Advanced Programming Languages for AI
Constraint Logic Programming

Gerda Janssens
Departement computerwetenschappen
A01.26

Constraint (Logic) Programming

1. Top-down search with passive constraints (Prolog)
2. Delaying automatically (arithmetic constraints) using the suspend library
3. Constraint propagation in ECLiPSe
   the symbolic domain library (`sd`)
   the interval constraints library (`ic`)
4. Top-down search witch active constraints, also variable and value ordering heuristics
5. Optimisation with active constraints
6. Constraints on reals (`locate` library)
7. Linear constraints over continuous and integer variables (`eplex` library)

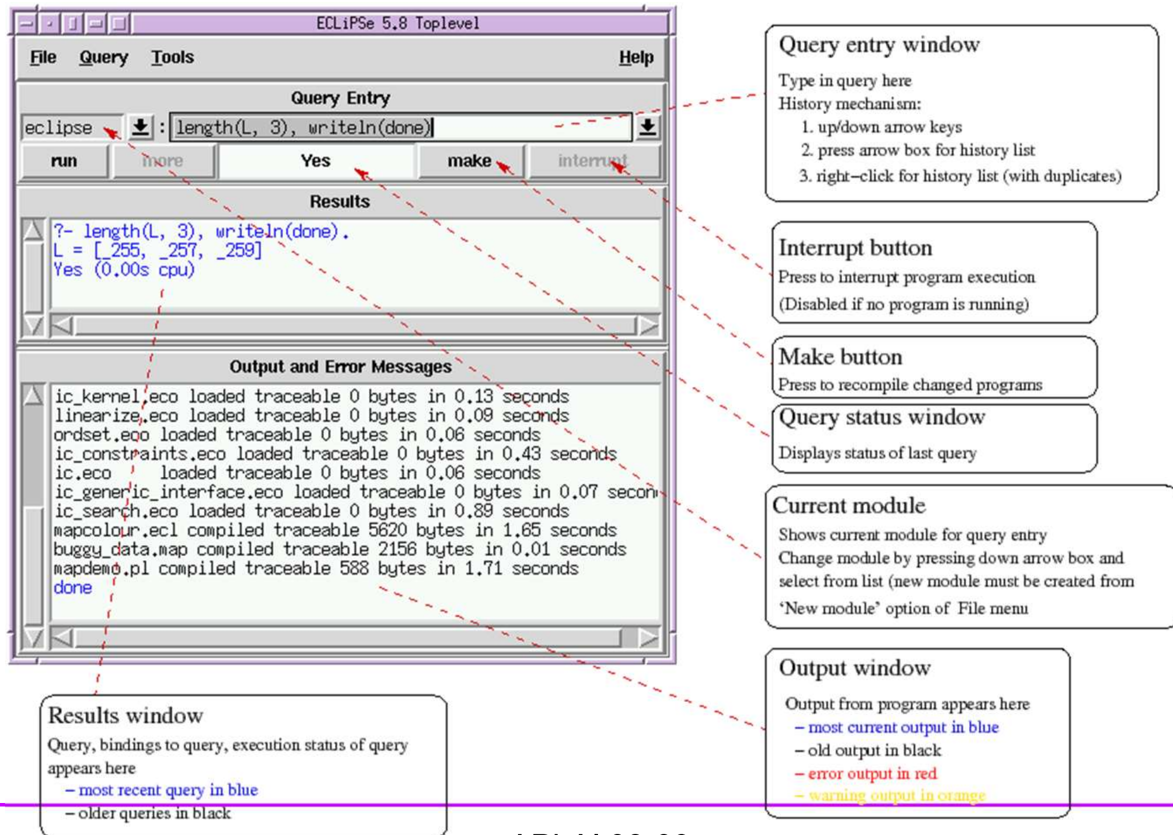
1. TOP-DOWN SEACRH WITH PASSIVE CONSTRAINTS

1. Top-down search with passive constraints

- Solving finite CSPs using Prolog (ECLiPSe)
- Backtracking search in Prolog
- Incomplete search: lds search
- Counting the number of backtracks (to measure efficiency)
- Prolog implies: constraints are passive and can only be used as tests

