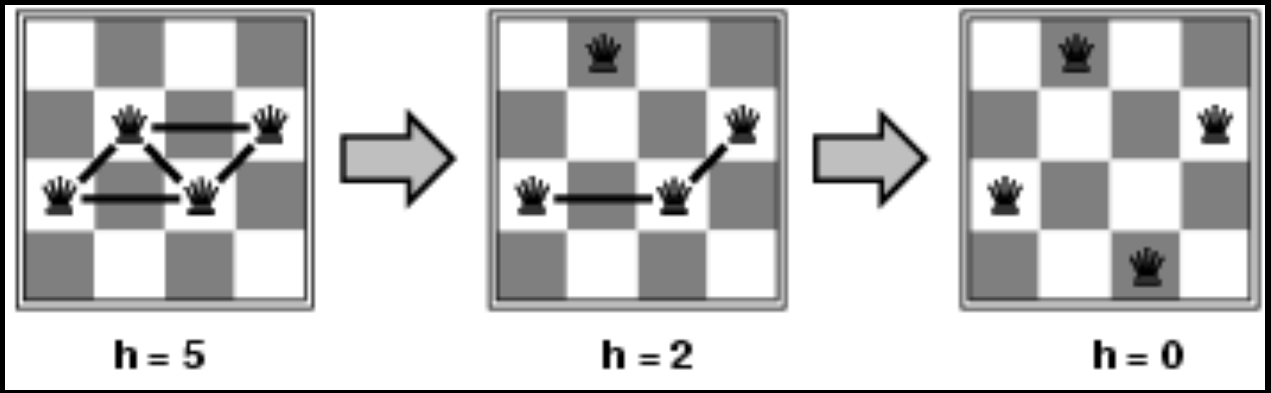
Local search for CSPs

- 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)

WRITING CONSTRAINT SOLVER IN PROLOG

Goal of this part

- See exactly how the solvers work
- Practise your PROLOG programming

- First only on binary CSP (constraints on only two variables)

Solver toolkit 1

To define the variables and their constraint, write the predicate Variables/1 which succeeds if the list argument has the following format:

```
?- variables(L).
L = [x(1):[1,2,3,4], x(2):[1,2,3,4], x(3):
[1,2,3,4], x(4):[1,2,3,4]]
yes
```

Solver toolkit 2

Write the predicate consistent/2 which describe the binary constraints (ex for 4-queens). It succeeds if the partial assignment is consistent (i.e if the binary relations bw two variables is true)

```
|?- consistent((x(1),2),(x(4),2)).
no

| ?- consistent((x(1),1),(x(2),3)).
yes
```

N-Queens CSP example

```
variables([x(1):[1,2,3,4], x(2):[1,2,3,4], x(3):
[1,2,3,4], x(4):[1,2,3,4]]).

consistent((x(i),Vi),(x(j),Vj)) :-
        Vi =\= Vj,
        i+Vi =\= j+Vj,
        i-Vi =\= j-Vj.
```

Generate-and-test 1

- Generate(V,A) succeeds if A is a total assignment of the variables and the domains defined in V.
  - Ex : 4-Queens

```
| ?- variables(V), generate(V,A).
A = [(x(1),1),(x(2),1),(x(3),1),(x(4),1)] ? ;
A = [(x(1),1),(x(2),1),(x(3),1),(x(4),2)] ? ;
A = [(x(1),1),(x(2),1),(x(3),1),(x(4),3)] ? ;
A = [(x(1),1),(x(2),1),(x(3),1),(x(4),4)] ? ;
A = [(x(1),1),(x(2),1),(x(3),2),(x(4),1)] ? ;
A = [(x(1),1),(x(2),1),(x(3),2),(x(4),2)] ? ;
A = [(x(1),1),(x(2),1),(x(3),2),(x(4),3)] ? ;
etc...
```

Generate-and-test 2

- *test(A)* succeeds if A is a consistent assignment

```
| ?- test([(x(1),1),(x(2),3),(x(3),2)]).
No
| ?- test([(x(1),2),(x(2),4),(x(3),1),(x(4),3)]).
Yes
```

