U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Programação em Lógica com Restrições no SICStus Prolog

Novembro de 2020

SICStus Prolog User's Manual (Release 4.6)

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

Parcialmente baseado em slides anteriores de
Henrique Lopes Cardoso (hlc@fe.up.pt),
Luís Paulo Reis (lpreis@fe.up.pt),
outros autores e no manual
do SCIStus Prolog
(v. 4.6)

Daniel Castro Silva
dcs@fe.up.pt

30/11/2020

Conteúdo

1. Domínios de Restrições Disponíveis
2. Interface do solver CLP(FD)
   - Estrutura de um Programa em PLR
   - Declaração de Domínios
   - Colocação de Restrições
   - Restrições Materializadas
3. Restrições Disponíveis
4. Predicados de Enumeração
   - Pesquisa e otimização
5. Predicados de Estatísticas

PLR no SICStus Prolog

1. DOMÍNIOS DE RESTRIÇÕES DISPONÍVEIS

Domínios Booleanos e Reais

- Booleanos:
  - Esquema clp(B)
  - *use_module(library(clpb)).*
    - Secção 10.9 do manual do SICStus

- Reais e Racionais
  - Esquema clp(Q,R)
  - *use_module(library(clpq)).* *use_module(library(clpr)).*
    - Secção 10.11 do manual do SICStus

- 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, ...
- 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 refletido numa variável binária (0/1) – materialização (ou reificação)
  - Podem-se adicionar novas restrições primitivas escrevendo indexicais
  - Podem ser escritas novas restrições globais em Prolog

PLR no SICStus Prolog

2. 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
- 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 inteiros pequenos 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
    - Possível usar o predicado *prolog_flag/2* para obter estes valores

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 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

Estrutura de um Programa em PLR

- Um programa em PLR estrutura-se nas três etapas seguintes:
  - Declaração de variáveis e seus domínios
  - Declaração de restrições sobre as variáveis
  - Pesquisa de uma solução

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

example:-
        A in 1..7,
        domain( [B, C], 1, 10),
        A + B + C #= A * B * C,
        A #> B,
        labeling( [], [A, B, C] ).
```

variáveis e domínios

restrições

pesquisa de solução

```
| ?- example.
A = 2,
B = 1,
C = 3 ?
```

Estrutura de um Programa em PLR

- Ordem destas etapas é importante
  - Se invertermos a ordem, colocando primeiro a pesquisa de solução e depois as restrições, resulta no mecanismo *Generate&Test* tradicional, muito menos eficiente

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

badExample:-
        A in 1..7,
        domain( [B, C], 1, 10),
        labeling( [], [A, B, C] ),
        write('.'),
        A + B + C #= A * B * C,
        A #> B.
```

```
| ?- badExample.
.....................................
.....................................
.....................
A = 2,
B = 1,
C = 3 ?
```

Domínios das Variáveis

- Uma variável pode ter o seu domínio declarado usando ***in/2*** e um intervalo (*ConstantRange*):
  - *NotaPLOG in 16..20*

- A definição de *ConstantRange* permite a declaração de domínios mais complexos
  - *VarA in (2..8) \/ (15..20)*
  - *VarB in {4, 8, 15, 16, 23, 42}*

ConstantSet ::= `{integer,…,integer}`

ConstantRange ::= *ConstantSet*
| *Constant .. Constant*
| *ConstantRange* `/\` *ConstantRange*
| *ConstantRange* `\/` *ConstantRange*
| `\` *ConstantRange*

Domínios das Variáveis

- Pode ainda ser usado ***in_set/2*** para declaração de domínio de uma variável
  - O segundo argumento de ***in_set/2*** é um *Finite Domain Set*, que pode ser obtido a partir de uma lista usando o predicado ***list_to_fdset(+List, -FD_Set)***.
    - Ver secção 10.10.9.3 para operações sobre *FD Sets*

```
Numbers = [4, 8, 15, 16, 23, 42],
list_to_fdset(Numbers, FDS_Numbers),
Var in_set FDS_Numbers.
```

Domínios das Variáveis

- Para declarar um mesmo domínio simples para uma lista de variáveis pode ser usado o predicado ***domain(+List_of_Variables, +Min, +Max)***:
  - *domain([A, B, C], 5, 12)*

- Outras restrições limitam domínios das variáveis envolvidas
  - *A #> 8*
  - *B + C #< 12*
  - *A + B + C #= 20*

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 existência de uma resposta mostra a existência de domínios válidos para as variáveis
  - Não são visualizadas as restrições associadas a cada variável

Colocação de Restrições

- Ao colocar uma restrição, é chamado o mecanismo de propagação, que limita os domínios das variáveis
  - Este mecanismo pode ser computacionalmente pesado em alguns casos
- É possível colocar um conjunto de restrições de uma vez (em lote), suspendendo o mecanismo de propagação até que estas restrições tenham sido todas colocadas
  - ***fd_batch(+Constraints)***
    - Onde *Contraints* é uma lista de restrições a colocar
  - *domain([A,B,C], 5, 12),*
    fd_batch([A #> 8, B + C #< 12, A + B + C #= 20])

Restrições Materializadas (*Reified*)

- Por vezes é útil fazer refletir 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

3. RESTRIÇÕES DISPONÍVEIS

Restrições Disponíveis

- Restrições Aritméticas
- Restrições de Pertença
- Restrições Proposicionais
- Restrições Combinatórias
  - Aritmético-lógicas
  - Extensão
  - Grafo
  - Escalonamento
  - Posicionamento
  - Sequenciamento

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 lineares mantêm consistência de intervalos
  - Restrições Aritméticas podem ser materializadas
  - Exemplo:

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

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 aproximadamente 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
  - Exemplos:

```
| ?- 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
```

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 e/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
  - ***Options*** é uma lista de opções
    - ***among(Least, Most, Range)*** indica que no mínimo ***Least*** e no máximo ***Most*** elementos de ***Xs*** têm valores no intervalo (*ConstantRange*) indicado em ***Range***
    - ***consistency(Cons)*** indica que nível de consistência deve ser usado pela restrição. ***Cons*** pode tomar os valores ***domain*** (deve manter consistência de domínios; útil apenas se ***RelOp*** for *#=* e todas as variáveis de domínio devem ter domínios finitos); ***bounds*** ou ***value*** (opção por omissão), indica consistência de limites.