Tools

TkEclipse



Tools
Tracer and Data Inspector

Arithmetic constraints: passive

- Compare f(a,X) = f(Y,b) with 3*X < Y +2

Both put restrictions on the variables

But arithmetic constraint can only be processed when all the variables are ground

High level program for solving CSPs

```
solve(List) :-
    declareDomain(List),          %info about domains
    search(List),                 %launch search process
    testConstraints(List).
```

Generate and Test approach: INEFFICIENT

Example: SEND+ MORE = MONEY

number of decision variables: 8

number of leaves in the search tree: 10^8

(Better approach: interleave …)

Backtracking search in Prolog

- labelling as the branching method
- degrees of freedom:
  - order in which variables are labeled
  - which values are selected in the variable domains

The variable ordering

- variables X and Y;
  X has 2 possible values and Y has 4
- number of leaves in the search tree?
- number of internal nodes?
- to keep the number of internal nodes low:
  - label the variables with fewer choices earlier

```
search(X,Y) :- member(X,[1,2]),
      member(Y,[1,2,3,4]),
      X + Y =:= 6 .          % passive constraint
```

The value ordering

- Is the size of the search tree affected by different value orderings?
- No, as all values have to be explored.
- (Except in the case of incomplete search)

extra: iteration and recursion in ECLiPSe

- how do you write a predicate to write all elements of a given list on separate lines??
- iteration over the elements of a list:

```
[eclipse 1]: (foreach(El, [a,b,c]) do writeln(El)).
```

```
foreach(El,List) do Query(El)
```

Iterate `Query(El)` over each element `El` of the list `List`

extra: The iterator fromto in ECLiPSe

`fromto(First,In,Out,Rest) do Query(In,Out)`

Iterate `Query(In,Out)` starting with `In = First`, until `Out = Rest`

```
[eclipse 2]: (fromto([a,b,c], [H|Tail], Tail, [])
                do
                   writeln(H)
                ).
a       % [a,b,c] = [H |Tail]    and  Tail is threaded
b       % [b,c]                        [c]
c       % [c]                          []
% replaces recursion
% User Manual: Ch 5 ECLiPSe specific language features
```

Combining iterators: synchronous iteration

```
[eclipse 3]: (fromto([a,b,c], [H|Tail], Tail, []),
             foreach(El,List)
              do
                 El = H
              ).
[eclipse 4]: (fromto([], Tail, [H|Tail], [a,b,c]),
             foreach(El,List)
              do
                 El = H
              ).
```

extra: iterators

- write ordered(List) with fromto
- write reverse/2 with fromto and foreach

```
ordered(List) :-
    ( fromto(List,[El|Rest],Rest,[])
    do
      ordered2(El, Rest)
    ).

ordered2(_, []).
ordered2(X, [Y|_]) :- X =< Y.
```

Variable and value orderings in Prolog

```
% assign values from the variable domains to all the
  Var-Domain pairs in List
search(List) :-
  ( fromto(List, Vars, Rest, [])
  do
      choose_var(Vars, Var-Domain, Rest),
      choose_val(Domain, Val),
      Var = Val
  ).
choose_var(List, Var, Rest) :- List = [Var|Rest].
choose_val(Domain,Val) :- member(Val, Domain).
```

Incomplete search

- Suppose that the 'better' values appear earlier in the domains of the variables (by the use of some heuristic)
- Explore only the N best/first values
  `choose_val(N, Domain, BestList)`
- Results in greedy search

