APLAI
Active Constraints in ECLiPSe

Gerda Janssens
Departement computerwetenschappen
A01.26
http://www.cs.kuleuven.be/~gerda/APLAI

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
   (NOT: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)

3. Constraint Propagation in ECLiPSe

- The Interval Constraints (`ic`) library

  Differences with a plain finite domain solver:
  - Real-valued variables
  - Integrality is a constraint
  - Infinite domains supported
  - Subsumes finite domain functionality
- Disjunctive constraints and reification

Constraint Propagation

Active constraints.

Removal of values from domains of variables that do not participate in any solution.

Eager constraint propagation.

Set of delayed constraints is called the constraint store.

If a delayed constraint becomes solved, it is removed from the constraint store.

If the constraint store becomes failed, backtracking is triggered.

A computation is successful if it yields an answer and the constraint store is empty.

Constraints over interval constraints (`ic`)

- support for all the core constraints
- enhances the capabilities of `suspend` library (and Prolog)
- variable declaration syntax: `[Y,Z] :: [1..3,5,7..9]`

```
[eclipse 1]: ic:(X or Y), X = 0.
X = 0  Y = 1
Yes
[eclipse 2]: [X,Y]::1..4, X/2 - Y/2 #= 1.
X = 4 Y = 2        % #= imposes: all subexpressions integral
Yes                % 3/2 is not integral
```

- use of $-syntax: switch freely between real(continuous) domains and integer (finite) domains.
- automatic initialisation `-1.0Inf .. 1.0Inf`

Interval variables

```
Vars :: Domain
e.g.  X :: 1..9              X #:: 1..9
      Y :: [2,5..7]          Y #:: [2,5..7]
      Z :: -0.5..3.5         Z $:: -0.5..3.5
      W :: -0.5..1.0Inf      W $:: -0.5..1.0Inf
      V :: -1.0Inf..3        V #:: -1.0Inf..3
```

- Attaches an initial domain to a variable or intersects its old domain with the new one.
- Type of bounds gives type of variable for `::`
  (`1.0Inf` is considered type-neutral)
- `#::` always imposes integrality
- `$::` never imposes integrality

Some ic examples

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

ordered(_, []).
ordered(X, [Y|_]) :- X $< Y.


[eclipse 1]: [X,Y,Z,U,V]::1..1000, ordered([X,Y,Z,U,V]).
X = X{1 .. 996}  Y = {2 .. 997}
ic: (Y{2 .. 997} - X{1 .. 996} > 0)  …
```

More examples: propagating bound information (interval reasoning)

```
[eclipse 1]: X::[5..10], Y::[3..7], X $< Y.
X = X{[5,6]}
Y = Y{[6,7]}
(0) <3>  ic : (Y{[6,7]} - X{[5,6]} > 0)

[eclipse 2]: X::[5..10], Y::[3..7], X $< Y, X $\= 6.
X = 5           % \=  arc-consistency
Y = Y{[6,7]}

[eclipse 3]: [X,Y,Z]:: [1..1000],
             X $> Y, Y $> Z, Z $> X.
No     %  bounds propagation iterates over bounds …
```

Examples: arc consistency for disequality

```
[eclipse 1]: X :: 0..10, X #\=3 . %!!integer variable
X = X{[0 .. 2, 4 .. 10]}

[eclipse 2]: [X,Y] :: 0..10, X - Y #\=3 .
X = X{0 .. 10}
Y = Y{0 .. 10}
(0) <3>  -(Y{0 .. 10}) + X{0 .. 10} #\= 3


