Programação em Lógica com Restrições no SICStus Prolog

SICStus Prolog User's Manual (Release 4.3.2)

Secção 10.10: Constraint Logic Programming over Finite Domains—library(clpfd)

Henrique Lopes Cardoso
hlc@fe.up.pt
DEI/FEUP
Novembro 2017

Conteúdo

- Domínios
  - Domínios Booleanos e Reais
  - Domínios Finitos
- Interface do solver CLP(FD)
- Restrições Disponíveis
- Predicados de Enumeração
  - Pesquisa e otimização
- Predicados de Estatísticas

PLR no SICStus Prolog

DOMÍNIOS

Domínios Booleanos e Reais

- Booleanos:
  - Esquema clp(B)
  - use_module(library(clpb)).

- Reais e Racionais
  - Esquema clp(Q,R)
  - use_module(library(clpq)).
  - use_module(library(clpr)).

- Não são abordados na unidade curricular de PLOG!

Domínios Finitos

- *Solver* **clp(FD)** é um instância do esquema geral de PLR (CLP) introduzido em [Jafar & Michaylov 87].

- Útil para modelizar problemas de otimização discreta
  - Escalonamento, planeamento, alocação de recursos, empacotamento, geração de horários, etc.

- Características do *solver* **clp(FD)**:
  - Duas classes de restrições: primitivas e globais
  - Propagadores para restrições globais muito eficientes
  - O valor lógico de uma restrição primitiva pode ser reflectido numa variável binária (0/1) – materialização
  - Podem ser adicionadas novas restrições primitivas escrevendo indexicais
  - Podem ser escritas novas restrições globais em Prolog

PLR no SICStus Prolog

INTERFACE DO SOLVER CLP(FD)

Interface do Solver CLP(FD)

- O solver **clp(FD)** está disponível como uma biblioteca

```
:- use_module(library(clpfd)).
```

- Contém predicados para testar a consistência e o vínculo (*entailment*) de restrições sobre domínios finitos, bem como para obter soluções atribuindo valores às variáveis do problema

- Um **domínio finito** é um subconjunto de inteiros pequenos e uma **restrição sobre domínios finitos** é uma relação entre um tuplo de inteiros pequenos

- Só pequenos inteiros e variáveis não instanciadas são permitidos em restrições sobre domínios finitos
  - Inteiro pequeno: [-2^28,2^28-1] em plataformas de 32-bits, ou [-2^60,2^60-1] em plataformas de 64-bits

Interface do Solver CLP(FD)

- Todas as **variáveis de domínio** têm um domínio finito associado, declarado explicitamente no programa ou imposto implicitamente pelo *solver*
  - temporariamente, o domínio de uma variável pode ser infinito, se não tiver um limite mínimo *(lower bound*) ou máximo (*upper bound*) finito
  - o domínio das variáveis vai-se reduzindo à medida que são adicionadas mais restrições

- Se um domínio ficar vazio então as restrições não são, em conjunto, "satisfazíveis", e o ramo atual de computação falha
- No final da computação é usual que cada variável tenha o seu domínio restringido a um único valor (*singleton*)
  - para tal é necessária, tipicamente, alguma pesquisa

- Cada restrição é implementada por um (conjunto de) propagador(es)
  - indexicais
  - propagadores globais

Colocação de Restrições

- Uma restrição é chamada como qualquer outro predicado Prolog

```
| ?- X in 1..5, Y in 2..8, X+Y #= T.
X in 1..5,
Y in 2..8,
T in 3..13
```

```
| ?- X in 1..5, T in 3..13, X+Y #= T.
X in 1..5,
T in 3..13,
Y in -2..12
```

- A resposta mostra a existência de domínios válidos para as variáveis

Problemas de Satisfação de Restrições

- CSP – *Constraint Satisfaction Problem* (ou PSR – Problema de Satisfação de Restrições)
  - Classe de problemas para os quais o solver clp(FD) é mais adequado

- Objectivo num CSP:
  - Escolher valores (de domínios pré-definidos) para certas variáveis de forma a que as restrições sobre as variáveis sejam todas satisfeitas.

- Exemplo: puzzle "Send More Money"
  - Variáveis: letras S, E, N, D, M, O, R, Y
  - Cada letra representa um dígito entre 0 e 9
  - Problema: atribuir um valor distinto a cada dígito tal que: SEND + MORE = MONEY

Problemas de Satisfação de Restrições

- Passos típicos na escrita de um programa clp(FD):

    1. Declarar variáveis e seus domínios
    2. Colocar as restrições do problema
    3. Pesquisar uma solução possível através de pesquisa com retrocesso (*backtracking*) ou uma solução ótima usando pesquisa tipo *branch-and-bound*

- Por vezes, um passo extra precede a pesquisa: colocação de restrições redundantes de modo a eliminar simetrias e ajudar a reduzir o espaço de pesquisa

Exemplo: SEND+MORE=MONEY

```
:- use_module(library(clpfd)).

puzzle([S,E,N,D,M,O,R,Y]) :-
    domain([S,E,N,D,M,O,R,Y], 0, 9),            % passo 1
    S#>0, M#>0,
    all_different([S,E,N,D,M,O,R,Y]),           % passo 2
    sum(S,E,N,D,M,O,R,Y),
    labeling([], [S,E,N,D,M,O,R,Y]).            % passo 3

sum(S, E, N, D, M, O, R, Y) :-
    1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.

| ?- puzzle([S,E,N,D,M,O,R,Y]).
D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2
```

Exemplo: SEND+MORE=MONEY

- Variáveis e Domínios
  - Os domínios são definidos através do predicado ***domain/3*** e das restrições S**#>**0 e M**#>**0
- Restrições
  - Predicado *sum/8* coloca a restrição essencial
  - Predicado *all_different/1* garante que os valores atribuídos às variáveis serão distintos
- Pesquisa
  - Pesquisa com "backtrack" realizada por ***labeling/2***
  - Podem ser indicadas diferentes estratégias de pesquisa (primeiro parâmetro de labeling)
  - No exemplo é usada a estratégia por defeito: selecionar a variável à esquerda e tentar valores do seu domínio em ordem crescente

Exemplo: SEND+MORE=MONEY

- E se invertermos? Primeiro labeling, depois restringir?

```
puzzle([S,E,N,D,M,O,R,Y]) :-
    domain([S,E,N,D,M,O,R,Y], 0, 9),
    labeling([], [S,E,N,D,M,O,R,Y]),
    S#>0, M#>0,
    all_different([S,E,N,D,M,O,R,Y]),
    sum(S,E,N,D,M,O,R,Y).
```

- Resulta no mecanismo "*GENERATE and TEST*" tradicional!

Restrições Materializadas (*Reified*)

- Por vezes é útil fazer reflectir o valor de verdade de uma restrição numa variável booleana *B* (0/1) tal que:
  - a restrição é colocada se *B* for colocado a 1
  - a negação da restrição é colocada se *B* for colocado a 0
  - *B* é colocado a 1 se a restrição for vinculada (*entailed*)
  - *B* é colocado a 0 se a restrição não for vinculada (*disentailed*)

- Este mecanismo é conhecido como materialização (*reification*)

- Uma restrição materializada é escrita da forma:

  | ?- Constraint #<=> B.