Produto Escalar

- ***scalar_product_reif(+Coeffs, +Xs, +RelOp, ?Value, ?Reif)***
- ***scalar_product_reif(+Coeffs, +Xs, +RelOp, ?Value, ?Reif, +Options)***
  - Versão com reificação de scalar_product/4 e /5.
  - Equivalente a materializar a restrição anterior
  - 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
```

Mínimo/Máximo

- ***minimum(?Value, +Xs)***
- ***maximum(?Value, +Xs)***
  - ***Xs*** é uma lista de inteiros e/ou variáveis de domínio
  - ***Value*** é um inteiro ou variável de domínio
  - Verdadeira se ***Value*** é o valor mínimo (ou máximo) de ***Xs***
  - Corresponde a *min_member/2* (*max_member/2*) da *library(lists)*
  - Não podem ser materializadas
  - Exemplos:

```
| ?- 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
```

Mínimo/Máximo

- ***minimum_arg(+Xs, ?Index)***
- ***maximum_arg(+Xs, ?Index)***
  - ***Xs*** é uma lista de inteiros e/ou variáveis de domínio
  - ***Index*** é um inteiro ou variável de domínio
  - Verdadeira se ***Index*** é o índice do valor mínimo (ou máximo) de ***Xs***
  - Se o valor se repetir, ***Index*** será o índice da primeira ocorrência
  - Não podem ser materializadas
  - Exemplos:

```
| ?- minimum_arg( [3,1,2,5], A).
A = 2

| ?- maximum_arg( [3,1,2,5], B).
B = 4
```

```
| ?- domain([A,B,C], 1, 5),
     sum([A,B], #=, 10),
     minimum_arg( [A,B,C], X).
A = 5,
B = 5,
C in 1..5
X in {1} \/ {3}
```

Restrições de Pertença (*Membership*)

- Predicados de definição de domínios das variáveis
  - ***domain(+Vars, +Min, +Max)***
    - Verdadeira se todos os elementos de ***Vars*** estão no intervalo ***Min..Max***
  - ***?X in +Range***
    - Verdadeira se ***X*** é um elemento do intervalo ***Range***
  - ***?X in_set +FDSet***
    - Verdadeira se ***X*** é um elemento do conjunto ***FDSet***
  - *in/2* e *in_set/2* mantêm consistência do domínio e são materializáveis
  - Exemplos:

```
| ?- 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 (NOT)
:P #/\ :Q	verdadeira se as restrições P e Q são ambas verdadeiras (AND)
:P #\ :Q	verdadeira se exatamente uma das restrições P e Q é verdadeira (XOR)
:P #\/ :Q	verdadeira se pelo menos uma das restrições P e Q é verdadeira (OR)
:P #=> :Q :Q #<= :P	verdadeira se a restrição Q é verdadeira ou se a restrição P é falsa (implicação)
:P #<=> :Q	verdadeira se P e Q são ambas verdadeiras ou ambas falsas (equivalência)

- Note-se que o esquema de materialização é um caso particular da restrição proposicional de equivalência

Restrições Combinatórias

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

Arithmetic-Logical

- *smt/1 (deprecated)*
- *count/4 (deprecated)*
- *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]*

Scheduling

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

Automata

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

Graph

- *circuit/[1,2]*
- *subcircuit/[1,2]*

Placement

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

Count

(*deprecated*, ver *global_cardinality*)

- ***count(+Val, +List, +RelOp, ?Count)***
  - Restringe o número de ocorrências do valor ***Val*** na lista ***List*** a ter a relação ***RelOp*** com o valor ***Count***
  - ***Val*** é um inteiro, ***List*** uma lista de inteiros ou variáveis de domínio, ***Count*** um inteiro ou variável de domínio, e ***RelOp*** um operador relacional
  - 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
```

Global Cardinality

- ***global_cardinality(+Xs, +Vals)***
- ***global_cardinality(+Xs, +Vals, +Options)***
  - Restringe o número de ocorrências de cada valor numa lista de variáveis
  - ***Xs*** é uma lista de inteiros e/ou variáveis de domínio; ***Vals*** é uma lista de termos ***K-V***, onde ***K*** é um inteiro único e ***V*** é um inteiro ou variável de domínio
  - 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
  - ***Options*** é lista de opções (para controlar funcionamento) (ver documentação)
  - 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
```

Nvalue

- ***nvalue(?N, +Variables)***
  - Restringe a lista de variáveis ***Variables*** de forma a que existam exatamente ***N*** valores distintos
  - ***Variables*** é uma lista de inteiros e/ou variáveis de domínio com limites finitos e ***N*** é um inteiro ou variável de domínio
  - 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
```

All Different / All Distinct

- ***all_different(+Variables)*** / ***all_different(+Variables, +Options)***
- ***all_distinct(+Variables)*** / ***all_distinct(+Variables, +Options)***
  - Verdadeira quando todos os valores da lista ***Variables*** são distintos
    - Equivalente a uma restrição ***#\=*** para cada par de variáveis
  - ***Variables*** é uma lista de inteiros e/ou variáveis de domínio
  - ***Options*** é uma lista de zero ou mais opções (ver documentação):
    - ***L #= R*** – restrição adicional com uma expressão
    - ***on(On)*** - quando acordar a restrição
    - ***consistency(Cons)*** - que algoritmo utilizar
  - Exemplos:

```
| ?- 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
```

```
| ?- 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
```

All Different / Distinct Except 0

- ***all_different_except_0(+Variables)***
- ***all_distinct_except_0(+Variables)***
  - Verdadeira quando as variáveis da lista ***Variables*** têm valores distintos, com exceção de variáveis com o valor 0
  - ***Variables*** é uma lista de inteiros ou variáveis de domínio
  - Exemplos:

```
| ?- L = [A,B,1,D], domain(L, 0, 2),
     all_distinct(L).
no

| ?-  L = [A,B,1,D], domain(L, 0, 2),
     all_distinct_except_0(L).
A in {0} \/ {2},
B in {0} \/ {2},
D in {0} \/ {2}  ?
```

```
| ?- L = [A,B,C,D], domain(L, 0, 2),
     all_distinct_except_0(L), labeling([], L).
L = [0,0,0,0] ? ;
L = [0,0,0,1] ? ;
L = [0,0,0,2] ? ;
L = [0,0,1,0] ? ;
L = [0,0,1,2] ? ;
L = [0,0,2,0] ? ;
L = [0,0,2,1] ? ;
L = [0,1,0,0] ? ;
...
```

Symmetric All Different / Distinct

- ***symmetric_all_different (+Variables)***
- ***symmetric_all_distinct (+Variables)***
  - Verdadeira quando as variáveis da lista ***Variables*** têm valores distintos, e para todas as variáveis Xi=j sse Xj=i
  - ***Variables*** é uma lista de inteiros e/ou variáveis de domínio
  - Exemplos:

```
| ?- L = [A,B,C,D],
     symmetric_all_distinct(L), A #= 3.
A = 3,
C = 1,
B in {2} \/ {4},
D in {2} \/ {4} ?
```

```
| ?- L = [A,B,C],  symmetric_all_distinct(L),
     labeling([], L).
L = [1,2,3] ? ;
L = [1,3,2] ? ;
L = [2,1,3] ? ;
L = [3,2,1] ? ;
no
```

Assignment

- ***assignment(+Xs, +Ys)***
- ***assignment(+Xs, +Ys, +Options)***
  - ***Xs*** = [X1,...,Xn] e ***Ys*** = [Y1,...,Yn] são listas de comprimento n de variáveis de domínio e/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_distinct/2**
    - *circuit(Boolean)*: se *true*, *circuit(Xs,Ys)* tem que se verificar
    - *cost(Cost,Matrix)*: permite associar um custo à restrição
  - Exemplos:

```
| ?- assignment([4,1,5,2,3], Ys).
Ys = [2,4,5,1,3]
```

```
| ?-  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
```

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
```

Keysorting

- ***keysorting(+Xs, +Ys)***
- ***keysorting(+Xs, +Ys, +Options)***
  - Generalização de ***sorting/3*** mas ordenando tuplos de variáveis
  - Os tuplos são separados em chave e valor, sendo ordenados apenas pela chave (mantém ordem de tuplos com a mesma chave)
  - ***Xs*** e ***Ys*** são listas, com o mesmo tamanho n, de tuplos de variáveis; todos os tuplos (listas de variáveis) têm o mesmo tamanho m
  - ***Options*** é uma lista de opções:
    - ***keys(Keys)*** - ***Keys*** é o tamanho da chave (inteiro positivo; valor por omissão é 1)
    - ***permutation(Ps)*** - Ps é lista de variáveis (permutação de [1,n], tal que para cada i em [1,n], j em [1,m] : Ys[i,j] = Xs[Ps[i],j].)
  - Exemplo:

```
ln2(X):-
   length(X,2).
```

```
| ?-  _List = [[1,5], [6,5], [4,3], [7,9], [4,5], [7,8], [3,3]],
      length(_List, _Len), length(Sorted, _Len), maplist(ln2, Sorted),
      length(P, _Len), keysorting(_List, Sorted, [permutation(P)] ).
Sorted = [[1,5],[3,3],[4,3],[4,5],[6,5],[7,9],[7,8]]
P = [1,7,3,5,2,4,6] ? ;
no
```

Lex Chain

- ***lex_chain(+Vectors)***
- ***lex_chain(+Vectors, +Options)***
  - ***Vectors*** é uma lista de vetores (listas) de inteiros ou variáveis de domínio
  - A restrição verifica-se se ***Vectors*** está por ordem lexicográfica ascendente (na realidade, não descendente por omissão)
  - ***Options*** é uma lista de opções:
    - ***op(Op)*** - ***Op*** é ***#<=*** (omissão) ou ***#<*** (estritamente ascendente)
    - ***increasing*** - listas internas ordenadas de forma estritamente ascendente
    - ***among(Least, Most, Values)*** – entre ***Least*** e ***Most*** valores de cada ***Vector*** pertencem à lista ***Values***
  - Exemplo:

```
| ?- domain([A,B,C], 1, 2),
     lex_chain([ [A,B,C], [B,C,A], [C,B,A] ]),
     labeling([], [A,B,C]).
A = 1,B = 1,C = 1 ? ;     A = 1,B = 1,C = 2 ? ;
A = 1,B = 2,C = 2 ? ;     A = 2,B = 2,C = 2 ? ;
no
```

Element

- ***element(?X, +List, ?Y)***
  - ***X*** e ***Y*** são inteiros ou variáveis de domínio; ***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 uma só vez
  - Verdadeira se ***MapList*** contém um par ***X-R*** e ***Y*** está no intervalo indicado em ***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
```

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***; ***Options*** é lista de opções que permitem controlar ordem de variáveis usada internamente e estrutura de dados e algoritmo (ver doc.)
  - 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}
```