- *generateAndTest(A)* unifies *A* with a solution of the CSP described by *variables/1* et *consistent/2*.

```
| ?- generateEttest(A).
A = [(x(1),2),(x(2),4),(x(3),1),(x(4),3)] ? ;
A = [(x(1),3),(x(2),1),(x(3),4),(x(4),2)] ? ;
(10 ms) no
```

Solution

```
generateANDTest(Assignment) :- variables(Variables),
generate(Variables,Assignment),
test(Assignment).

generate([],[]).
generate([X:DX|Variables],[(X,V)|Assignment]) :-
member(V,DX),
generate(Variables,Assignment).

test([]).
test([(Xi,Vi)|L]) :-
test((Xi,Vi),L),
test(L).

test(_,[]).
test((Xi,Vi),[(Xj,Vj)|L]) :-
consistent((Xi,Vi),(Xj,Vj)),
test((Xi,Vi),L).
```

Simple backtracking

- First rewrite test/1 into test*((X,V),A)* which succeed if the new assignment X to the value V is consistent with what was already assigned in A.
- Write the predicate simplebacktrack (similar to generate/1) which succeed if
  - V contains the list of not yet assigned variables with their corresponding domain.
  - *PartialA* contains the list of already assigned variables (*PartialA=[]* at first call) ;
  - *AV* unifies with an assignment of variables in V consistent with *PartialA.*

Solution

```
simpleBacktrack(Assignment) :-
variables(Variables),
simpleBacktrack(Variables,[],Assignment).

simpleBacktrack([],_,[]).
simpleBacktrack([X:DX|Variables],AssignmentPartial,
[(X,V)|Solution]) :-
member(V,DX),
test((X,V),AssignmentPartial),
simpleBacktrack(Variables,[(X,V)|
AssignmentPartial],Solution).

test(_,[]).
test(Ai,[Aj|L]) :-
consistent(Ai,Aj),
test(Ai,L).
```

Ex: simplebacktrack on N-Queens

- ```
  |?- simplebacktrack([x(3): [1,2,3,4], x(4):
  [1,2,3,4]], [(x(1),2),(x(2),4)],A).
  A = [(x(3),1),(x(4),3)] ? ;
  no

  | ?- simplebacktrack( [x(3): [1,2,3,4], x(4):
  [1,2,3,4]], [(x(1),1),(x(2),3)],A).
  no

  | ?- simplebacktrack([x(1): [1,2,3,4], x(2):
  [1,2,3,4], x(3): [1,2,3,4], x(4): [1,2,3,4]],
  [],A).
  A = [(x(1),2),(x(2),4),(x(3),1),(x(4),3)] ? ;
  A = [(x(1),3),(x(2),1),(x(3),4),(x(4),2)] ? ;
  no
  ```
  ```

Integration of Node-consistency 1/2

Add to the simplebacktrack predicate a filtering part.

filter(X, V, Variables, VariablesFiltered) succeeds if

- *X is a variable*
- *V* is the value assigned to *X,*
  - *Variables* contains the list of not yet assigned variables with their corresponding domain.
  - *VariablesFiltered unifies with the list of not yet assigned variables with their corresponding FILTERED domain (i.e, the value incompatible with the assignment of X to the value V are removed from the domain; each filtered domain must contain at least one value or the predicate fails).*

Example

```
| ?- filter(x(1),1,[x(2):[1,2,3,4], x(3):
[1,2,3,4], x(4):[1,2,3,4]],VarFiltered).
VarFiltered = [x(2):[3,4], x(3):[2,4], x(4):
[2,3]] ?
yes

| ?- filter(x(1),1,[x(2):[1,2], x(3):[1,2],
x(4):[1,2]], VarFiltered).
no
```

Integration of Node-consistency 2/2

- Using the predicate filter (and similarly to the predicate simplebacktracking) , write the predicate anticipate(*)* which succeeds if:
  - V contains the list of not yet assigned variables with their corresponding domain.
  - *PartialA* contains the list of already assigned variables (*PartialA=[]* at first call) ;
  - *CompleteA* unifies with an assignment of variables in V consistent with *PartialA.*
- Then write anticipate/1 such that *anticipate(A)* unifies *A* with a solution of the CSP described by the predicates *variables/1* et *consistent/2*.

Solution