- Credit based search: give preference to values which seem more promising

```
search(List, Credit) :-
   ( fromto(List, Vars, Rest, []),
     fromto(Credit, CurCredit, NewCredit, _)
   do  choose_var(Vars, Var-Domain, Rest),
       choose_val(Domain, Val, CurCredit, NewCredit),
       Var = Val
   ).
choose_val(Domain, Val, CurCredit, NewCredit) :-
   share_credit(Domain, CurCredit, DomCredList),
   member(Val-NewCredit, DomCredList).
% share_credit(Domain, N, DomCredList) admits
% only the first N values.
share_credit(Domain, N, DomCredList) :-
   ( fromto(N, CurCredit, NewCredit, 0),
     fromto(Domain, [Val|Tail], Tail, _),
     foreach(Val-N, DomCredList),
     param(N)   % normally: to pass N into body of iterator
     % here: to thread the initial value of N into the loop
   do ( Tail = [] -> NewCredit is 0 ;
        NewCredit is CurCredit - 1 )
   ).
```

Credit based search

```
?- share_credit([1,2,3,4,5,6,7,8,9],5, Dlist).
Dlist = [1 - 5, 2 - 5, 3 - 5, 4 - 5, 5 - 5]
?- share_credit([1,2,3],5, Dlist).
Dlist = [1 - 5, 2 - 5, 3 - 5]
?-


% 5*4*4 solutions
```

- How to allocate half the credit to the first value of the domain, half of the remaining value to the second value, and so on. When only 1 credit is left, the next value is selected and is the last.

Credit based search: binary chop

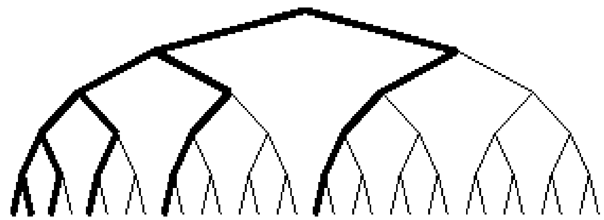
```
% share_credit(Domain, N, DomCredList)
% Allocate credit N by binary chop
share_credit(Domain, N, DomCredList) :-
( fromto(N, CurCredit, NewCredit, 0),
  fromto(Domain, [Val|Tail], Tail, _),
  foreach(Val-Credit, DomCredList)
  do ( Tail = [] -> Credit is CurCredit
     ;
      Credit is fix(ceiling(CurCredit/2))
            % smallest integer >= CurCredit/2
     ),
     NewCredit is CurCredit - Credit
).
```

Examples: binary chop

```
?- share_credit([1,2,3,4,5,6,7,8,9],5, Dlist).
Dlist = [1 - 3, 2 - 1, 3 - 1]

?- share_credit([1, 2, 3, 4, 5, 6, 7, 8, 9], 1000,
   Dlist).
Dlist = [1 - 500, 2 - 250, 3 - 125, 4 - 63, 5 - 31, 6 -
   16, 7 - 8, 8 - 4, 9 - 3]

?- search([X-[1,2,3,4,5,6,7,8,9], Y-[1,2,3,4],Z-
   [1,2,3,4]],5).
% only 5 solutions: 1 1 1 ; 1 1 2; 1 2 1; 2 1 1 ; 3 1 1

?- search([X-[1,2,3], Y-[1,2,3],Z-[1,2,3]],8).
% 1 1 1 ; 1 1 2; 1 2 1; 1 3 1 ; 2 1 1 ; 2 2 1 ; 3 1 1 ;
   3 2 1
```

Tree Search : incomplete strategies: lds(1)

Limited Discrepancy Search:



lds(1)

Credit based search: limited discrepancy credit allocation

- credit as a measure of distance from the preferred left-hand branch of the search tree

```
% allocate credit N by discrepancy
share_credit(Domain, N, DomCredList) :-
  ( fromto(N, CurCredit, NewCredit, 0),
    fromto(Domain, [Val|Tail], Tail, _),
    foreach(Val-CurCredit, DomCredList),
do ( Tail = [] -> NewCredit is 0 ;
      NewCredit is CurCredit - 1 )
  ).
```

Examples lds search

```
?- share_credit([1, 2, 3, 4, 5, 6, 7, 8, 9], 5,
   Dlist).
Dlist = [1 - 5, 2 - 4, 3 - 3, 4 - 2, 5 - 1]

?- share_credit([1, 2, 3], 5, Dlist).
Dlist = [1 - 5, 2 - 4, 3 - 3]

?- search([X-[1,2], Y-[1,2],Z-[1,2], U-[1,2], V-
   [1,2]],2).
% 6 solutions 1 1 1 1 1; 1 1 1 1 2; 1 1 1 2 1;
1 1 2 1 1 ; 1 2 1 1 1 ; 2 1 1 1 1
```