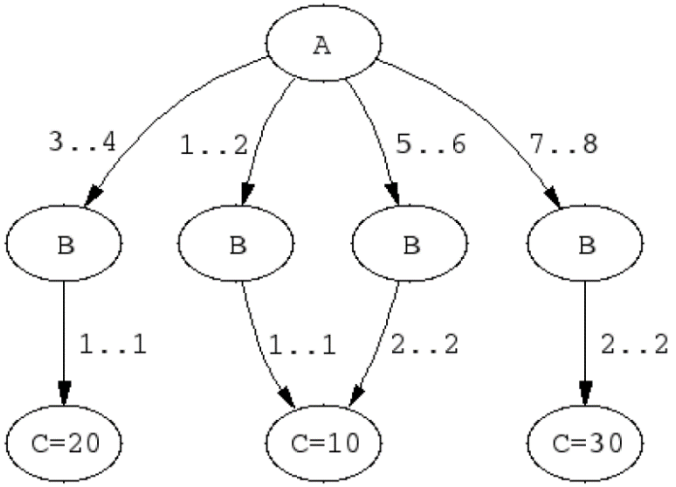
```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]),
     labeling([],[A,B]).
A = 1, B = 1 ? ;
A = 1, B = 2 ? ;
A = 2, B = 10 ? ;
A = 2, B = 20 ? ;
no
```

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*
  - ***Tuples*** é uma lista de termos da mesma forma que ***Template*** (não devem partilhar variáveis)
  - ***Dag*** é uma lista de termos 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*)

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}
```

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] ? ;
...
```

Subcircuit

- ***subcircuit(+Succ)***
- ***subcircuit(+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; ou i se o elemento não estiver incluído no sub-circuito
  - Verdadeiro se os valores incluídos formam no máximo um circuito Hamiltoniano
  - 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),
     subcircuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 1..5,
_B in 1..5,
_C in 1..5,
_D in 1..5,
_E in 1..5 ?
```

Cumulative

- ***cumulative(+Tasks)***
- ***cumulative(+Tasks, +Options)***
  - Restringe n tarefas de forma que o consumo de recursos não exceda um limite em qualquer altura
  - ***Tasks*** é uma lista de n 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 omissão)
    - ***Hi*** é contabilizado apenas nos instantes entre ***Oi*** e ***Ei***; senão é 0
  - ***Options*** é uma lista de opções:
    - ***limit(L)***: L é o limite de recursos a usar
    - ***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

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

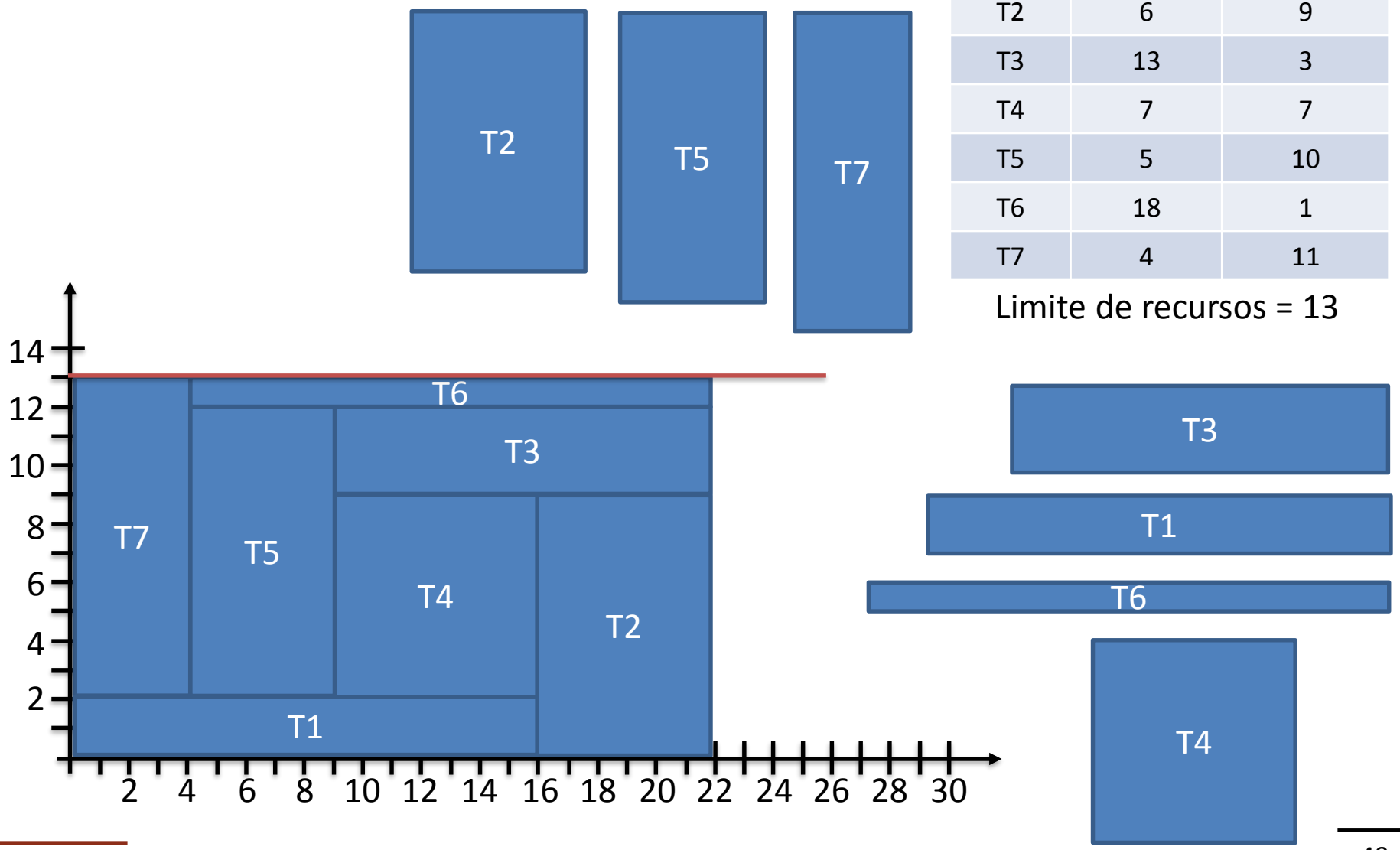
    - Limite de recursos = 13