```
anticipate(Assignment) :-
variables(Variables),
anticipate(Variables,[],Assignment).

anticipate([],_,[]).
anticipate([X:DX|Variables],AssignmentPartial,[(X,V)|
Solution]) :-
member(V,DX),
filter(X,V,Variables,Variablesfilters),
anticipate(Variablesfilters,[(X,V)|
AssignmentPartial],Solution).

filter(_,_,[],[]).
filter(Xi,Vi,[Xj:Dj|Variables],[Xj:Djfilter|
Domainesfilters]) :-
bagof(Vj, (member(Vj,Dj),consistants((Xi,Vi),(Xj,Vj))),
Djfilter),
Djfilter \= [],
filter(Xi,Vi,Variables,Domainesfilters).
```

Integration of First-fail heuristic

This heuristic allow the solver to select the variable with the smallest number of elements in its domain. If several variables have the same number of elements the leftmost variable is selected (in our case, we chose the variable with the smallest domain after filtering)

First, we need to write the predicate *extractMin(Variables,Xmin,VariableswoXmin)* which succeeds if *Variables is a list of* variables with their corresponding domain.

- *Xmin* unifies with the variable (and its domain) of the list with the smallest domain ;
- *VariableswoXmin* unifies with the list *Variables* without the variable *Xmin* (and its domain).

Example

- | ?- extractMin([a:[1,2,3,4], b:[1,2], c:[1,2,3]], X, LwithoutX).
  X = b:[1,2]
  LwithoutX = [a:[1,2,3,4],c:[1,2,3]]

First-fail heuristic

Using the predicate *extractMin/3*, and similarly to anticipate, write the predicate firstfail/3 which chose at each step the variable with the smallest domain then filter the domain of the not yest assigned variable to fulful node consistency.

- *V,PartialA, CompleteA)* which succeeds if:
  - V contains the list of not yet assigned variables with their corresponding domain.
  - *PartialA* contains the list of already assigned variables (*PartialA=[]* at first call) ;
  - *CompleteA* unifies with an assignment of variables in V consistent with *PartialA.*
- Then write firstfail/1 such that *anticipate(A)* unifies *A* with a solution of the CSP described by the predicates *variables/1* et *consistent/2*.

Solution

```
firstFail(Assignment) :-
variables(Variables),
firstFail(Variables,[],Assignment).

firstFail([],_,[]).
firstFail(Variables,AssignmentPartial,[(X,V)|Solution]) :-
extractMin(Variables,X:DX,Variables-X),
member(V,DX),
filter(X,V,Variables-X,Variablesfilters),
firstFail(Variablesfilters,[(X,V)|AssignmentPartial],Solution).

filter(_,_,[],[]).
filter(Xi,Vi,[Xj:Dj|Variables],[Xj:Djfilter|Domainesfilters]) :-
bagof(Vj,(member(Vj,Dj),consistants((Xi,Vi),(Xj,Vj))), Djfilter),
Djfilter \= [],
filter(Xi,Vi,Variables,Domainesfilters).

extractMin([X:DX|Variables],Xmin:DXmin,Variables-Xmin) :-
length(DX,N),
extractMin(Variables,X:DX,N,Xmin:DXmin,Variables-Xmin).