  - onde *Constraint* é uma restrição materializável

Restrições Materializadas (*Reified*)

- Exemplo: ***exactly(X,L,N)***
  - verdadeira se X ocorre exatamente N vezes na lista L
  - pode ser definida como:

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

- Restrições materializáveis podem ser usadas como termos em expressões aritméticas:

```
| ?- X #= 10, B #= (X#>=2) + (X#>=4) + (X#>=8).
B = 3,
X = 10
```

```
| ?- X in 1..3, B #= (X#>=1) + (X#>=2) + (X#>=3),
     labeling([],[X]).
X = 1, B = 1 ? ;
X = 2, B = 2 ? ;
X = 3, B = 3 ? ;
no
```

PLR no SICStus Prolog

RESTRIÇÕES DISPONÍVEIS

Restrições Disponíveis

- Restrições Aritméticas
- Restrições de Pertença
- Restrições Proposicionais
- Restrições Combinatórias
- Restrições definidas pelo Utilizador

Restrições Aritméticas

?Expr RelOp ?Expr

- ***RelOp***: **#=** | **#\=** | **#<** | **#=<** | **#>** | **#>=**
- Expressões podem ser lineares ou não lineares.
- Expressões lineares conduzem a maior propagação
  - por exemplo, *X/Y* e *X mod Y* bloqueiam até *Y* estar "ground" (definido)

- Restrições Aritméticas podem ser materializadas

```
| ?- X in 1..2, Y in 3..5, X#=<Y #<=> B.
B = 1,
X in 1..2,
Y in 3..5
```

- Restrições aritméticas lineares mantêm consistência de intervalos

Restrições Aritméticas
Soma

sum(+Xs,+RelOp,?Value)

- ***Xs*** é uma lista de inteiros ou variáveis de domínio, ***RelOp*** é um operador relacional e ***Value*** é um inteiro ou variável de domínio
- verdadeira se: ***sum(Xs) RelOp Value*** (a soma dos elementos de ***Xs*** tem a relação ***RelOp*** com ***Value***)
- corresponde a *sumlist/2* da *library(lists)*
- utiliza um algoritmo dedicado e é muito mais eficiente do que a colocação de uma série de restrições simples
- não pode ser materializada

• Exemplo:

```
| ?- domain([X,Y],1,10), sum([X,Y],#<,10).
X in 1..8,
Y in 1..8
```

```
| ?- domain([X,Y],1,10), sum([X,Y],#=,Z).
X in 1..10,
Y in 1..10,
Z in 2..20
```

Restrições Aritméticas
Produto Escalar

scalar_product(+Coeffs,+Xs,+RelOp,?Value)

scalar_product(+Coeffs,+Xs,+RelOp,?Value,+Options)

- ***Coeffs*** é uma lista de comprimento *n* de inteiros, ***Xs*** é uma lista de comprimento *n* de inteiros ou variáveis de domínio, ***RelOp*** é um operador relacional e ***Value*** é um inteiro ou variável de domínio
- verdadeira se ***sum(Coeffs*Xs) RelOp Value***
- utiliza um algoritmo dedicado e é muito mais eficiente do que a colocação de uma série de restrições simples
- não pode ser materializada

• Exemplo:

```
| ?- domain([A,B,C],1,5), scalar_product([1,2,3],[A,B,C],#=,10).
A in 1..5,
B in 1..3,
C in 1..2
```

Restrições Aritméticas

Mínimo/Máximo

minimum(?Value,+Xs)
maximum(?Value,+Xs)

- ***Xs*** é uma lista de inteiros ou variáveis de domínio e ***Value*** é um inteiro ou variável de domínio
- verdadeira se ***Value*** é o mínimo (máximo) de ***Xs***
- corresponde a *min_member/2* (*max_member/2*) da *library(lists)*
- não podem ser materializadas

- Exemplo:

```
| ?- domain([A,B],1,10), C in 5..15, minimum(C,[A,B]).
A in 5..10,
B in 5..10,
C in 5..10

| ?- domain([A,B,C],1,5), sum([A,B,C],#=,10), maximum(3,[A,B]).
A in 2..3,
B in 2..3,
C in 4..5
```

Restrições de Pertença (*Membership*)

domain(+Variables,+Min,+Max)

- ***Variables*** é uma lista de variáveis de domínio ou inteiros, ***Min*** é um inteiro ou o átomo ***inf*** (menos infinito), e ***Max*** é um inteiro ou o átomo ***sup*** (mais infinito)
- verdadeira se as variáveis são todas elementos do intervalo ***Min..Max***
- não materializável

?X in +Range

- ***X*** é um inteiro ou variável de domínio e ***Range*** é um ***ConstantRange***
- verdadeira se ***X*** é um elemento do intervalo

?X in_set +FDSet

- ***X*** é um inteiro ou variável de domínio e ***FDSet*** é um conjunto FD
- verdadeira se ***X*** é um elemento do conjunto
- in/2 e in_set/2 mantêm consistência do domínio e são materializáveis

- Exemplo:

```
| ?- domain([X],1,3), X in 3..5 #<=> B, labeling([],[X]).
X = 1, B = 0 ? ;
X = 2, B = 0 ? ;
X = 3, B = 1 ? ;
no
```

```
| ?- X in {1,2,3,5}.
X in(1..3)\/{5}

| ?- list_to_fdset([1,2,3,5],FD), X in_set FD.
FD = [[1|3],[5|5]],
X in(1..3)\/{5}
```

Restrições Proposicionais

- Podem definir fórmulas proposicionais sobre restrições materializáveis
- Exemplo:

  ```
  X #= 4 #\/ Y #= 6
  ```

  expressa a disjunção de duas restrições de igualdade

- As folhas das fórmulas proposicionais podem ser restrições materializáveis, as constantes 0 e 1, ou variáveis binárias (0/1)
- Podem ser definidas novas restrições materializáveis primitivas com "indexicais"

- Mantêm consistência do domínio

- Exemplo:

  ```
  | ?- X in 1..2, Y in 1..10, X #= Y #\/ Y #< X, labeling([],[X,Y]).
  X = 1, Y = 1 ? ;
  X = 2, Y = 1 ? ;
  X = 2, Y = 2 ? ;
  no
  ```

Restrições Proposicionais

#\ :Q
- verdadeira se a restrição Q for falsa

:P #/\ :Q
- verdadeira se as restrições P e Q são ambas verdadeiras

:P #\ :Q
- verdadeira se exatamente uma das restrições P e Q é verdadeira

:P #\/ :Q
- verdadeira se pelo menos uma das restrições P e Q é verdadeira

:P #=> :Q

:Q #<= :P
- verdadeira se a restrição Q é verdadeira ou se a restrição P é falsa

:P #<=> :Q
- verdadeira se P e Q são ambas verdadeiras ou ambas falsas

- Note-se que o esquema de materialização é um caso particular das restrições proposicionais