[eclipse 3]: [X,Y] :: 0..10, X - Y #\=3 , Y = 2 .
X = X{[0 .. 4, 6 .. 10]}
Y = 2
```

Different forms of propagation

- Forward checking: when a variable X is assigned a value, FC looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value of Y that is inconsistent with the value chosen for X.
- Bounds consistency: if for the lower bound and the upper bound of every variable in a constraint, it is possible to find values for all its other variables between their lower and upper bounds which satisfy the constraint

Different forms of propagation

- Domain consistency: if for every variable and every value in its domain in a constraint, it is possible to find values in the domains of all its other variables which satisfy the constraint

Different behaviours (14.3 Tutorial)

- **Consistency Checking** wait until all variables instantiated, then check
- **Forward Checking** wait until one variable left, then compute consequences
- **Bounds Consistency** wait until a domain bound changes, then compute consequences for other bounds
- **Domain (Arc) Consistency** wait until a domain changes, then compute consequences for other domains

Propagation incompleteness

- It is important to understand that this kind of propagation incompleteness does not affect correctness: the constraint will simply detect the inconsistency later, when its arguments have become more instantiated.
- In terms of the search tree, this means that a branch will not be pruned as early as possible, and extra time might be spent searching.

An arithmetic puzzle

Is there a positive number which

- when divided by 3 gives a remainder of 1;
- when divided by 4 gives a remainder of 2;
- when divided by 5 gives a remainder of 3;

and

- when divided by 6 gives a remainder of 4?

(express the constraints with multiplications rather than divisions)

```
model(X) :-
    X #> 0,
  X #= A*3 + 1,
  X #= B*4 + 2,
  X #= C*5 + 3,
  X #= D*6 + 4.
```

An arithmetic puzzle

```
model(X) :-
    X #> 0,
    X #= A*3 + 1,
    X #= B*4 + 2,
    X #= C*5 + 3,
    X #= D*6 + 4.

?- model(X).
X = X{58 .. 1.0Inf}
Delayed goals:
    ic:(-3*A{19..1.0Inf} + X{58..1.0Inf} =:= 1)
    ic:(-4*B{14..1.0Inf} + X{58..1.0Inf} =:= 2)
    ic:(-5*C{11..1.0Inf} + X{58..1.0Inf} =:= 3)
    ic:(X{58..1.0Inf} - 6*D{9..1.0Inf} =:= 4)
Yes
```

An arithmetic puzzle

```
?- model(X).
X = X{58 .. 1.0Inf}
Delayed goals:
    ic:(-3*A{19..1.0Inf} + X{58..1.0Inf} =:= 1)
    ic:(-4*B{14..1.0Inf} + X{58..1.0Inf} =:= 2)
    ic:(-5*C{11..1.0Inf} + X{58..1.0Inf} =:= 3)
    ic:(X{58..1.0Inf} - 6*D{9..1.0Inf} =:= 4)
Yes

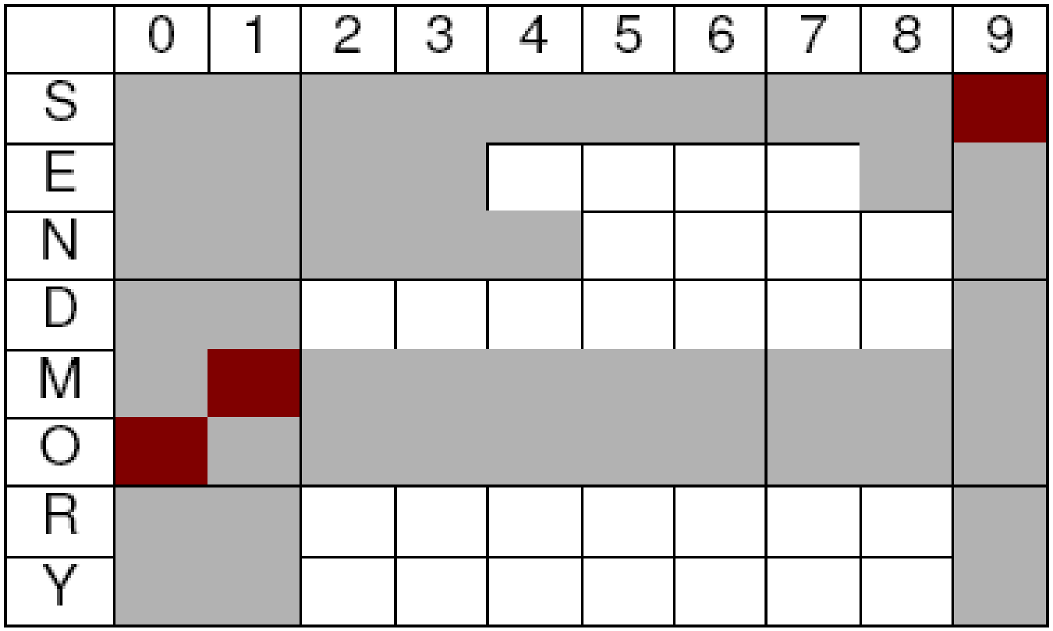
?- model(X), labeling([X]).
X = 58      More? (;)
X = 118     More? (;)
X = 178     More? (;)
...
```

SMM example with :- lib(ic)

```
send(List):-
   List = [S,E,N,D,M,O,R,Y],
   List :: 0..9,
   alldifferent(List),                % was 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.