Getting an idea of the amount of search

- by counting the number of backtracks
- you need some system predicates like …

```
[eclipse 3]: N is 3, setval(count,N), incval(count),
   getval(count, M).
N = 3    M = 4
% N is the number of times the query Q succeeds

succeed(Q,N) :-
   (setval(count,0),
    Q,
    incval(count),
    fail
   ;
    true
   ),
   getval(count,N).
```

Counting the number of backtracks

```
search(List, Backtracks) :-
   init_backtracks,
   ( fromto(List, Vars, Rest,[])
   do
        choose_var(Vars, Var-Domain, Rest),
        choose_val(Domain, Val),
        Var = Val,
        count_backtracks
   ),
   get_backtracks(Backtracks).

init_backtracks :- setval(backtracks,0).
get_backtracks(B) :- getval(backtracks,B).
count_backtracks :- on_backtracking(incval(backtracks)).
on_backtracking(_).         % Until a failure happens do nothing.
                            % The second clause is entered
on_backtracking(Q) :-       % on backtracking.
   once(Q),                 % Query Q is called, but only once.
   fail.                    % Backtracking continues afterwards.
```

2. DELAYING CONSTRAINTS USING THE SUSPEND LIBRARY

2. Delaying automatically constraints using the suspend library

- Why delay a constraint?
- What do we do with delayed constraints?
- Still only passive constraints
- First step towards realizing constraint programming; used by more sophisticated constraint solvers
- Core constraints and user defined constraints
- Examples using the `suspend` library

Interleaving generate and test

```
solve(List) :-
    declareDomain(List),          %info about domains
    search(List),                 %launch search process
    testConstraints(List).
```

What can be changed??

```
:- library(my_library).         % e.g. suspend
solve(List) :-
    declareDomain(List),          %info about domains
    generateConstraints_andcosts(List, Cost),
    search(List, Cost).           %launch search process
```

Library issues

```
[eclipse 1]: 2 < Y + 1, Y = 3.
instantiation fault in +(Y, 1, _173)
Abort

[eclipse 2]: suspend:(2 < Y + 1), Y = 3.
Y = 3
Yes

% delays the </2 constraint until it becomes ground
```

Meta-interpreter for Prolog with built-ins

```
% solve(X) :-
% the query X succeeds for the Prolog
% program accessible by clause/2.
solve(true) :- !.
solve((A,B)) :- !, solve(A), solve(B).
solve(A) :- rule(A, B), solve(B).

rule(A,B) :-
  functor(A,F,N), is_dynamic(F/N),
  clause(A,B).                        % user defined
rule(A, true) :- A.                   % for built-ins
```

Meta-interpreter for the suspend library

```
% pass delayed goals around; delay; re-activate/trigger
% Susp is a conjunction of suspend:G goals

solve(true, Susp, Susp):- !.
solve((A,B), SuspIn, SuspOut) :- !,
   solve(A, SuspIn, Susp2), solve(B, Susp2, SuspOut).
solve(A, Susp, (A, Susp)) :- postpone(A),!.     % delayed
solve(H, SuspIn, SuspOut) :- rule(H, B),     % not delayed
   solve(SuspIn, true, Susp2),
   solve(B, Susp2, SuspOut).

postpone(suspend:A) :- not ground(A).

rule(A,B) :-functor(A,F,N), is_dynamic(F/N), clause(A,B).
rule(suspend:A, true) :- !, A.
rule(A, true) :- A.
```

Core constraints in ECLiPSe

- Available in all the constraint solvers where they make sense
  - Boolean constraints
  - Arithmetic constraints
  - Variable declarations
  - so-called Reified constraints
- The programmer uses them to model the CSP (generate constraints) and can send them to several constraint solvers, also to the `suspend` library

Boolean constraints

```
[eclipse 1]: suspend:(X or Y), X = 0.   % 0 for false
X = 0
Y = Y
Delayed goals: suspend: (0 or Y)   % waits grounding
Yes

[eclipse 2]: suspend:(X or Y), X = 0, Y = 1.
X = 0
Y = 1
Yes
% also and/2, neg/1, =>/2
```

What happens with a core constraint that becomes fully instantiated?