Restrições Combinatórias

- Restrições Combinatórias são também designadas restrições simbólicas
- Não são materializáveis
- Normalmente mantêm consistência de intervalos nos seus argumentos

Arithmetic-Logical
- *smt/1*
- *count/4*
- *global_cardinality/[2,3]*
- *all_different/[1,2]*
- *all_distinct/[1,2]*
- *nvalue/2*
- *assignment/[2,3]*
- *sorting/3*
- *keysorting/[2,3]*
- *lex_chain/[1,2]*
- *bool_[and,or,xor]/2*
- *bool_channel/4*

Extensional
- *element/3*
- *relation/3*
- *table/[2,3]*
- *case/[3,4]*

Graph
- *circuit/[1,2]*

Scheduling
- *cumulative/[1,2]*
- *cumulatives/[2,3]*
- *multi_cumulative/[2,3]*

Placement
- *disjoint1/[1,2]*
- *disjoint2/[1,2]*
- *geost/[2,3,4]*

Automata
- *automaton/[3,8,9]*

Count

(*deprecated*, ver *global_cardinality*)

count(+Val,+List,+RelOp,?Count)

- ***Val*** é um inteiro, ***List*** é uma lista de inteiros ou variáveis de domínio, ***Count*** é um inteiro ou variável de domínio, e ***RelOp*** é um operador relacional
- verdadeira se *N* é o número de elementos de ***List*** que são iguais a ***Val*** e ***N RelOp Count*** é verdadeiro
- *count/4* é uma generalização de *exactly/3*
- mantém consistência de domínio, mas na prática *global_cardinality/2* é uma alternativa melhor

- Exemplos:

```
| ?- domain([X,Y,Z],1,3),
     count(1,[X,Y,Z],#>,Z).
X in 1..3,
Y in 1..3,
Z in 1..2
```

```
| ?- domain([A,B,C],1,3), X in 2..5,
     count(1,[A,B,C],#=,X), labeling([],[X]).
X = 2, A in 1..3, B in 1..3, C in 1..3 ? ;
A = 1, B = 1, C = 1, X = 3 ? ;
no
```

Restrições Combinatórias

Global Cardinality

global_cardinality(+Xs,+Vals)

global_cardinality(+Xs,+Vals,+Options)

- ***Xs*** é uma lista de inteiros ou variáveis de domínio e ***Vals*** é uma lista de termos *K-V*, onde *K* é um inteiro único e *V* é uma variável de domínio ou um inteiro
- verdadeira se cada elemento de ***Xs*** é igual a um *K* e para cada par *K-V* exatamente *V* elementos de ***Xs*** são iguais a *K*
- se ou ***Xs*** ou ***Vals*** estão “ground”, e noutros casos especiais, mantém a consistência de domínio; a consistência de intervalos não pode ser garantida

- Exemplos:

```
| ?- global_cardinality([A,B,C],[1-2,3-1]).
A in {1}\/{3},
B in {1}\/{3},
C in {1}\/{3}
```

```
| ?- A in 3..10,
     global_cardinality([A,B,C],[1-2,3-1]).
A = 3, B = 1, C = 1
```

Restrições Combinatórias

All Different / All Distinct

all_different(+Variables) / ***all_different(+Variables,+Options)***
all_distinct(+Variables) / ***all_distinct(+Variables,+Options)***

- ***Variables*** é uma lista de variáveis de domínio ou inteiros
- cada variável é restringida a tomar um valor único
  - equivalente a uma restrição **#\=** para cada par de variáveis
- ***Options*** é uma lista de zero ou mais opções
  - ***on(On)*** – quando acordar a restrição:
    - ***dom*** (defeito para all_distinct/[1,2]): domínio de uma variável é alterado
    - ***min***/***max***/***minmax***: "lower/upper/qualquer bound" de um domínio é mudado
    - ***val*** (defeito para all_different/[1,2]): variável fica "ground"
  - ***consistency(Cons)*** – que algoritmo utilizar:
    - ***domain*** (defeito para all_distinct/[1,2]): algoritmo de consistência de domínios
    - ***bound***: algoritmo de consistência de intervalos
    - ***value*** (defeito para all_different/[1,2]): algoritmo idêntico ao conjunto de restrições binárias #\=

- Exemplos:

```
| ?- domain([X,Y,Z],1,3), all_different([X,Y,Z]), X#<Y, labeling([],[X]).
X = 1, Y in 2..3, Z in 2..3 ? ;
X = 2, Y = 3, Z = 1 ? ;
no
```

```
| ?- domain([X,Y,Z],1,2), all_different([X,Y,Z]).
X in 1..2, Y in 1..2, Z in 1..2
```

```
| ?- domain([X,Y,Z],1,2), all_distinct([X,Y,Z]).
no
```

Restrições Combinatórias

Nvalue

nvalue(?N,+Variables)

- ***Variables*** é uma lista de inteiros ou variáveis de domínio com limites finitos e ***N*** é um inteiro ou variável de domínio
- verdadeiro se ***N*** é o número de valores distintos em ***Variables***
- pode ser visto como uma versão relaxada de *all_distinct/2*

- Exemplos:

```
| ?- domain([X,Y],1,3), domain([Z],3,5), nvalue(2,[X,Y,Z]), X#\=Y, X#=1.
X = 1, Y = 3, Z = 3

| ?- domain([X,Y],1,3), domain([Z],1,5), nvalue(2,[X,Y,Z]), X#\=Y, X#=1.
X = 1, Y in 2..3, Z in 1..3

| ?- domain([X,Y],1,3), domain([Z],1,5), nvalue(2,[X,Y,Z]), X#\=Y.
X in 1..3, Y in 1..3, Z in 1..5
```

Assignment

assignment(+Xs,+Ys)
assignment(+Xs,+Ys,+Options)

- ***Xs*** *= [X1,...,Xn]* e ***Ys*** *= [Y1,...,Yn]* são listas de comprimento *n* de variáveis de domínio ou inteiros
- verdadeiro se todos os *Xi*, *Yi* estão em *[1,n]*, são únicos para a sua lista e *Xi=j* sse *Yj=i* (as listas são *duais*)
- ***Options*** é uma lista que pode conter as opções:
  - ***on(On)***, ***consistency(Cons)***: idênticas a *all_different/2*
  - ***circuit(Boolean)***: se *true*, *circuit(Xs,Ys)* tem que se verificar
  - ***cost(Cost,Matrix)***: permite associar um custo à restrição

- Exemplos:

```
| ?- length(Xs,3), domain(Xs,1,3), assignment(Xs,Ys), labeling([],Xs).
Xs = [1,2,3], Ys = [1,2,3] ? ;      Xs = [1,3,2], Ys = [1,3,2] ? ;
Xs = [2,1,3], Ys = [2,1,3] ? ;      Xs = [2,3,1], Ys = [3,1,2] ? ;
Xs = [3,1,2], Ys = [2,3,1] ? ;      Xs = [3,2,1], Ys = [3,2,1] ? ;
no
| ?- assignment([4,1,5,2,3],Ys).
Ys = [2,4,5,1,3]
```