[eclipse 6]: send(List)
List = [9, E{4..7}, N{5..8}, D{2..8}, 1, 0, R{2..8},
   Y{2..8}]             11 delayed goals
with search List = [9, 5, 6, 7, 1, 0, 8, 2]
reduction of execution time : from more 40.19s (suspend
   version)  to 0.00s
```

Domains after setup of constraints



Todo: follow animation http://4c.ucc.ie/~hsimonis/ELearning/index.htm
Chapter 4 Basic Constraint Reasoning, slides 31-94

N-queens with lib(ic)

- constraint propagation has no effect before search is launched.
- each time a `QueenStruct[I]` gets instantiated to a value, some, possibly different, values are removed from the domains of other subscripted variables (!!! `$\=`!!!)
- execution time from 78s to 0.01s

```
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)
    ).
```

Different constraint behaviours

lib(ic) implementation of alldifferent/1

```
?- [A,B,C]::1..3, D::1..5, ic:alldifferent([A,B,C,D]).
A = A{1 .. 3}
B = B{1 .. 3}
C = C{1 .. 3}
D = D{1 .. 5}
Delayed goals:
    outof(A{1 .. 3}, [], [B{1 .. 3}, C{1 .. 3}, D{1 .. 5}])
    outof(B{1 .. 3}, [A{1 .. 3}], [C{1 .. 3}, D{1 .. 5}])
    outof(C{1 .. 3}, [B{1 .. 3}, A{1 .. 3}], [D{1 .. 5}])
    outof(D{1 .. 5}, [C{1 .. 3}, B{1 .. 3}, A{1 .. 3}], [])
Yes
```

Different constraint behaviours

- lib(ic_global) implementation

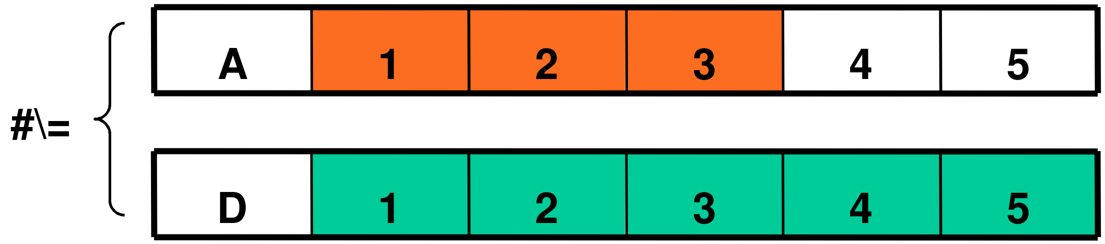
```
?- [A,B,C] :: 1..3, D::1..5,
   ic_global:alldifferent([A,B,C,D]).
A = A{1 .. 3}
B = B{1 .. 3}
C = C{1 .. 3}
D = D{[4, 5]}