```
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).
```

Cumulative

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



Cumulatives

- ***cumulatives(+Tasks, +Machines)***
- ***cumulatives(+Tasks, +Machines, +Options)***
  - Restringe ***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
  - ***Machines*** é lista de termos na forma ***machine(Mj, Lj)***
    - ***Mj*** = identificador, ***Lj*** = limite de recursos da máquina (inteiro ou variável com limites definidos)
  - 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 opções:
    - ***bound(B)*** - tipo de limite: ***lower*** (valor por omissão) ou ***upper***
    - ***prune(P)*** - ***all*** (valor por omissão) ou ***next***: indica nível de poda a efetuar
    - ***generalization(Boolean)***, ***task_intervals(Boolean)*** - se ***true*** é feito algum processamento extra

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).
```

Multi_Cumulative

- ***multi_cumulative(+Tasks, +Capacities)***
- ***multi_cumulative(+Tasks, +Capacities, +Options)***
  - Generalização da restrição ***cumulative*** permitindo que as tarefas consumam múltiplos recursos em simultâneo; estes podem ser de dois tipos:
    - *cumulative* - recursos tal como usados na restrição ***cumulative***
    - *colored* – cada tarefa especifica uma cor (codificada como um inteiro); número de cores em uso em cada momento não pode exceder determinado limite; cor 0 significa que a tarefa não usa nenhuma cor
  - ***Tasks*** é uma lista de termos da forma ***task(Oi, Di, Ei, Hsi, Ti)***
    - ***Oi*** = *start time*, ***Di*** = duração (não negativa), ***Ei*** = *end time*, ***Hsi*** = lista de consumos de recursos/cor utilizada, ***Ti*** = identificador da tarefa
    - ***Oi*** e ***Ei*** são variáveis de domínio; os restantes campos devem ser inteiros
  - ***Capacities*** é uma lista de termos no formato ***cumulative(Limit)*** ou ***colored(Limit)***
    - Tamanho da lista ***Capacities*** deve ser igual ao tamanho de todas as listas ***Hsi***
  - A restrição verifica-se se nenhum recurso excede o seu limite em nenhum momento
  - ***Options*** é uma lista de opções:
    - ***greedy(Flag)***: ***Flag*** é variável com domínio 0..1 indicando se deve ser usado modo *greedy*
    - ***precedences(Ps)***: precedências entre tarefas; ***Ps*** é uma lista de termos na forma ***Ti-Tj*** (***Ti*** e ***Tj*** são identificadores de tarefas) indicando que ***Ti*** deve terminar antes de ***Tj*** iniciar

Bin Packing

- ***bin_packing(+Items, +Bins)***
  - Atribui 'itens' de determinado tamanho a 'compartimentos' com determinada capacidade
  - ***Items*** é lista de termos no formato ***item(Bin, Size)***
    - ***Bin*** é o compartimento ao qual o item será alocado (variável de domínio); ***Size*** indica o tamanho do item (inteiro >=0)
  - ***Bins*** é lista de termos no formato ***bin(ID, Cap)***
    - ***ID*** é o identificador de cada compartimento (inteiro, todos diferentes); ***Cap*** indica a capacidade do compartimento (variável de domínio)
  - A restrição verifica-se se todos os itens são atribuídos a um compartimento existente e se o somatório do tamanho dos itens atribuídos a cada compartimento é igual à sua capacidade

Bin Packing

- Exemplo:
  - 6 objetos, 3 compartimentos

Item	Size
A	5
B	6
C	3
D	7
E	9
F	4

Bin	Cap
1	9
2	14
3	11

```
place(Vars) :-
    Vars = [A, B, C, D, E, F],
    Items = [ item(A, 5),
              item(B, 6),
              item(C, 3),
              item(D, 7),
              item(E, 9),
              item(F, 4)   ],
    Bins = [  bin(1, 9),
              bin(2, 14),
              bin(3, 11)   ],
    bin_packing(Items, Bins),
    labeling([], Vars).
```

```
| ?- place(Vars).
Vars = [2,1,1,3,2,3] ? ;
Vars = [2,2,2,3,1,3] ? ;
Vars = [3,3,2,2,1,2] ? ;
no
```

Disjoint

- ***disjoint1(+Lines)***
- ***disjoint1(+Lines, +Options)***
  - Restringe conjunto de linhas de forma a que não se sobreponham
    - Visão 1D do espaço (todas as linhas estão alinhadas)
  - ***Lines*** é uma lista de termos no formato ***F(Sj,Dj)*** ou ***F(Sj, Dj, Tj)***
    - ***Sj*** e ***Dj*** representam origem e tamanho da linha j (variáveis de domínio ou inteiros);
      F é um qualquer functor;
      Tj é um termo atómico opcional (0 por omissão) que indica o tipo de linha
  - Options é lista de opções
    - ***global(Boolean)*** - se ***true*** um algoritmo redundante é usado para atingir uma poda mais completa
    - ***wrap(Min, Max)*** - espaço visto como um círculo, onde os valores ***Min*** e ***Max*** (inteiros) coincidem; esta opção força valores de origem ao intervalo [Min, Max-1]
    - ***margin(T1, T2, D)*** - impõe uma distância mínima ***D*** entre o final de qualquer linha do tipo ***T1*** e o início de qualquer linha do tipo ***T2***; ***D*** deve ser inteiro positivo ou ***sup***: todas as linhas do tipo ***T2*** terminam antes de qualquer linha do tipo ***T1***