Sorting

sorting(+Xs,+Ps,+Ys)

- captura a relação entre uma lista de valores, uma lista de valores ordenada de forma ascendente e as suas posições na lista original
- ***Xs***, ***Ps*** e ***Ys*** são listas de igual comprimento *n* de variáveis de domínio ou inteiros
- a restrição verifica-se se:
  - ***Ys*** está em ordenação ascendente
  - ***Ps*** é uma permutação de *[1,n]*
  - para cada *i* em *[1,n]*, ***Xs***[*i*] = ***Ys***[***Ps***[*i*]]

- Exemplos:

```
| ?- length(Ys,5), length(Ps,5), sorting([2,7,9,1,3],Ps,Ys).
Ps = [2,4,5,1,3], Ys = [1,2,3,7,9]

| ?- length(Ys,5), length(Ps,5), sorting([2,7,3,1,3],Ps,Ys).
Ps = [2,5,_A,1,_B], Ys = [1,2,3,3,7],
_A in 3..4, _B in 3..4
```

Restrições Combinatórias
Lex Chain

lex_chain(+Vectors)

lex_chain(+Vectors,+Options)

- ***Vectors*** é uma lista de vetores (listas) de variáveis de domínio ou inteiros
- a restrição verifica-se se ***Vectors*** está por ordem lexicográfica ascendente
- ***Options*** é uma lista de:
  - ***op(Op)***, ***increasing***, ***among(Least,Most,Values)***

- Exemplo:

```
| ?- A in {1}\/{3}, B in {1}\/{3}, C in {2}\/{4}, D in {2}\/{4},
     lex_chain([[A,B,C,D],[B,A,D,C],[A,B,D,C],[B,A,C,D]]), labeling([],[A,B,C,D]).
A = 1, B = 1, C = 2, D = 2 ? ;
A = 1, B = 1, C = 4, D = 4 ? ;
A = 3, B = 3, C = 2, D = 2 ? ;
A = 3, B = 3, C = 4, D = 4 ? ;
no
```

Restrições Combinatórias
Element

element(?X,+List,?Y)

- ***X*** e ***Y*** são inteiros ou variáveis de domínio e ***List*** é uma lista de inteiros ou variáveis de domínio
- verdadeira se o ***X***-ésimo elemento de ***List*** é ***Y***
- operacionalmente os domínios de ***X*** e ***Y*** são restringidos de forma a que, para cada elemento no domínio de ***X***, existe um elemento compatível no domínio de ***Y***, e vice versa
- mantém consistência de domínio em ***X*** e consistência de intervalos em ***List*** e ***Y***
- corresponde a *nth1/3* da *library(lists)*

- Exemplos:

```
| ?- element(X,[10,20,30],Y), labeling([],[Y]).
X = 1, Y = 10 ? ;
X = 2, Y = 20 ? ;
X = 3, Y = 30 ? ;
no
```

```
| ?- L=[A,B,C], domain(L,1,5), element(2,L,4).
B = 4,
L = [A,4,C],
A in 1..5, C in 1..5
```

Relation

(*deprecated*, ver *table*)

relation(?X,+MapList,?Y)

- ***X*** e ***Y*** são inteiros ou variáveis de domínio e ***MapList*** é uma lista de pares *Inteiro-ConstantRange*, onde cada chave *Inteiro* ocorre só uma vez
- verdadeira se ***MapList*** contém um par ***X***-*R* e ***Y*** está no intervalo denotado por *R*

- Exemplos:

```
| ?- domain([Y],1,3),
     relation(X,[1-{3,4,5},2-{1,2}],Y),
     labeling([],[X]).
X = 1, Y = 3 ? ;
X = 2, Y in 1..2 ? ;
no
```

```
| ?- domain([Y],1,3),
     relation(X,[1-{3,4,5},2-{1,2,3}],Y),
     labeling([],[Y]).
Y = 1, X = 2 ? ;
Y = 2,X = 2 ? ;
Y = 3, X in 1..2 ? ;
no
```

Restrições Combinatórias

Table

table(+Tuples,+Extension)
table(+Tuples,+Extension,+Options)

- define uma restrição *n*-ária por extensão
- ***Tuples*** é uma lista de listas de variáveis de domínio ou inteiros, cada uma de comprimento *n*, ***Extension*** é uma lista de listas de inteiros, cada uma de comprimento *n*
- a restrição verifica-se se cada *Tuple* em ***Tuples*** ocorre em ***Extension***

• Exemplos:

```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]).
A in 1..2,
B in (1..2)\/{10}\/{20}

| ?- table([[A,B],[B,C]],[[1,1],[1,2],[2,10],[2,20]]).
A = 1,
B in 1..2,
C in (1..2)\/{10}\/{20}
```

```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]),
     labeling([],[A,B]).
A = 1, B = 1 ? ;
A = 1, B = 2 ? ;
A = 2, B = 10 ? ;
A = 2, B = 20 ? ;
no
```

Restrições Combinatórias

Case

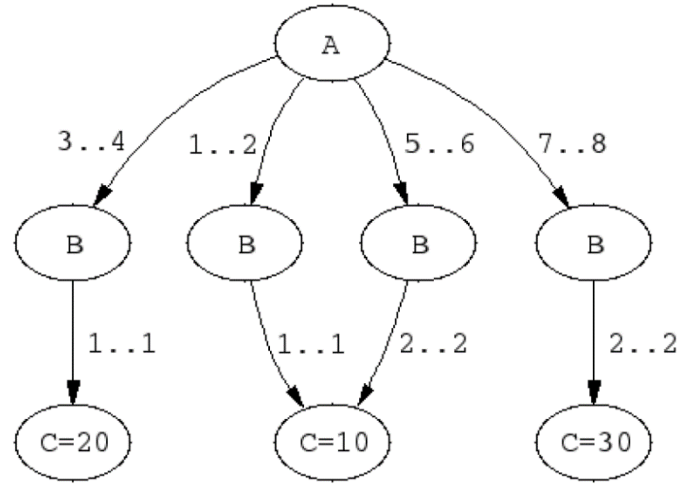
case(+Template,+Tuples,+Dag)
case(+Template,+Tuples,+Dag,+Options)

- codifica uma restrição *n*-ária, definida por extensão e/ou desigualdades lineares
- usa um DAG: nós correspondem a variáveis, cada arco é etiquetado por um intervalo admissível para a variável no nó de onde parte, ou por desigualdades lineares
- ordem das variáveis é fixa: cada caminho desde a raiz até a uma folha deve visitar cada variável uma vez, pela ordem em que ocorrem em ***Template***
- ***Template*** é um termo arbitrário *non-ground* Prolog
- ***Tuples*** é uma lista de termos da mesma forma que ***Template*** (não devem partilhar variáveis)
- ***Dag*** é uma lista de nós na forma ***node(ID,X,Children)***, onde ***X*** é uma variável do *template* e ***ID*** é um inteiro identificando o nó; o primeiro nó da lista é a raiz
  - nó interno: ***Children*** é uma lista de termos ***(Min..Max)-ID2*** (ou *(Min..Max)-SideConstraints-ID2*), onde ***ID2*** identifica um nó filho
  - nó folha: ***Children*** é uma lista de termos ***(Min..Max)*** (ou *(Min..Max)-SideConstraints*)