Delayed goals:
    alldifferent([A{1 .. 3}, B{1 .. 3}, C{1 .. 3}], 1)
Yes
```

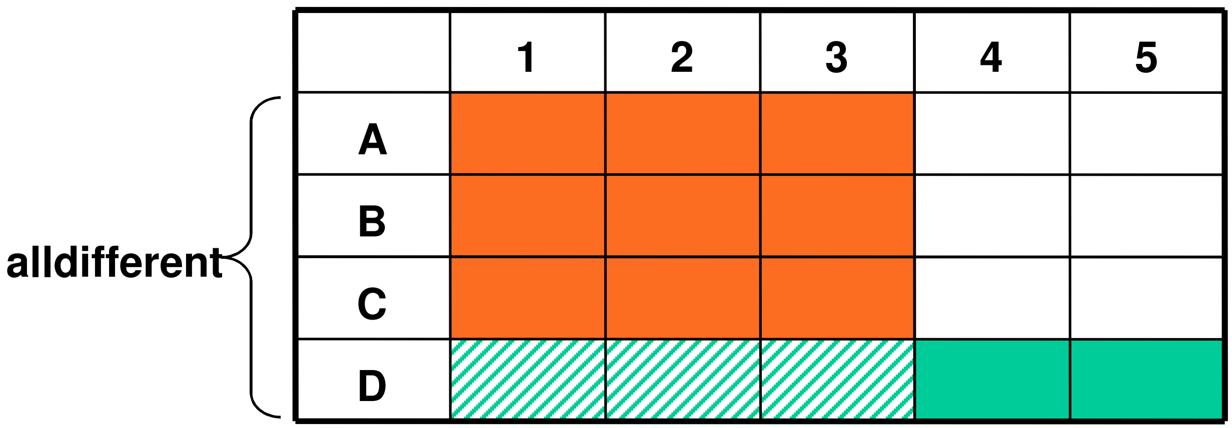
- ```
  :- lib(ic).
  :- import alldifferent/1 from ic_global.
  ```
  ```

Why is it better?

Primitive constraints see only e.g.



Global view enables more reasoning:



Different alldifferent versions

- Forward Checking: lib(ic)
  - Only reacts when variables are assigned
  - Equivalent to decomposition into binary constraints
- Bounds Consistency: lib(ic_global)
  - Typical best compromise speed/reasoning
  - Works well if no holes in domain
- Domain Consistency: lib(ic_global_gac)
  - Extracts all information from single constraint
  - Cost only justified for very hard problems

Constraint Propagation for non-linear constraints

```
?- [X, Y] :: [1.0 .. 1000], X * X + Y * Y $= 1000.
X = X{1.0 .. 31.606961258558218}
Y = Y{1.0 .. 31.606961258558218}
There are 3 delayed goals.     % sqr(square), not sqrt!
 (0) <4>  ic : (_694{1.0 .. 999.0} =:= sqr(Y{1.0 ..
  31.606961258558218}))
  (0) <4>  ic : (_803{1.0 .. 999.0} =:= sqr(X{1.0 ..
  31.606961258558218}))
  (0) <3>  ic : (_694{1.0 .. 999.0} + _803{1.0 ..
  999.0} =:= 1000)


% auxiliary variables
```

Non-linear combined with integrality

```
?- [X, Y] :: [1 .. 1000], X * X + Y * Y $= 1000.
X = X{10 .. 30}
Y = Y{10 .. 30}
There are 3 delayed goals.
(0) <4>  ic : (_696{100.0 .. 900.0} =:= sqr(Y{10 ..
   30}))
(0) <4>  ic : (_805{100.0 .. 900.0} =:= sqr(X{10 ..
   30}))
(0) <3>  ic : (_696{100.0 .. 900.0} + _805{100.0 ..
   900.0} =:= 1000)
```

Narrowing the bounds for continuous variables

```
?- X :: 1 .. 1000, X * (X + 1) * (X + 2) $=< 1000.
X = X{1 .. 22}
There are 6 delayed goals.

?- X :: 1 .. 1000, X * (X + 1) * (X + 2) $=< 1000,
   squash([X], 0.1, lin).
X = X{1 .. 9}
There are 6 delayed goals.

% arg 2: how near to the bounds values are tried
   (and if no solution is found, domain is narrowed)
% arg 3: whether to divide the domain lin or log
```

Disjunctive constraints and reification

- Uptill now, generation of constraints did not leave any choicepoints!!!!
- How to express a disjunctive constraint `|X - Y| $= Z`

```
dist(X,Y,Z) :- X - Y $= Z .
dist(X,Y,Z) :- Y - X $= Z .
[eclipse 1]: X::[1..4], Y::[3..6], dist(X,Y,1).
X = 4  Y = 3
and on backtracking  X = X{2..4}  Y = Y{3..5} and
  delayed goal ic : (X{2..4} - Y{3..5} =:= -1)
```

Not a good idea: choicepoints should be only in search part; they reduce amount of propagation; dynamic reordering of choices is not allowed.