Known: Arithmetic comparison predicates

- Less than `<`
- Less than or equal `=<`
- Equality `=:=`
- Disequality `=\=`
- Greater than or equal `>=`
- Greater than `>`

Available as core constraints: `suspend:(1+Y>3)`

Shorthands for arithmetic constraints

once the suspend library is loaded

`1 + 2 $= Y` is a shorthand for

```
    suspend:( 1 + 2 =:= Y)
```

```
also $<, $=<, $\=, $>=, $>   (for reals)
```

```
also for integers #<, #=<, #\=, #>=, #>, #=
```

Quicksort with delayed tests

```
% qs(Xs, Ys) :-
% Ys is an =<-ordered permutation of the list Xs.
qs([], []).
qs([X | Xs], Ys) :- part(X, Xs, Littles, Bigs),
   qs(Littles, Ls), qs(Bigs, Bs),
   app(Ls, [X | Bs], Ys).
% part(X, Xs, Ls, Bs) :-
% Ls is a list of elements of Xs which are < X,
% Bs is a list of elements of Xs which are >= X
part(_, [], [], []).
part(X, [Y | Xs], [Y | Ls], Bs) :-
   X $> Y, part(X, Xs, Ls, Bs).
part(X, [Y | Xs], Ls, [Y | Bs]) :-
   X $=< Y, part(X, Xs, Ls, Bs).

[eclipse 5]: qs([3.14,Y,1,5.5],[T,2,U,Z]).    %???
```

Variable declarations: just unary constraints

- are not really relevant for suspend context; only used as a test whether the variable becomes correctly instantiated.

```
[S,E,N,D,M,O,R,Y] :: 0..9      % over an integer interval

[eclipse 1]: X :: 1..9, X = 5
X = 5
Yes
[eclipse 2]: X :: 1..9, X = 0
No
[eclipse 3]: X $:: 1..9, X = 2.5   % over a real interval
X = 2.5                           % or use reals as bounds
Yes
[eclpse 4]:  X :: 1 .. 9, X = 2.5.
No (0.00s cpu)

[eclipse 5]: reals(X), X = [1,2.3], reals(Y), Y = [1,a].
%kind of type declaration
```

Reified constraints

- are constraints that can be switched to true or false by setting an extra Boolean variable
- all the core constraints can be reified
- `[eclipse 11]: $>(5,4,1).`
  `Yes`
- `[eclipse 12]: $>(4,5,1).`
  `No`
- `[eclipse 12]: $>(4,5,Bool).`
  `Bool = 0`
- `[eclipse 13]: $::(X,1..9,0), X = 10.`
  `Yes`

Reification (once more)

- From Latin
- *res* thing + *facere* to make
- reification can be 'translated' as thing-making; the turning of something abstract into a concrete thing or object.

User defined suspensions

```
[eclipse 4]: suspend( X =:= 10, 3, X -> inst).
X = X
Delayed goals:  X =:= 10

[eclipse 5]: suspend( X =:= 10, 3, X -> inst), X is
  2 + 8.
X = 10
Yes
```

2nd argument is priority of the goal when it wakes up

3rd argument is wakeup condition Term -> Cond

xor(X,Y) has to wake up when both variables are instantiated

```
susp_xor(X,Y) :-
   ( nonvar(X) ->
     susp_y_xor(X,Y)
   ;
     suspend( susp_y_xor(X,Y), 3, X -> inst)
   ).

susp_y_xor(X,Y) :-
   ( nonvar(Y) ->
     xor(X,Y)
   ;
     suspend(xor(X,Y), 3, Y -> inst)
    ).
xor(1,0).
xor(0,1).
```