Restrições Combinatórias

Case

- Exemplo:




```
element(X, [1,1,1,1,2,2,2,2], Y),
element(X, [10,10,20,20,10,10,30,30], Z)
```

```
elts(X, Y, Z) :-
    case(f(A,B,C), [f(X,Y,Z)],
         [node(0, A,[(1..2)-1,(3..4)-2,(5..6)-3,(7..8)-4]),
          node(1, B,[(1..1)-5]),
          node(2, B,[(1..1)-6]),
          node(3, B,[(2..2)-5]),
          node(4, B,[(2..2)-7]),
          node(5, C,[(10..10)]),
          node(6, C,[(20..20)]),
          node(7, C,[(30..30)])]).
```

```
| ?- elts(X, Y, Z).
X in 1..8,
Y in 1..2,
Z in {10}\/{20}\/{30}

| ?- elts(X, Y, Z), Z #>= 15.
X in(3..4)\/(7..8),
Y in 1..2,
Z in {20}\/{30}

| ?- elts(X, Y, Z), Y = 1.
Y = 1,
X in 1..4,
Z in {10}\/{20}
```

Restrições Combinatórias
Circuit

circuit(+Succ)
circuit(+Succ,+Pred)

- ***Succ*** é uma lista de comprimento *n* de variáveis de domínio ou inteiros
- o *i*-ésimo elemento de ***Succ*** (***Pred***) é o sucessor (predecessor) de *i* no grafo
- verdadeiro se os valores formam um circuito Hamiltoniano
  - nós estão numerados de *1* a *n*, o circuito começa no nó *1*, visita cada um dos nós e regressa à origem

- Exemplos:

```
| ?- length(L,5), domain(L,1,5), circuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 2..5, _B in {1}\/(3..5), _C in (1..2)\/(4..5), _D in (1..3)\/{5}, _E in 1..4 ?
yes
| ?- length(L,5), domain(L,1,5), circuit(L), labeling([],L).
L = [2,3,4,5,1] ? ;
L = [2,3,5,1,4] ? ;
…
```

Cumulative

cumulative(+Tasks)
cumulative(+Tasks,+Options)

- restringir *n* tarefas de forma que o consumo de recursos não exceda um limite em qualquer altura
- ***Tasks*** é uma lista de termos da forma ***task(Oi,Di,Ei,Hi,Ti)***
  - *Oi* = start time, *Di* = duração (não negativa), *Ei* = end time, *Hi* = consumo de recursos (não negativo), *Ti* = identificador da tarefa
  - todos os campos são variáveis de domínio ou inteiros
- a restrição verifica-se se para todas as tarefas *Oi+Di=Ei* e em todos os instantes *H1+H2+…+Hn =< L* (limite de recursos, 1 por defeito)
  - *Hi* é contabilizado apenas nos instantes entre *Oi* e *Ei*, senão é 0
- ***Options*** é uma lista de:
  - ***limit(L)***: limite de recursos
  - ***precedences(Ps)***: precedências entre tarefas, ***Ps*** é uma lista de termos na forma *Ti-Tj #= Dij*, com *Oi-Oj = Dij*
  - ***global(Boolean)***: se *true*, utiliza um algoritmo mais custoso para obter maior poda dos intervalos

Restrições Combinatórias

Cumulative

- Exemplo:
  - Escalonamento de tarefas:

Tarefa	Duração	Recursos
T1	16	2
T2	6	9
T3	13	3
T4	7	7
T5	5	10
T6	18	1
T7	4	11

- Limite de recursos = 13

```
schedule(Ss, End) :-
        Ss = [S1,S2,S3,S4,S5,S6,S7],
        Es = [E1,E2,E3,E4,E5,E6,E7],
        Tasks = [
                task(S1, 16, E1,  2,  1),
                task(S2,  6, E2,  9,  2),
                task(S3, 13, E3,  3,  3),
                task(S4,  7, E4,  7,  4),
                task(S5,  5, E5, 10,  5),
                task(S6, 18, E6,  1,  6),
                task(S7,  4, E7, 11,  7)
                ],
        domain(Ss, 1, 30),
        maximum(End, Es),
        cumulative(Tasks, [limit(13)]),
        labeling([minimize(End)], Ss).
```

Restrições Combinatórias

Cumulatives

cumulatives(+Tasks,+Machines)

cumulatives(+Tasks,+Machines,+Options)

- restringir *n* tarefas a serem realizadas no tempo em *m* máquinas, onde cada máquina tem um limite de recursos (mínimo ou máximo)
- ***Tasks*** é uma lista de termos da forma ***task(Oi,Di,Ei,Hi,Mi)***
  - *Oi* = start time, *Di* = duração (não negativa), *Ei* = end time, *Hi* = consumo de recursos (se positivo) ou produção de recursos (se negativo), *Mi* = identificador da máquina
  - todos os campos são variáveis de domínio ou inteiros
- máquina representada por termo ***machine(Mj,Lj)***
  - *Mj* = identificador, *Lj* = limite de recursos da máquina (ambos inteiros)
- a restrição verifica-se se para todas as tarefas *Oi+Di=Ei* e em todas as máquinas e instantes *H1m+H2m+…+Hnm >= Lm* (se *lower bound*), ou *H1m+H2m+…+Hnm =< Lm* (se *upper bound*)
- ***Options*** é uma lista de:
  - ***bound(B)***: tipo de limites, *lower* (valor por defeito) ou *upper*
  - ***prune(P)***, ***generalization(Boolean)***, ***task_intervals(Boolean)***

Cumulatives

- Exemplo:
  - Escalonamento de tarefas:

Tarefa	Duração	Recursos	Máquina
T1	16	2	1
T2	6	9	2
T3	13	3	1
T4	7	7	2
T5	5	10	1
T6	18	1	2
T7	4	11	1

- Limite de recursos M1 = 12
- Limite de recursos M2 = 10

```
schedule(Ss, End) :-
        Ss = [S1,S2,S3,S4,S5,S6,S7],
        Es = [E1,E2,E3,E4,E5,E6,E7],
        Tasks = [
                task(S1, 16, E1,  2,  1),
                task(S2,  6, E2,  9,  2),
                task(S3, 13, E3,  3,  1),
                task(S4,  7, E4,  7,  2),
                task(S5,  5, E5, 10,  1),
                task(S6, 18, E6,  1,  2),
                task(S7,  4, E7, 11,  1)
                ],
        Machines = [machine(1,12), machine(2,10)],
        domain(Ss, 1, 30),
        maximum(End, Es),
        cumulatives(Tasks, Machines, [bound(upper)]),
        labeling([minimize(End)], Ss).
```