Disjunctive constraint using abs/1

```
[eclipse 3]: X::[1..4], Y::[3..6], abs(X-Y) $= 1

X = X{2..4}  Y = Y{3..5} and  delayed goals

(0) <4>  ic : (_730{-1 .. 1} + Y{3 .. 5} - X{2 .. 4}
  =:= 0)
(0) <4>  ic : (1 =:= abs(_730{-1 .. 1}))
```

General Approach: Reified constraints

```
X #> Y          #>(X, Y, B)    % reified version of #>
X #= Y          #=(X, Y, B)
```

% B indicates the truth of the constraint

- B=1 if the constraint is satisfied (entailed)
- B=0 if the constraint is false (disentailed)
- B{0..1} while unknown

B can be set to

- 1 to enforce the constraint
- 0 to enforce its negation

Disjunctive constraints using reification

```
dist(X, Y, Z) :-
   $=(X-Y, Z, B1),      % ternary reified version $=
   $=(Y-X, Z, B2),      % B2 indicates the truth of Y-X $= Z
   B1 + B2 $>= 1.

% equivalently with the boolean constraint or/2
dist(X, Y, Z) :- X - Y $= Z or Y - X $= Z.

[eclipse 7]: X::[1..4],Y::[3..6],
             X - Y $= 1 or Y - X $= 1.
X = X{1 .. 4}
Y = Y{3 .. 6}
There are 3 delayed goals.
 (0) <3>  ic : =:=(-(Y{3 .. 6}) + X{1 .. 4}, 1, _917{[0, 1]})
 (0) <3>  ic : =:=(Y{3 .. 6} - X{1 .. 4}, 1, _1006{[0, 1]})
 (0) <3>  -(_1006{[0, 1]}) - _917{[0, 1]} #=< -1
```

Exercise 1: Disjunctive constraints via reified constraints

```
no_overlap(S1, D1, S2, D2) :-
    #>=(S2, S1+D1, B), #<(S1, S2+D2, B).
```

What constraint is imposed on

task 1 with start time S1 and duration D1

task 2 with start time S2 and duration D2

(what if B equals 1? what if B equals 0?)

Exercise 2: using reified constraints

- **occurrences(++Value, +Vars, ?N)**
- The value Value occurs in Vars N times

- *Value* Atomic term
- *Vars* Collection (a la collection_to_list/2) of atomic terms or domain variables
  % thus an ordinary list or an array
- *N* Variable or integer

4. Top-down search with active constraints

1. Backtrack-free search
2. Shallow backtracking search
3. Backtracking search
4. Variable ordering heuristics
5. Value ordering heuristics
6. The `search/6` generic search predicate

4.1 Backtrack-free search

- Assume some fixed variable ordering.
- Variable by variable assign to it (X) the minimum(maximum) value (Min) in its current domain.
  `get_min(X,Min)`
- Domain reduction by constraint propagation.
- Complete search procedure??

Backtrack-free n-queens

```
queens(QueenStruct, Number) :- dim(QueenStruct,[Number]),
   constraints(QueenStruct, Number),
   backtrack_free(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
    )
   ).
backtrack_free(QueenStruct) :-
   ( foreachelem(Col,QueenStruct) do get_min(Col,Col) ).
```

% for N < 500 only a solution for 5 and 7 is found

4.2 Shallow backtracking search

- Limited form of backtracking in value selection process.
- When trying to assign a value to the current variable, constraint propagation is done and if it yields a failure, the next value is tried.
  Once a non-failure value is found, its choice is final.

```
[eclipse 3]: X::[1..4], get_domain_as_list(X,Dom),
   X #\=2, get_domain_as_list(X,NewDom).
X = X{[1,3,4]}   Dom = [1,2,3,4]  NewDom = [1,3,4]
```