Examples

```
?- susp_xor(X, Y).
X = X     Y = Y
There is 1 delayed goal. (0) <3>  susp_y_xor(X, Y)

?- susp_xor(X, Y), X = 0.
X = 0
Y = Y
There is 1 delayed goal. (0) <3>  xor(0, Y)

?- susp_xor(X, Y), Y = 1.
X = X
Y = 1
There is 1 delayed goal. (0) <3>  susp_y_xor(X, 1)

?- suspend(xor(X, Y), 3, [X, Y] -> inst), Y = 0.
X = 1    Y = 0                    % one of [X, Y]
Yes (0.00s cpu)
```

Generating CSPs

- $< x \neq y, y \neq z, x \neq z;$
  $x \in \{0,1\}, y \in \{0,1\}, z \in \{0,1\}>$

```
[eclipse 1]: [X,Y,Z] :: 0..1, X #\= Y, Y #\= Z,
                   X #\= Z.
X = X  Y = Y  Z = Z
There are 4 delayed goals.
  (0) <2>  suspend : ([X, Y, Z] :: 0 .. 1)
  (0) <2>  suspend : (X #\= Y)
  (0) <2>  suspend : (Y #\= Z)
  (0) <2>  suspend : (X #\= Z)
```

Generating CSPs

- x1 < x2, x2 < x3, … , xn-1 < xn ;
  x1 ∈ { 1..1000}, …, xn ∈ {1..1000}>

```
[eclipse 2]: List = [X,Y,Z,U,V,W], List :: 1..1000,
  ordered(List).

ordered(List) :-
    ( fromto(List,[El|Rest],Rest,[])
    do
      ordered(El, Rest)
    ).
```

diff_list(List) succeeds when List is a list of different values

- write it

Generating CSPs

- Why suspend?? what to delay??
- Results in an adequate reordering of the goals so that they are evaluated as soon as their arguments have become instantiated.
- Examples:
  - SEND+MORE=MONEY
  - Map colouring
  - N-queens
- Array representation in ECLiPSe

SMM: representation 1

- 1 equality constraint

$$\begin{aligned}&1000.S + 100.E + 10.N + D\\+ &1000.M + 100.O + 10.R + E\\= &10000.M + 1000.O + 100.N + 10.E + Y,\end{aligned}$$

- 2 disequality constraints: $S \neq 0$, $M \neq 0$
- And 28 disequality constraints $x \neq y$ for x,y ranging over the set {S,E,N,D,M,O,R,Y}

```
solve(List) :-
  declareDomain(List),
  generateConstraints(List),
  search(List).
```

SMM with :-lib(suspend)

```
send(List):-
  List = [S,E,N,D,M,O,R,Y],
  List :: 0..9,
  diff_list(List),
              1000*S + 100*E + 10*N + D
            + 1000*M + 100*O + 10*R + E
  $= 10000*M + 1000*O + 100*N + 10*E + Y,
  S $\= 0, M $\= 0,
  search(List).
search(List) :-
  ( foreach(Var,List) do select_val(0, 9, Var) ).
```

select_val(Min,Max,Val)

```
% Min, Max are ground arithmetic expressions
% and Val is an integer between Min and Max inclusive.

select_val(Min, Max, Val) :- Min =< Max, Val is Min.
select_val(Min, Max, Val) :-
  Min < Max,
  Min1 is Min+1,
  select_val(Min1, Max, Val).
```

Programs can be found at

- http://homepages.cwi.nl/~apt/eclipse/

- send_more_money_ch9.pl
- map_colouring.pl
- queens_ch9.pl

Arrays in ECLiPSe : creation

Structures with functor `[]` and `dim/2` built-in

```
[eclipse 1]: dim(Array, [3])
Array = [](_162,_163,_164)
Yes

[eclipse 2]: dim(Array, [3,2])
Array = []([](_174,_175),[](_171,_172),[](_168,_169))
Yes
```