Restrições Combinatórias

Disjoint

disjoint1(+Lines) / ***disjoint1(+Lines,+Options)***

disjoint2(+Rectangles) / ***disjoint2(+Rectangles,+Options)***

- conjunto de linhas ou retângulos que não se devem sobrepor
- ***Lines*** é uma lista de termos *F(Sj,Dj)* ou *F(Sj,Dj,Tj)*, *Sj* e *Dj* são variáveis de domínio ou inteiros (origem e comprimento da linha *j*), *F* é um qualquer functor, e *Tj* é um termo atómico opcional (default 0) denotando o tipo de linha
- ***Rectangles*** é uma lista de termos *F(Xj,Lj,Yj,Hj)* ou *F(Xj,Lj,Yj,Hj,Tj)*, …

- Exemplo:

```
| ?- domain([Ai,Bi,Ci],1,10),
     disjoint1([f(Ai,5),f(Bi,7),f(Ci,3)]), Ai#<Ci.
Ai in 1..5,
Bi in 9..10,
Ci in 2..7
```

```
| ?- domain([Ai,Bi,Ci],1,10),
     disjoint1([f(Ai,5),f(Bi,7),f(Ci,3)]), Ai#<Ci,
     labeling([],[Ai,Bi,Ci]).
Ai = 1, Bi = 9, Ci = 6 ? ;
Ai = 1, Bi = 10, Ci = 6 ? ;
Ai = 1, Bi = 10, Ci = 7 ? ;
Ai = 2, Bi = 10, Ci = 7 ? ;
no
```

Restrições Combinatórias

Geost

geost(+Objects,+Shapes)

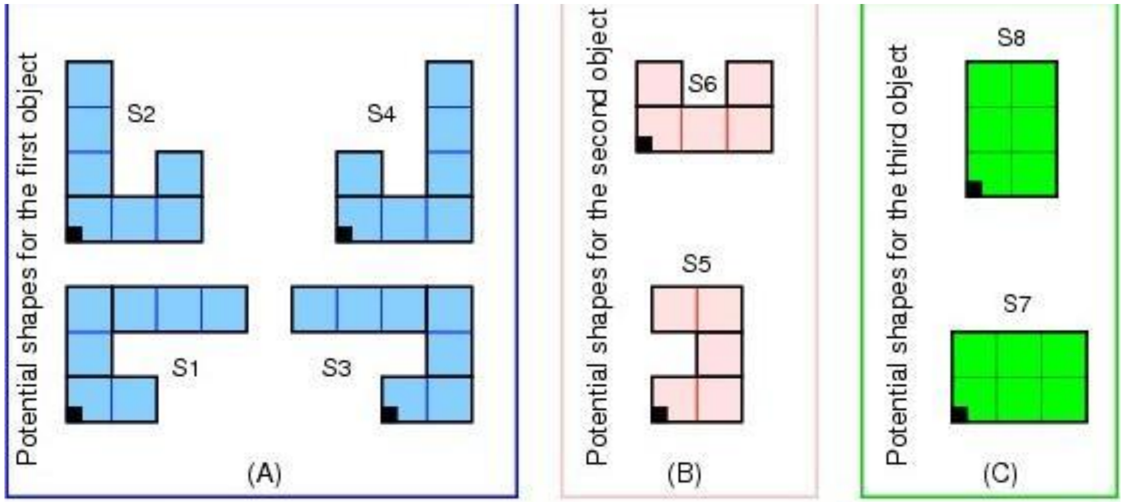
geost(+Objects,+Shapes,+Options)

geost(+Objects,+Shapes,+Options,+Rules)

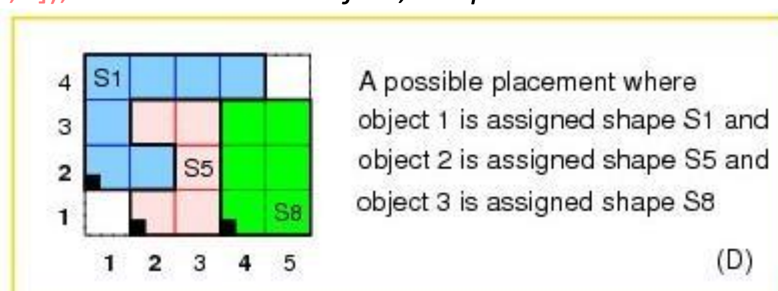
- restringe a localização no espaço de objetos ***Objects*** multidimensionais não sobrepostos, cada um dos quais tendo uma forma de entre um conjunto ***Shapes***
- ***Objects*** é uma lista de termos *object(Oid,Sid,Origin)*, onde *Oid* identifica o objeto, *Sid* é um inteiro ou variável de domínio que identifica a forma do objeto e *Origin* é uma lista de inteiros ou variáveis de domínio indicando as coordenadas de origem do objeto
- ***Shapes*** é uma lista de termos *sbox(Sid,Offset,Size)*, sendo cada forma definida pelos termos *sbox/3* com o mesmo *Sid*

Geost

- Exemplo:




```
| ?- domain([X1,X2,X3,Y1,Y2,Y3],1,4), S1 in 1..4, S2 in 5..6, S3 in 7..8,
   geost(    [ object(1,S1,[X1,Y1]), object(2,S2,[X2,Y2]), object(3,S3,[X3,Y3]) ],
             [ sbox(1,[0,0],[2,1]), sbox(1,[0,1],[1,2]), sbox(1,[1,2],[3,1]),      % first object, shape S1
               sbox(2,[0,0],[3,1]), sbox(2,[0,1],[1,3]), sbox(2,[2,1],[1,1]),      % first object, shape S2
               sbox(3,[0,0],[2,1]), sbox(3,[1,1],[1,2]), sbox(3,[2,2],[3,1]),      % first object, shape S3
               sbox(4,[0,0],[3,1]), sbox(4,[0,1],[1,1]), sbox(4,[2,1],[1,3]),      % first object, shape S4
               sbox(5,[0,0],[2,1]), sbox(5,[1,1],[1,1]), sbox(5,[0,2],[2,1]),      % second object, shape S5
               sbox(6,[0,0],[3,1]), sbox(6,[0,1],[1,1]), sbox(6,[2,1],[1,1]),      % second object, shape S6
               sbox(7,[0,0],[3,2]),          % third object, shape S7
               sbox(8,[0,0],[2,3])           % third object, shape S8
             ]),
   labeling([],[X1,X2,X3,Y1,Y2,Y3]).
```



Automaton

automaton(Signature,SourcesSinks,Arcs)

automaton(Sequence,Template,Signature,SourcesSinks,Arcs,Counters,Initial,Final)