ECLiPSe built-in `indomain/1`

```
indomain(X) :-
    get_domain_as_list(X, Domain),
    member(X, Domain).

shallow_backtrack(List) :-
    ( foreach(Var, List) do once(indomain(Var))).
% once(Q) generates only the first solution to the
    query Q

[eclipse 1]: List = [X,Y,Z], X::1..3, [Y,Z]::1..2,
    alldifferent(List), shallow_backtrack(List).
% with shallow_backtrack: what if X = 1?  X = 2?
X = 3  Y = 1  Z = 2
```

Shallow backtracking SMM

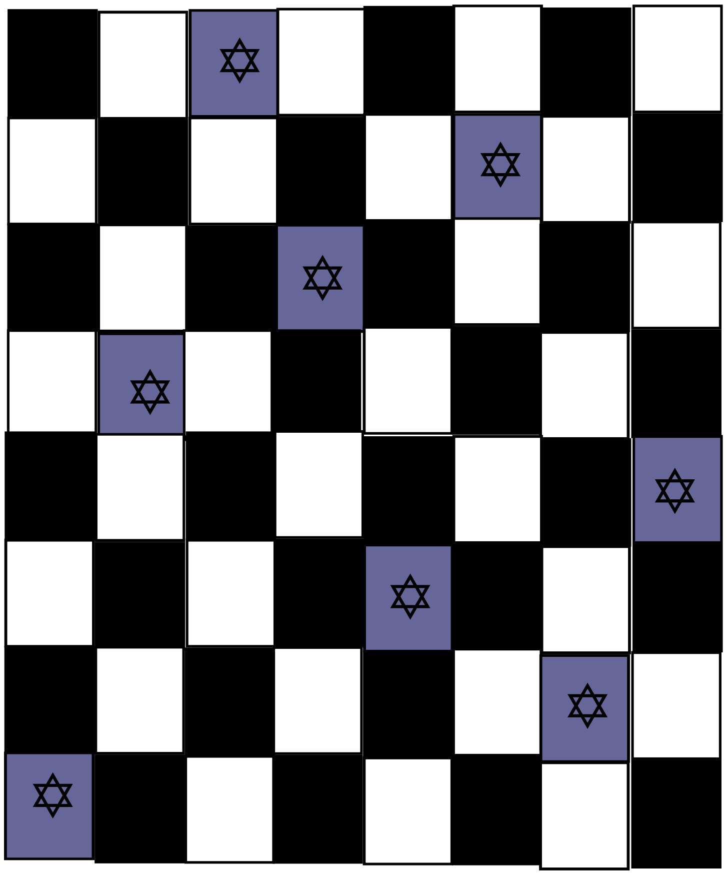
- The unique solution to this puzzle can be found by using shallow backtracking.

4.3 Backtracking search using ***current*** domain

```
search(List) :-   % initial Domain associated with Var
   ( fromto(List, Vars, Rest,[])
   do
     choose_var(Vars, Var-Domain, Rest),
     choose_val(Domain, Val),
     Var = Val
   ).
search_with_dom(List) :-    % just list of decision vars
   ( fromto(List, Vars, Rest,[])
   do
     choose_var(Vars, Var, Rest),
     indomain(Var)
   ).
```

% is actually known as labeling(List)