Arrays: set/get value

```
[eclipse 2]: dim(Array, [3,2]),
  subscript(Array,[1,2],5).
Array = [](([](_174,5),[](_171,_172),[](_168,169))
Yes

[eclipse 3]: dim(Array, [3,2]),
  subscript(Array,[1,2],5), X is Array[1,2] - 2,
  Y = f(Array[1,2]).
…
  X = 3
  Y = f(??)
```

Arrays and is/2

```
[eclipse 5]: A = []([](1,2),[](3,4),[](5,X)),
   El is A[3,2],
   Row is A[1, 1..2],
   Col is A[2..3, 2],
   Sub is A[2..3,1..2],
   RowOne is A[1,*].

Row = [](1,2), Col =[](4,X),
   Sub = []([](3, 4), [](5, X)), RowOne = [](1,2)

% old: Eclipse 6
    Row = [1,2], Col = [4,X]
    Sub = [[3,4],[5,X]]       % subarray as list of lists
```

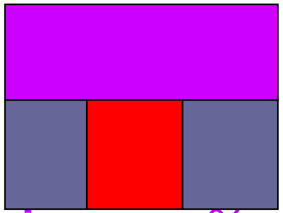
Array iterator: foreachelem/2

```
[eclipse 6]: dim(Array,[3,2]),
  ( foreach(El,[e11,e12,e21,e22,e31,e32]),
    foreachelem(El, Array)
  do
    true
  ),
  X is Array[2,2].

Array = [] ([] (e11,e12),[] (e21,e22),[] (e31,e32))
X = e22
```

More iterators

```
[eclipse 1]: ( for(I,1,3)
              do
                 ( for(J,5,9),
                   param(I)
                  do
                    K is I*J, write(K), write(‘ ’)
                  )
               ).
5 6 7 8 9 10 12 14 16 18 15 18 21 24 27

[eclipse 2]: ( multifor([I,J],[1,5],[3,9])
              do
               K is I*J, write(K), write(‘ ‘)
             ).
 5 6 7 8 9 10 12 ...
```

Map colouring



- A finite set of regions `Regions % array`
- A (smaller) set of colours `colour(1).  %blue`
- A neighbour relation between pairs of regions

`neighbour(1,2).    neighbour(1,3).`

Associate a colour with each region so that no two neighbours have the same colour!

Check constraints ASAP!!!

Decision variables? `dim(Regions,[Count])`

Domains? Constraints?

Map colouring with lib(suspend)

```
colour_map(Regions) :-
   constraints(Regions),
   search(Regions).              % problemspecs : colour/1
                                 % neighbour/2

constraints(Regions) :-
   no_of_regions(Count),         % also nb. of regions
   dim(Regions,[Count]),
  ( multifor([I,J],1,Count),
    param(Regions)
  do
    ( neighbour(I, J) -> Regions[I] $\= Regions[J]
      ;
      true
    )
  ).
search(Regions):- ( foreachelem(R,Regions) do colour(R) ).
```

N-queens (repr. 2)

- x_i denotes the position of the queen in the ith column. `% 1-dim array`
- Implies that no two queens are placed in the same column.
- For $i \in [1..n]$ and $j \in [1..i-1]$
  - At most one queen per row: $x\_i \neq x\_j$
  - At most one queen per SE-NW diagonal $x\_i - x\_j \neq i - j$
  - At most one queen per SW-NE diagonal $x\_i - x\_j \neq j - i$

N-queens with lib(suspend)

```
queens(QueenStruct, Number) :- dim(QueenStruct,[Number]),
   constraints(QueenStruct, Number), search(QueenStruct).

constraints(QueenStruct, Number) :-
   ( for(I,1,Number),
     param(QueenStruct,Number)
   do
     QueenStruct[I] :: 1..Number,
    ( for(J,1,I-1),
      param(I,QueenStruct)
    do
      QueenStruct[I] $\= QueenStruct[J],
      QueenStruct[I]-QueenStruct[J] $\= I-J,
      QueenStruct[I]-QueenStruct[J] $\= J-I
    )
   ).
search(QueenStruct) :- dim(QueenStruct,[N]),
   ( foreachelem(Col,QueenStruct), param(N)
   do select_val(1, N, Col)
   ).
```