extractMin([],X:DX,_,X:DX,[]).
extractMin([X:DX|Variables],Y:DY,N,Z:DZ,[Y:DY|Variables-Z]) :-
length(DX,M),
M<N, !,
extractMin(Variables,X:DX,M,Z:DZ,Variables-Z).
extractMin([X:DX|Variables],Y:DY,N,Z:DZ,[X:DX|Variables-Z]) :-
extractMin(Variables,Y:DY,N,Z:DZ,Variables-Z).
```

Comparison

Nb of queens	”generate and test "	"simple backtracking"	“ node consistency"	“node consistency + firstfail heuristic ”
6	730 ms	10 ms	10 ms	10 ms
8	288 530 ms	210 ms	140 ms	130 ms
10	-	5 880 ms	2 520 ms	2 310 ms
12	-	183 110 ms	62 820 ms	52 240 ms

http://gprolog.univ-paris1.fr/manual/html_node/gprolog055.html

CLP(FD) IN GNU PROLOG

Example: N Queens

- reines4_modele2(Vars,Options) :-
  /* Declare variables : simple PROLOG list*/
  Vars = [X1,X2,X3,X4],
  /* The domain of each variable is {1,2,3,4} */
  fd_domain(Vars,1,4),
  /* Queens must stand on different lines*/
  fd_all_different(Vars),
  /* Queens must not be on the same up diagonal*/
  1+X1 #\= 2+X2, 1+X1 #\= 3+X3, 1+X1 #\= 4+X4,
  2+X2 #\= 3+X3, 2+X2 #\= 4+X4, 3+X3 #\= 4+X4,
  /* Queens must not be on the same down diagonal*/
  1-X1 #\= 2-X2, 1-X1 #\= 3-X3, 1-X1 #\= 4-X4,
  2-X2 #\= 3-X3, 2-X2 #\= 4-X4, 3-X3 #\= 4-X4,
  /* find an assignment which is a solution*/
  fd_labeling(Vars,Options).

Finite Domain variables 1/2

A new type of data is introduced: FD variables which can only take values in their domains.

- The initial (default) domain of an FD variable is 0..fd_max_integer where fd_max_integer represents the greatest value that any FD variable can take.
  - You can ask what fd_max_integer is using the predicate *fd_max_integer/1*.
- There are two internal representations for an FD variable:
  - interval representation: only the *min* and the *max* of the variable are maintained. In this representation it is possible to store values included in 0..fd_max_integer.
  - sparse representation: an additional bit-vector is used to store the set of possible values for the variable (i.e. the domain). In this representation it is possible to store values included in 0..vector_max. By default vector_max is set to 127.

Initial value constraint

- To explicitely declare a FD variable (ex when the domain is different from *0..fd_max_integer*) you can use:

```
fd_domain(+fd_variable_list_or_fd_variable, +integer, +integer)
fd_domain(?fd_variable, +integer, +integer)
fd_domain_bool(+fd_variable_list)
fd_domain_bool(?fd_variable)
```

- fd_domain(Vars, Lower, Upper) constraints each element X of Vars to take a value in Lower..Upper. (ex to set the initial domain of variables to an interval). Vars can be also a single FD variable (or a single Prolog variable).
- fd_domain_bool(Vars) is equivalent to fd_domain(Vars, 0, 1) and is used to declare boolean FD variables.

FD variable information

- These predicate allow the user to get some information about FD variables. They are not constraints, they only return the current state of a variable.

fd_min(+fd_variable, ?integer)
fd_max(+fd_variable, ?integer)
fd_size(+fd_variable, ?integer)
fd_dom(+fd_variable, ?integer_list)

- fd_min(X, N) succeeds if N is the minimal value of the current domain of X.
- fd_max(X, N) succeeds if N is the maximal value of the current domain of X.
- fd_size(X, N) succeeds if N is the number of elements of the current domain of X.
- fd_dom(X, Values) succeeds if Values is the list of values of the current domain of X.

Arithmetic expressions

FD variable X	domain of X
integer number N	domain N..N
+ E	same as E
- E	opposite of E
E1 + E2	sum of E1 and E2
E1 - E2	subtraction of E2 from E1
E1 * E2	multiplication of E1 by E2
E1 / E2	integer division of E1 by E2 (only succeeds if the remainder is 0)
E1 ** E2	E1 raised to the power of E2 (E1 or E2 must be an integer)
min(E1,E2)	minimum of E1 and E2
max(E1,E2)	maximum of E1 and E2
dist(E1,E2)	distance, i.e. |E1 - E2|
E1 // E2	quotient of the integer division of E1 by E2
E1 rem E2	remainder of the integer division of E1 by E2

Arithmetic constraints (for partial AC)

Templates	Description
#=(?fd_evaluable, ?fd_evaluable)	FdExpr1 #= FdExpr2 constrains FdExpr1 to be equal to FdExpr2.
#\=(?fd_evaluable, ?fd_evaluable)	FdExpr1 #\= FdExpr2 constrains FdExpr1 to be different from FdExpr2.
#<(?fd_evaluable, ?fd_evaluable)	FdExpr1 #< FdExpr2 constrains FdExpr1 to be less than FdExpr2.
#=<(?fd_evaluable, ?fd_evaluable)	FdExpr1 #=< FdExpr2 constrains FdExpr1 to be less than or equal to FdExpr2.
#>(?fd_evaluable, ?fd_evaluable)	FdExpr1 #> FdExpr2 constrains FdExpr1 to be greater than FdExpr2.
#>=(?fd_evaluable, ?fd_evaluable)	FdExpr1 #>= FdExpr2 constrains FdExpr1 to be greater than or equal to FdExpr2.

FdExpr1 and FdExpr2 are arithmetic FD expressions

Boolean constraints

FD Expression	Result
Prolog variable	domain `0..1`
FD variable x	domain of x, x is constrained to be in `0..1`
`0` (integer)	`0` (false)
`1` (integer)	`1` (true)
`#\ E`	not `E`
`E1 #<=> E2`	`E1` equivalent to `E2`
`E1 #\<=> E2`	`E1` not equivalent to `E2` (i.e. `E1` different from `E2`)
`E1 ## E2`	`E1` exclusive OR `E2` (i.e. `E1` not equivalent to `E2`)
`E1 #==> E2`	`E1` implies `E2`
`E1 #\==> E2`	`E1` does not imply `E2`
`E1 #/\ E2`	`E1` AND `E2`
`E1 #\/\ E2`	`E1` NAND `E2`
`E1 #\/ E2`	`E1` OR `E2`
`E1 #\\/ E2`	`E1` NOR `E2`

Reified constraints?

- Instead of merely posting constraints it is often useful to reflect its truth value into a 0/1-variable *B*, so that:
  - the constraint is posted if *B* is set to 1
  - the negation of the constraint is posted if *B* is set to 0
  - *B* is set to 1 if the constraint becomes entailed
  - *B* is set to 0 if the constraint becomes disentailed
- This mechanism is known as *reification*. Several frequently used operations can be defined in terms of reified constraints. A reified constraint is written:
- | ?- *Constraint* #<=> *B*. where *Constraint* is reifiable

Example reification

- As an example of a constraint that uses reification, consider *exactly(X,L,N)*, defined to be true if *X* occurs exactly *N* times in the list *L*. It can be defined thus:

```
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
        X #= Y #<=> B,
        N #= M+B,
        exactly(X, L, M).
```

- fd_reified_in(X, Lower, Upper, B) captures the truth value of the constraint X ∈ [Lower..Upper] in the boolean variable B.

Constraints on constraints

fd_cardinality(+fd_bool_evaluable_list, ?fd_variable)
fd_cardinality(+integer, ?fd_variable, +integer)
fd_at_least_one(+fd_bool_evaluable_list)
fd_at_most_one(+fd_bool_evaluable_list)
fd_only_one(+fd_bool_evaluable_list)

- fd_cardinality(List, Count) unifies Count with the number of constraints that are true in List. This is equivalent to post the constraint $B_1 + B_2 + \ldots + B_n$ #= Count where each variable Bi is a new variable defined by the constraint B_i #<=> C_i where C_i is the i*th* constraint of List. Each C_i must be a boolean FD expression (section 9.7.1).
- fd_cardinality(Lower, List, Upper) is equivalent to fd_cardinality(List, Count), Lower #=< Count, Count #=< Upper
- fd_at_least_one(List) is equivalent to fd_cardinality(List, Count), Count #>= 1.
- fd_at_most_one(List) is equivalent to fd_cardinality(List, Count), Count #=< 1.
- fd_only_one(List) is equivalent to fd_cardinality(List, 1).

Symbolic constraints 1/2

fd_all_different(+fd_variable_list)

fd_all_different(List) constrains all variables in List to take distinct values. This is equivalent to posting an inequality constraint for each pair of variables. This constraint is triggered when a variable becomes ground, removing its value from the domain of the other variables.

fd_element(?fd_variable, +integer_list, ?fd_variable)

fd_element(I, List, X) constraints X to be equal to the I*th* integer of List.

fd_element_var(?fd_variable, +fd_variable_list, ?fd_variable)

fd_element_var(I, List, X) constraints X to be equal to the I*th* variable of List. This constraint is similar to fd_element/3 but List can also contain FD variables (rather than just integers).

Symbolic constraints 2/2

fd_relation(+integer_list_list, ?fd_variable_list)
fd_relationc(+integer_list_list, ?fd_variable_list)

- fd_relation(Relation, Vars) constraints the tuple of variables Vars to be equal to one tuple of the list Relation. A tuple is represented by a list.

Ex: definition of the boolean AND relation so that X AND Y ⇔ Z:

and(X,Y,Z):- fd_relation([[0,0,0],[0,1,0],[1,0,0],[1,1,1]], [X,Y,Z]).

- fd_relationc(Columns, Vars) is similar to fd_relation/2 except that the relation is not given as the list of tuples but as the list of the columns of the relation. A column is represented by a list.

Ex: and(X,Y,Z):- fd_relationc([[0,0,1,1],[0,1,0,1],[0,0,0,1]], [X,Y,Z]).

Solving a CSP

- *fd_labeling(Vars, Options)* assigns a value to each variable X of the list Vars according to the list of labeling options given by Options. Vars can be also a single FD variable. This predicate is re-executable on backtracking.

- *fd_labeling(Vars)* is equivalent to fd_labeling(Vars, []).
- *fd_labelingff(Vars)* is equivalent to fd_labeling(Vars, [variable_method(ff)]).

Labeling options (1/2)

- Variable_method(V): specifies the heuristics to select the variable to enumerate:
  - standard: no heuristics, the leftmost variable is selected.
  - **first_fail (or ff): selects the variable with the smallest number of elements in its domain. If several variables have the same number of elements the leftmost variable is selected.**
  - most_constrained: like first_fail but when several variables have the same number of elements selects the variable that appears in most constraints.
  - smallest: selects the variable that has the smallest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - largest: selects the variable that has the greatest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - max_regret: selects the variable that has the greatest difference between the smallest value and the next value of its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - random: selects randomly a variable. Each variable is chosen only once.
- The default value is standard.

Labeling options (2/2)

- reorder(true/false): specifies if the variable heuristics should dynamically reorder the list of variable (true) or not (false). Dynamic reordering is generally more efficient but in some cases a static ordering is faster. The default value is true.
- value_method(V): specifies the heuristics to select the value to assign to the chosen variable:
  - min: enumerates the values from the smallest to the greatest **(default**).
  - max: enumerates the values from the greatest to the smallest.
  - middle: enumerates the values from the middle to the bounds.
  - bounds: enumerates the values from the bounds to the middle.
  - random: enumerates the values randomly. Each value is tried only once.
  - bisect: recursively creates a choice between X #=< M and X #> M, where M is the midpoint of the domain of the variable. Values are thus tried from the smallest to the greatest. This is also known as domain splitting.

- backtracks(B): unifies B with the number of backtracks during the enumeration.

N-Queen labeling examples

- To use the « first-fail » heuristic: *fd_labeling([X1,X2,X3,X4], [variable_method(first_fail)])*
- If we also want the values to be testd in increasing order (instead of diceasing order):

fd_labeling([X1,X2,X3,X4],[variable_method(first_fail), value_method(max)])

- To know how many backtracks were made: *fd_labeling([X1,X2,X3,X4],[variable_method(first_fail), value_method(max),backtracks(B)])*

then print B

Optimization constraints

fd_minimize(+callable_term, ?fd_variable)
fd_maximize(+callable_term, ?fd_variable)

- fd_minimize(Goal, X) repeatedly calls Goal to find a value that minimizes the variable X. Goal is a Prolog goal that should instantiate X, a common case being the use of fd_labeling/2. This predicate uses a branch-and-bound algorithm with restart: each time call(Goal) succeeds the computation restarts with a new constraint X #< V where V is the value of X at the end of the last call of Goal. When a failure occurs (either because there are no remaining choice-points for Goal or because the added constraint is inconsistent with the rest of the store) the last solution is recomputed since it is optimal.

- fd_maximize(Goal, X) is similar to fd_minimize/2 but X is maximized.