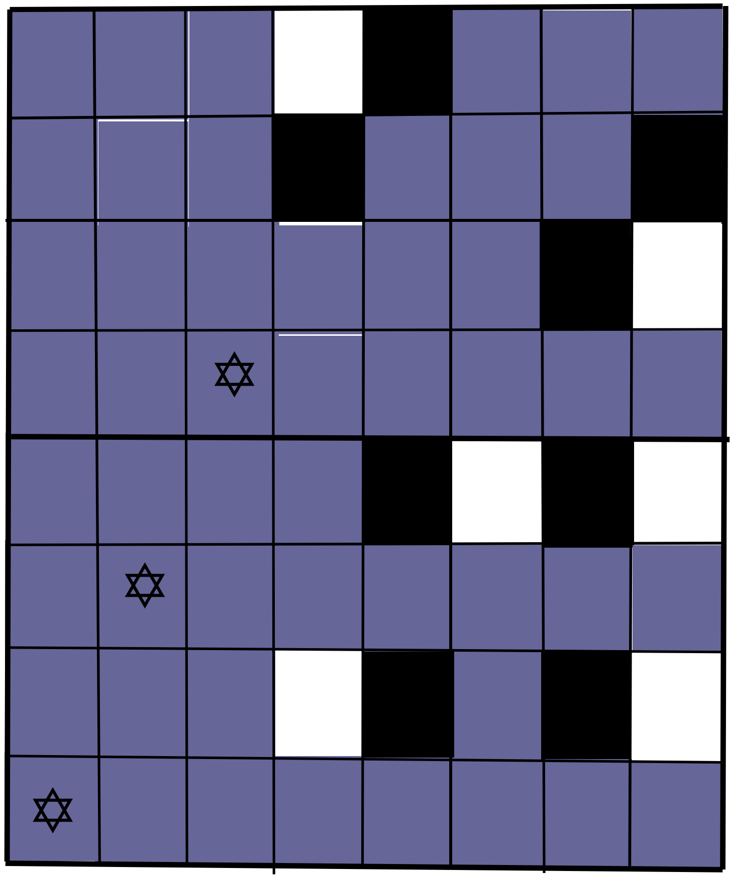
N-Queens with backtrack_search



```
backtrack_search(QueenStruct) :-

  (foreachelem(Col, QueenStruct)
       do indomain(Col)).
```

N-Queens: Propagation and search



Constraint Propagation

- **eliminates choices**
- **e.g. instantiates 6th queen**

Naive search

- **Alright for 8 queens**
- **0.01s first solution**
- **0.2s all 92 solutions**

Better than naieve search: 6th

4.4 Variable ordering heuristics

- Running example n-queens with a parameter `Heur`.
- Convert (array) structure into a list

```
struct_to_list(Struct, List):-
  ( foreacharg(Arg, Struct),
    foreach(Var,List)
  do
    Var = Arg
  ).
```

N-queens with Heur in search predicate

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

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
    )
   ).
```

naive heuristic

- again labeling/1 `search(naive,List)`
- labels the variables in the order they appear.
- tries values starting with the smallest element in the current domain and trying the rest in increasing order.
- for n-queens:
  - solution for 8-queens: after 10 backtracks
  - solves 16-queens with 542 backtracks
  - limit of 50s, even 32-queens timed out

middle_out heuristic (static)

```
:- lib(lists).                    %predefined predicates
middle_out(List, MOutList) :-
  halve(List, FirstHalf, LastHalf),
  reverse(FirstHalf, RevFirstHalf),
  splice(LastHalf, RevFirstHalf, MOutList). %merge

search(middle_out,List):- middle_out(List, MOList),
  labeling(MOList).
```

- for n-queens: start with the centre queens
  - solution for 8-queens: after 0 backtracks
  - solves 16-queens with 17 backtracks
  - limit of 50s, 32-queens timed out

first_fail heuristic (dynamic)

- Select as the next variable the one with the fewest values remaining in its domain
- Label variable with smallest domain first
- Focus on Bottleneck

- Forward-checking enhances first-fail
- Quite general, works almost always

first_fail heuristic (dynamic)

- selects as the next variable the one with the fewest values remaining in its domain

```
search(first_fail, List) :-
  ( fromto(List, Vars, Rest,[])
  do
    delete(Var,Vars,Rest,0,first_fail),  %ic built-in
    indomain(Var)
  ).
```

- for n-queens:
  - solution for 8-queens: after 10 backtracks
  - solves 16-queens with 4 backtracks
  - 75-queens is solved after 818 backtracks !!!!!

middle_out combined with first_fail

```
search(moff, List) :- middle_out(List, MOList),
  ( fromto(MOList, Vars, Rest,[])
  do
    delete(Var,Vars,Rest,0,first_fail),
    indomain(Var)
  ).
```

- for n-queens:
  - solution for 8-queens: after 0 backtracks
  - solves 16-queens with 0 backtracks
  - 75-queens is solved after 719 backtracks

4.5 Value ordering heuristics

- choosing a location near the centre of the board will lead to more conflicts
- start with the middle positions