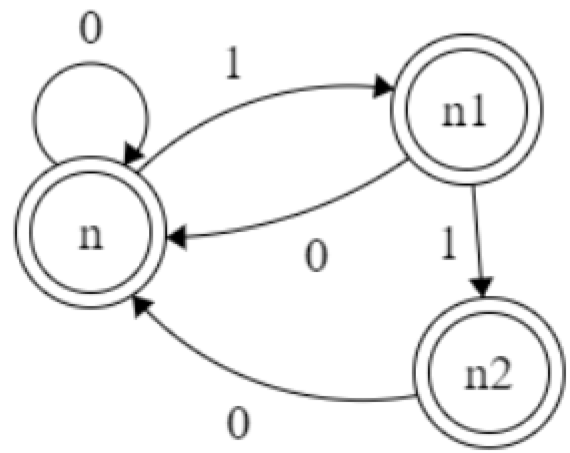
automaton(Sequence,Template,Signature,SourcesSinks,Arcs,Counters,Initial,Final,Options)

- forma geral de definir qualquer restrição envolvendo sequências que podem ser verificadas por um autómato finito determinístico ou não determinístico, evetualmente com operações de contagem nos seus arcos
- se não forem usados contadores, mantém consistência de domínios
- ***Signature*** é uma sequência de inteiros ou variáveis de domínio, com base na qual serão efetuadas as transições no autómato
- ***SourcesSinks*** é uma lista de elementos da forma *source(node)* e *sink(node)*, identificando os nós iniciais e de aceitação do autómato, respetivamente
- ***Arcs*** é uma lista de elementos da forma *arc(node, integer, node)* ou *arc(node, integer, node, exprs)*, identificando as transições possíveis entre nós e eventualmente operações sobre variáveis em *Counters*
- ***Counters***, ***Initial*** e ***Final*** são listas de igual tamanho identificando variáveis contadores, seus valores iniciais e finais, respetivamente

Restrições Combinatórias

Automaton

```
at_most_two_consecutive_ones(Vars) :-
    automaton(Vars,
        [ source(n),sink(n),sink(n1),sink(n2) ],
        [ arc(n, 0, n),
          arc(n, 1, n1),
          arc(n1, 1, n2),
          arc(n1, 0, n),
          %arc(n2, 1, false),
          arc(n2, 0, n) ]).
```

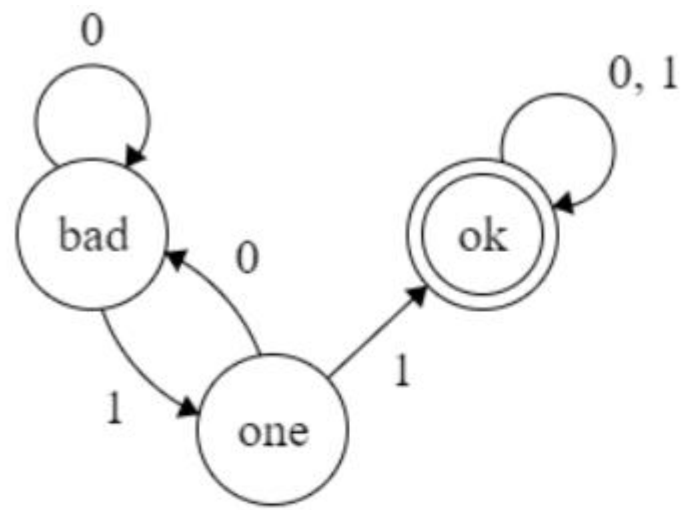



```
| ?- at_most_two_consecutive_ones([0,0,0,1,1,1]).
no
| ?- at_most_two_consecutive_ones([0,1,1,0,1,1]).
yes
| ?- at_most_two_consecutive_ones([0,1,1,0,1,0]).
yes
```

```
| ?- length(L,3), at_most_two_consecutive_ones(L).
L = [_A,_B,_C], _A in 0..1, _B in 0..1, _C in 0..1

| ?- length(L,3), at_most_two_consecutive_ones(L),
     L=[1|_], labeling([],L).
L = [1,0,0] ? ;
L = [1,0,1] ? ;
L = [1,1,0] ? ;
no
```

Automaton

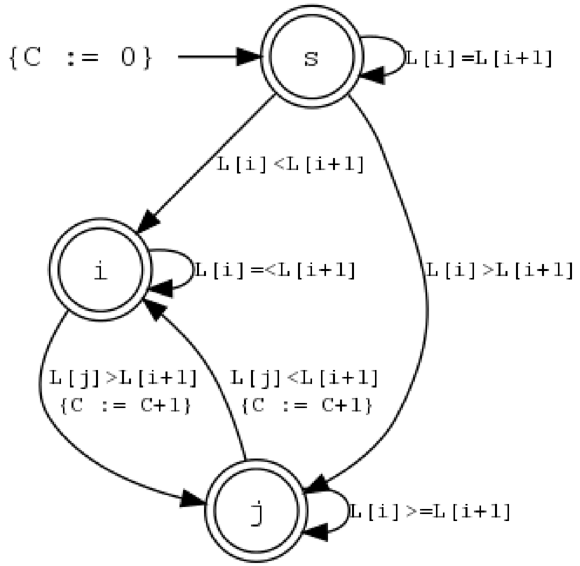
```
at_least_two_consecutive_ones(Vars, N) :-
    length(Vars, N),
    %domain(Vars, 0, 1),

    automaton(Vars,
            [ source(bad), sink(ok) ],
            [ arc(bad, 0, bad),  arc(bad, 1, one),
              arc(one, 0, bad),  arc(one, 1, ok),
              arc(ok, 0, ok), arc(ok, 1, ok)]),

    labeling([], Vars).
```




```
| ?- at_least_two_consecutive_ones(L,3).
L = [0,1,1] ? ;
L = [1,1,0] ? ;
L = [1,1,1] ? ;
no
```

Automaton

```
inflexion(N, Vars) :-
    inflexion_signature(Vars, Sign),
    automaton(Sign, _, Sign,
        [ source(s), sink(i), sink(j), sink(s) ],
        [ arc(s,1,s), arc(s,2,i), arc(s,0,j),
          arc(i,1,i), arc(i,2,i), arc(i,0,j,[C+1]),
          arc(j,1,j), arc(j,0,j), arc(j,2,i,[C+1])],
        [C],[0],[N]).

inflexion_signature([], []).
inflexion_signature([_], []) :- !.
inflexion_signature([X,Y|Ys], [S|Ss]) :-
    S in 0..2,
    X #> Y #<=> S #= 0,
    X #= Y #<=> S #= 1,
    X #< Y #<=> S #= 2,
    inflexion_signature([Y|Ys], Ss).
```




```
| ?- inflexion(N, [1,1,4,8,8,2,7,1]).
N = 3

| ?- length(L,4), domain(L,0,1), inflexion(2,L), labeling([],L).
L = [0,1,0,1] ? ;
L = [1,0,1,0] ? ;
no
```

PLR no SICStus Prolog

PREDICADOS DE ENUMERAÇÃO

Pesquisa

- Usualmente os *solvers* de restrições em domínios finitos não são completos, ou seja, não garantem que o conjunto de restrições tem solução
- É necessário pesquisa (enumeração) para verificar a "satisfabilidade" e conseguir soluções concretas