Disjoint

– Exemplo:

```
place(Starts) :-
    Starts = [A, B, C],
    domain(Starts, 1, 10),
    Lines = [
                line(A, 5),
                line(B, 7),
                line(C, 3)
                ],
    A #< C,
    disjoint1(Lines),
    labeling([], Starts).
```

```
Starts = [1,9,6] ? ;
Starts = [1,10,6] ? ;
Starts = [1,10,7] ? ;
Starts = [2,10,7] ? ;
no
```

```
place(Starts) :-
    Starts = [A, B, C, D],
    domain(Starts, 1, 12),
    Lines = [
                line(A, 4, r),
                line(B, 2, g),
                line(C, 3, r),
                line(D, 2, g)
                ],
    A #< B,
    disjoint1(Lines, [margin(r, g, 3)]),
    labeling([], Starts).
```

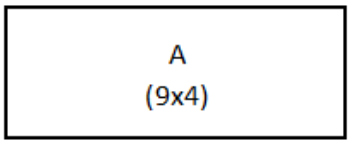
```
Starts = [1,8,12,10] ? ;
Starts = [1,10,12,8] ? ;
Starts = [3,10,12,1] ? ;
no
```

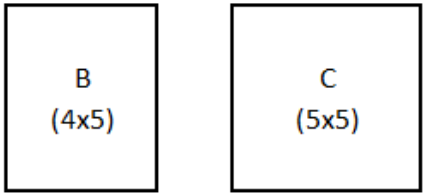
Disjoint

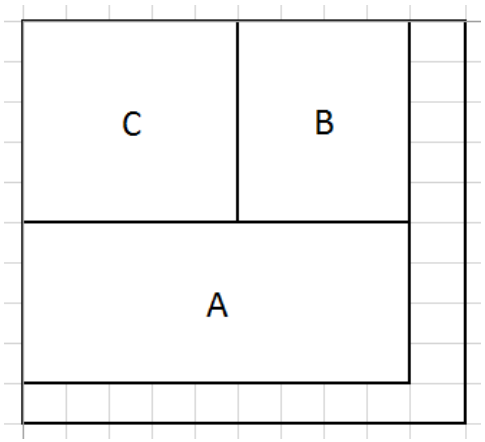
- ***disjoint2(+Rectangles)***
- ***disjoint2(+Rectangles, +Options)***
  - Restringe conjunto de retângulos de forma a que não se sobreponham
  - ***Rectangles*** é uma lista de termos no formato ***F(Xj, Lj, Yj, Hj)*** ou ***F(Xj, Lj, Yj, Hj, Tj)***
    - ***Xj*** e ***Yj*** representam a origem do retângulo j, enquanto ***Lj*** e ***Hj*** representam as suas dimensões (variáveis de domínio ou inteiros); ***F*** é um qualquer functor; ***Tj*** é um termo atómico opcional (0 por omissão) que indica o tipo de retângulo
  - Options é lista de opções
    - ***global(Boolean)*** - se ***true*** usado algoritmo para atingir uma poda mais completa
    - ***wrap(Min1, Max1, Min2, Max2)*** - ***Min1*** e ***Max1*** referem-se à dimensão X, enquanto ***Min2*** e ***Max2*** se referem à dimensão Y; se todos os valores forem inteiros, o espaço é visto como toroidal; podem ser usados os valores ***inf*** e ***sup*** (para Min e Max numa das dimensões) para atingir um espaço cilíndrico
    - ***margin(T1, T2, D1, D2)*** - impõe uma distância mínima ***D1*** em X e ***D2*** em Y entre o final de qualquer retângulo do tipo ***T1*** e o início de qualquer retângulo do tipo ***T2***; ***D1*** e ***D2*** devem ser inteiros positivos ou ***sup***: todos os retângulos do tipo ***T2*** terminam antes de qualquer retângulo do tipo ***T1*** na dimensão relevante

Disjoint

- Exemplo:
  - Colocar três retângulos numa grelha 10x10








```
place(StartsX, StartsY) :-
    StartsX = [Ax, Bx, Cx],
    StartsY = [Ay, By, Cy],
    domain(StartsX, 1, 10),
    domain(StartsY, 1, 10),
    Rectangles = [
                rect(Ax, 9, Ay, 4),
                rect(Bx, 4, By, 5),
                rect(Cx, 5, Cy, 5)
                ],
    Ax + 9 #=< 10, Ay + 4 #=< 10,
    Bx + 4 #=< 10, By + 5 #=< 10,
    Cx + 5 #=< 10, Cy + 5 #=< 10,
    disjoint2(Rectangles),
    append(StartsX, StartsY, Vars),
    labeling([], Vars).
```

StartsX = [1,6,1],
StartsY = [6,1,1] ?