```
search(moffmo, List) :- middle_out(List, MOList),
  ( fromto(MOList, Vars, Rest,[])
  do
    delete(Var,Vars,Rest,0,first_fail),
    indomain(Var, middle)    % also min and max
  ).
```

Results for n-queens:

number of backtracks

	naive	middle_out	first_fail	moff	moffmo
8-queens	10	0	10	0	3
16-queens	542	17	3	0	3
32-queens	--	--	4	1	7
75-queens	--	--	818	719	0
120-queens	--	--	--	--	0

For map colouring: reusing values is a bad idea;

Thus other … heuristic.

For scheduling and packing problems: min!!

4.6 The `search/6` generic search procedure

```
search(List, Arg, VarChoice, ValChoice, Method, Options)

Arg=0 then List is the list of decision variables
Arg>0 then List is a list of compound terms

VarChoice: predicate name of predefined or user-defined
      input_order  first_fail
ValChoice: indomain  indomain_middle

Method: complete …
Options: []
```

Search/6 as we know it …

```
search(naive,List) :-
    search(List,0,input_order,indomain,complete, []).
search(middle_out,List) :-
    middle_out(List,MOLits),
    search(MOList,0,input_order,indomain,complete, []).
search(first_fail,List) :-
    search(List,0,first_fail,indomain,complete, []).
search(moff,List) :-
    middle_out(List,MOLits),
    search(MOList,0,first_fail,indomain,complete, []).
search(moffmo,List) :-
    middle_out(List,MOLits),
    search(MOList,0,first_fail,
        indomain_middle,complete, []).
```

Search/6 and incomplete search

```
:- lib(ic).
:- lib(lists).

queens(debug, Queens, VarChoice, Method, Number) :-
    init(QueenStruct, Number),
    constraints(QueenStruct, Number),
    struct_to_list(QueenStruct, Queens),
    struct_to_queen_list(QueenStruct, QList), % q(Col,V)
    search(QList, 2, VarChoice, show_choice, Method,
     [backtrack(B)]), writeln(backtracks-B).

show_choice(q(I, Var)) :-
    indomain(Var),
    writeln(col(I):square(Var)).
```

Struct_to_queen_list

```
struct_to_queen_list(Struct,Qlist) :-
   (foreacharg(Var,Struct),
    count(Col,1,_),
    foreach(Term,QList)
   do
     Term = q(Col,Var)
   ).
```

```
?- queens(debug, Qs, first_fail, lds(2), 8).
```

```
Qs = [2, 4, 6, 8, 3, 1, 7, 5]
Yes (0.00s cpu, solution 1, maybe more)
col(1) : square(1)
col(2) : square(3)
col(3) : square(6)
col(1) : square(1)
col(2) : square(3)
col(3) : square(6)
col(3) : square(7)
col(4) : square(2)
col(2) : square(4)
col(3) : square(2)
col(1) : square(2)
col(2) : square(4)
col(5) : square(3)
col(7) : square(7)
col(3) : square(6)
col(4) : square(8)
col(6) : square(1)
col(8) : square(5)
backtracks - 3
```

Own credit based value ordering within search/6

```
search(our_credit_search(Var,Credit),List) :-
   search(List,0,first_fail,
     our_credit_search(Var,Credit,_),complete,[]).

our_credit_search(Var,TCredit,NCredit) :-
   get_domain_as _list(Var,Domain),
   share_credit(Domain,TCredit, DomCredList),
   member(Val-Credit, DomCredList),
   Var = Val,
   NCredit = Credit.
share_credit(DomList,InCredit,DomCredList) :-
   ( fromto(InCredit, TCredit,NCredit,0),
     fromto(DomList,[Val|Tail],Tail, _),
     foreach(Val-Credit, DomCredList)
   do
       Credit is fix(ceiling(TCredit/2)),
       NCredit is TCredit - Credit
).
```

Other search methods

- `?- help(ic:search/6).`

```
Method is one of the following:
  complete, lds(Disc:integer),
% Bounded backtrack search  bbs(20)
  bbs(Steps:integer),
% Depth bounded search   dbs(2,bbs(0))
 dbs(Level:integer, Extra:integer or
     bbs(Steps:integer) or lds(Disc:integer)),
%credit based search  credit(20,bbs(0))
  credit(Credit:integer, Extra:integer  or
  bbs(Steps:integer) or lds(Disc:integer)),
```