- Predicados para efetuar a pesquisa:

indomain(?X)

- ***X*** é uma variável de domínio ou um inteiro
- atribui, por *backtracking*, valores admissíveis a ***X***, por ordem ascendente

labeling(:Options,+Variables)

- ***Options*** é uma lista de opções de pesquisa e ***Variables*** é uma lista de variáveis de domínio (com domínios finitos) ou inteiros
- verdadeiro se pode ser encontrada uma atribuição de valores às variáveis que satisfaça todas as restrições

Otimização

- Os predicados de otimização permitem a busca de soluções ótimas (minimização/maximização de um custo/lucro):

  minimize(:Goal,?X) / minimize(:Goal,?X,+Options)

  maximize(:Goal,?X) / maximize(:Goal,?X,+Options)

  - utilizam um algoritmo *branch-and-bound* para procurar uma atribuição que minimize/maximize a variável de domínio ***X***
  - ***Goal*** deve ser um objetivo Prolog que restrinja ***X*** a ficar com um valor, podendo ser um objetivo *labeling/2*
  - o algoritmo chama ***Goal*** repetidamente com uma *upper* (*lower*) *bound* em ***X*** progressivamente mais restringida até a prova de otimalidade ser obtida (o que por vezes é demasiado demorado…)
  - ***Options*** é uma lista contendo um de:
    - ***best*** (opção por defeito): apenas solução ótima interessa
    - ***all***: enumera soluções cada vez melhores

Opções de Pesquisa

- O argumento ***Options*** de *labeling/2* controla a ordem de seleção de variáveis e valores, o tipo de solução a encontrar e a execução da pesquisa
  - Ordenação de variáveis:
    - ***leftmost***, ***min***, ***max***, ***first_fail***, ***anti_first_fail***, ***occurrence***, ***ffc***, ***max_regret***, ***variable(Sel)***
  - Forma de seleção de valores:
    - ***step***, ***enum***, ***bisect***, ***median***, ***middle***, ***value(Enum)***
  - Ordenação de valores:
    - ***up***, ***down***
  - Soluções a encontrar:
    - ***satisfy***, ***minimize(X)***, ***maximize(X)***, ***best***, ***all***
  - Esquema de pesquisa:
    - ***bab***, ***restart***
  - Assunções:
    - ***assumptions(K)***: ***K*** é unificado com o número de escolhas feitas
  - Discrepância:
    - ***discrepancy(D)***: no caminho para a solução há no máximo ***D*** pontos de escolha nos quais houve retrocesso
  - Tempo limite para a pesquisa:
    - ***time_out(Time,Flag)***: se o tempo limite for alcançado é obtida a melhor solução encontrada até então

Ordenação de Variáveis

- Como selecionar a próxima variável?
  - ***leftmost*** (opção por defeito): mais à esquerda
  - ***min***: menor *lower bound*
  - ***max***: maior *upper bound*
  - ***first_fail***: mais à esquerda com o menor domínio
  - ***anti_first_fail***: mais à esquerda com o maior domínio
  - ***occurrence***: mais restrições suspensas, mais à esquerda
  - ***ffc***: menor domínio, mais restrições suspensas (*most constrained heuristic*)
  - ***max_regret***: maior diferença entre os dois primeiros elementos do domínio, mais à esquerda
  - ***variable(Sel)***:
    - ***Sel*** é um predicado para selecionar a próxima variável: ***Sel(Vars,Selected,Rest)***
    - deve suceder deterministicamente unificando ***Selected*** e ***Rest*** com a variável selecionada e a lista remanescente

Seleção de Valores

- Como selecionar valores para uma variável?
  - ***step*** (opção por defeito): escolha binária entre ***X #= B*** e ***X #\= B***, onde ***B*** é a *lower* ou *upper bound* de ***X***
  - ***enum***: escolha múltipla para ***X*** correspondendo aos valores do seu domínio
  - ***bisect***: escolha binária entre ***X #=< M*** e ***X #> M***, onde ***M*** é o ponto médio do domínio de ***X***
  - ***median*** / ***middle***: escolha binária entre ***X #= M*** e ***X #\= M***, onde ***M*** é a mediana/média do domínio de ***X***
  - ***value(Enum)***:
    - ***Enum*** é um predicado que deve reduzir o domínio de ***X***: ***Enum(X,Rest,BB0,BB)***
      - ***Rest*** é a lista de variáveis que necessitam de "labeling" exceto ***X***
    - ***Enum*** deve suceder de forma não-determinística, dando por *backtracking* outras formas de redução de domínio
      - deve chamar o predicado auxiliar ***first_bound(BB0,BB)*** na sua primeira solução e ***later_bound(BB0,BB)*** em qualquer solução alternativa

Ordenação de Valores

- Como selecionar um valor para uma variável?
  (sem utilidade com a opção *value(Enum)*)

  - ***up*** (opção por defeito): domínio explorado por ordem ascendente

  - ***down***: domínio explorado por ordem descendente

Soluções a Encontrar

- Estas opções indicam se o problema é de satisfação (qualquer solução interessa) ou de otimização (apenas a melhor solução):
  - ***satisfy*** (opção por defeito): todas as soluções são enumeradas por *backtracking*
  - ***minimize(X)*** / ***maximize(X)***: pretende-se a solução que minimiza/maximiza a variável de domínio ***X***
    - o mecanismo de *labeling* deve restringir ***X*** a ficar com um valor para todas as atribuições das variáveis
    - é útil combinar esta opção com *time_out/2*, *best* ou *all*
    - ***best*** (opção por defeito): obtém a solução ótima
    - ***all***: obtém, por *backtracking*, soluções cada vez melhores

Opções de Pesquisa – Exemplos

- Enumerar soluções com ordenação de variáveis estática:

  | ?- constraints(Variables),

  labeling([], Variables).

  - **[]** é o mesmo que: **[leftmost,step,up,satisfy]**

- Minimizar uma função de custo, obter apenas a melhor solução, ordenação dinâmica de variáveis usando o *first-fail principle*, e divisão de domínio explorando a parte superior dos domínios primeiro:

  | ?- constraints(Variables, Cost),

  labeling([ff,bisect,down,minimize(Cost)], Variables).

PLR no SICStus Prolog

PREDICADOS DE ESTATÍSTICAS

Predicados de Estatísticas

- Estatísticas de execução específicas do *solver* CLP(FD)
  - ***fd_statistics(?Key,?Value)***: para cada chave possível ***Key***, ***Value*** é unificado com o valor atual de um contador:
    - ***resumptions***: número de vezes que uma restrição foi reatada
    - ***entailments***: número de vezes que um *(dis)entailment* foi detetado
    - ***prunings***. número de vezes que um domínio foi reduzido
    - ***backtracks***: número de vezes que foi encontrada uma contradição por um domínio ter ficado vazio ou uma restrição global ter falhado
    - ***constraints***: número de restrições criadas
  - ***fd_statistics***: mostra um resumo das estatísticas acima
- Outras estatísticas (e.g. tempo e consumo de memória):
  - *statistics/2*