Diffn

- ***diffn(+Boxes)***
- ***diffn(+Boxes, +Options)***
  - Restringe a localização no espaço de caixas (***Boxes***) multidimensionais não sobrepostas
  - ***Boxes*** é uma lista de caixas, sendo cada uma representada por uma lista de termos no formato ***Origin-Length***
    - ***Origin*** e ***Length*** são a origem e tamanho da caixa em cada dimensão
    - Todas as caixas devem ter a mesma dimensionalidade (ie, as listas devem ter o mesmo tamanho)
  - ***Options*** é uma lista de opções
    - ***strict(Boolean)*** – se ***false***, disjunção admite caixa sem comprimento em alguma(s) dimensão(ões); se ***true***, a disjunção é mais estrita

```
| ?- diffn([ [1-3, 1-3], [2-3, 4-3] ]).
yes
| ?- diffn([ [1-3, 1-3], [2-3, 2-3] ]).
no
```

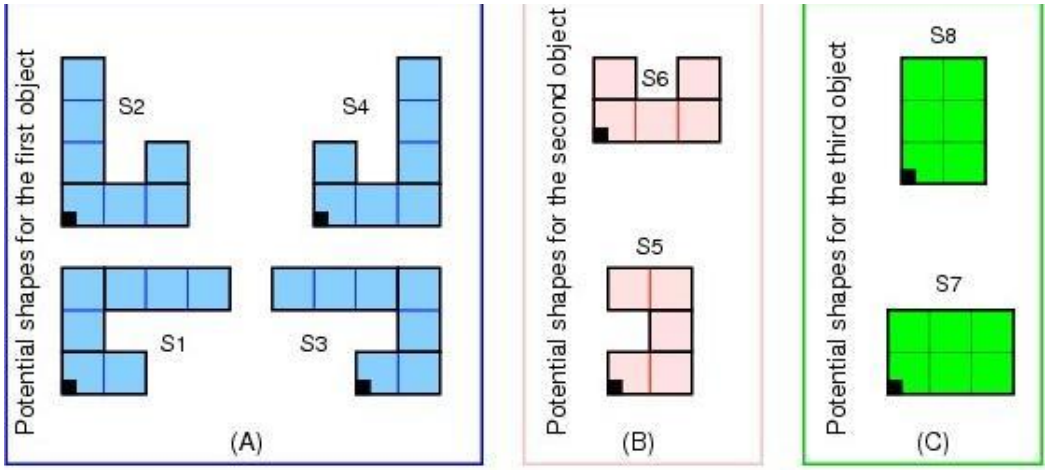
```
| ?- diffn([ [1-3, 1-0], [2-3, 0-3] ], [strict(false)]).
yes
| ?- diffn([ [1-3, 1-0], [2-3, 0-3] ], [strict(true)]).
no
```

Geost

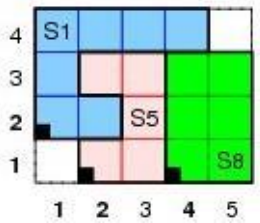
- ***geost(+Objects, +Shapes)***
- ***geost(+Objects, +Shapes, +Options)***
- ***geost(+Objects, +Shapes, +Options, +Rules)***
  - Restringe a localização no espaço de objetos (***Objects***) multidimensionais não sobrepostos, cada um dos quais tendo uma forma de entre um conjunto de formas (***Shapes***)
  - ***Objects*** é uma lista de termos no formato ***object(Oid, Sid, Origin)***
    - ***Oid*** identifica o objeto (inteiro único); ***Sid*** identifica a forma do objeto (inteiro ou variável de domínio); ***Origin*** indica coordenadas de origem do objeto (lista de inteiros ou variáveis de domínio)
  - ***Shapes*** é uma lista de termos no formato ***sbox(Sid, Offset, Size)***, representando caixas deslocadas (*shifted boxes*)
    - ***Sid*** é o identificador da forma (inteiro); ***Offset*** é uma lista de inteiros de tamanho n com o deslocamento em cada dimensão da caixa relativamente à origem do objeto; ***Size*** é uma lista de inteiros de tamanho n com o tamanho da caixa em cada dimensão
    - Cada forma é definida pelo conjunto de termos ***sbox/3*** com o mesmo ***Sid***
  - ***Options*** é uma lista de opções (ver documentação)

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]).
```



A possible placement where object 1 is assigned shape S1 and object 2 is assigned shape S5 and object 3 is assigned shape S8

(D)

Value Precede Chain

- ***value_precede_chain(+Values, +Vars)***
- ***value_precede_chain(+Values, +Vars, +Options)***
  - Forma de remover simetrias de valores
  - ***Values*** é lista de inteiros e ***Vars*** é lista de inteiros/variáveis de domínio
  - Verifica-se se para cada par de valores adjacentes ***X***, ***Y*** em ***Values***, ***Y*** não existe em Vars, ou, se ***Y*** existir em ***Vars***, ***X*** encontra-se antes de ***Y***
  - ***Options*** é lista de opções:
    - ***global(Bool)***: se ***false*** (valor por omissão) é feita uma decomposição da restrição em ***automaton/3***. Caso seja ***true***, é usado um algoritmo personalizado. Ambos mantêm consistência de domínios, mas o desempenho relativo pode variar
  - Exemplos:

```
| ?- length(L,3), domain(L, 1, 2),
value_precede_chain([3,2,1], L).
no
```

```
| ?- length(L,3), domain(L, 1, 3),
value_precede_chain([3,4,2,1], L).
L = [3,3,3] ? ;
no
```

Sequence Precede Chain

- ***seq_precede_chain(+Vars)***
- ***seq_precede_chain(+Vars, +Options)***
  - Semelhante à restrição anterior, assumindo que ***Values*** = [1, 2, 3, ...]

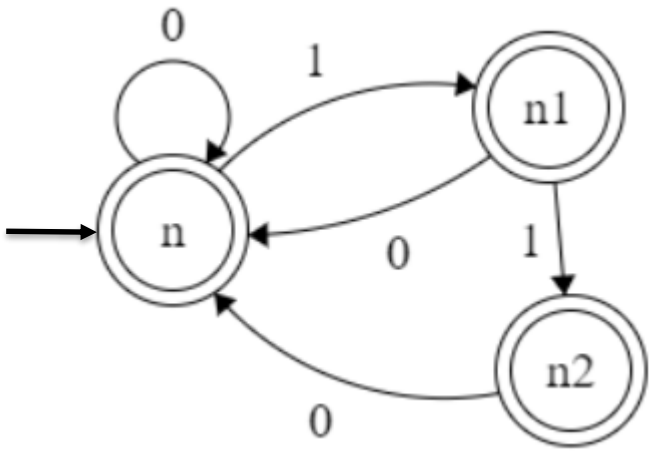
Automaton

- ***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, estendido com possíveis operações de contagem nos 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)*** ou ***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, os seus valores iniciais (normalmente instanciados) e finais (normalmente não instanciados), respetivamente
  - ***Options*** é lista de opções (ver documentação)

Automaton

- Exemplo:

```
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) ]).
```



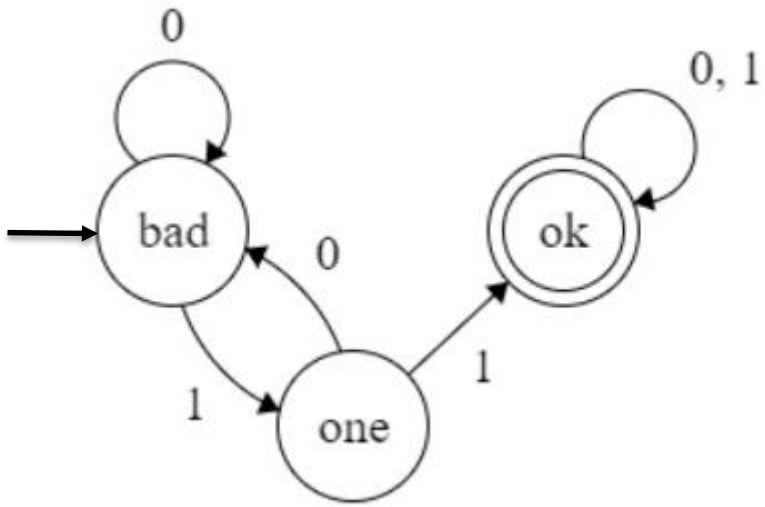

```
| ?- at_most_two_consecutive_ones([0,0,0,1,1,1]).
no
| ?- at_most_two_consecutive_ones([0,1,1,0,1,1]).
yes
| ?- at_most_two_consecutive_ones([0,1,1,0,1,0]).
yes
```

```
| ?- 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

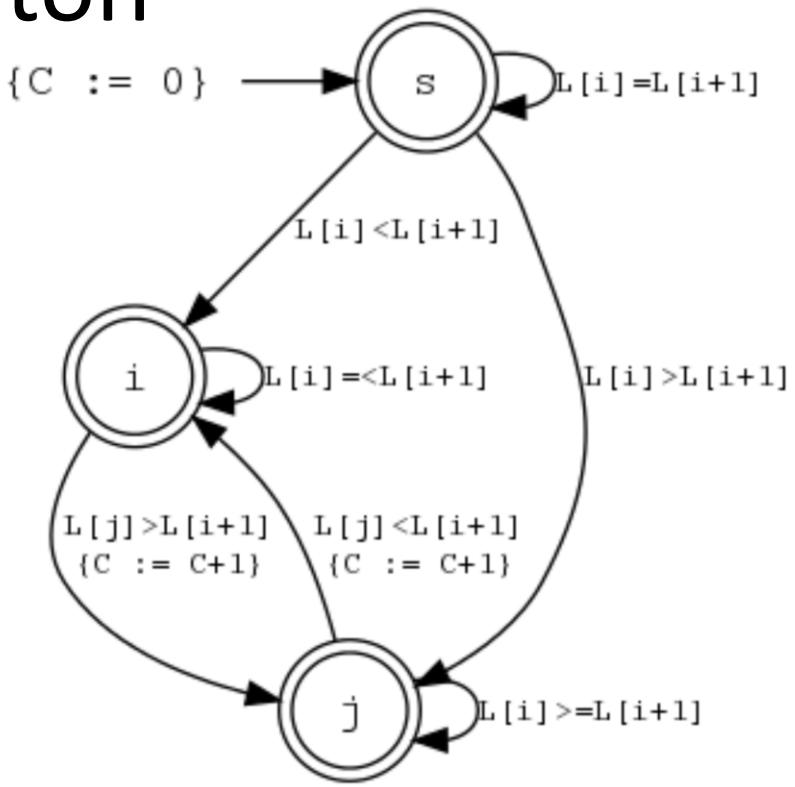
- Exemplo:

```
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

4. 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 "*satisfatibilidade*" 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)***
  - ***solve(:Options, :Searches)***

Pesquisa

- ***labeling(:Options, +Variables)***
  - ***Options*** é uma lista de opções de pesquisa
  - ***Variables*** é uma lista de variáveis de domínio ou inteiros
  - Predicado sucede se pode ser encontrada [pelo menos] uma atribuição de valores às variáveis que satisfaça todas as restrições, falhando se não houver solução / não encontrar pelo menos uma solução dentro do tempo limite
  - Exemplos:

```
| ?- declareVariables(Vars),
     postConstraints(Vars),
     labeling([], Vars).
```

```
| ?- declareVariables(Vars),
     postConstraints(Vars),
     objectiveFunction(Vars, Profit),
     labeling( [maximize(Profit), ffc, bisect,
           time_out(5000, Flag)], Vars).
```

Opções de Pesquisa

- O argumento ***Options*** de *labeling/2* (usado também em *solve/2*) controla a ordem de seleção de variáveis e valores, o tipo de solução a encontrar e a execução da pesquisa
  - Forma de ordenação de variáveis
  - Forma de seleção de valores
  - Ordenação de valores
  - Soluções a encontrar
  - Tempo limite para a pesquisa
  - Esquema de pesquisa (útil em problemas de otimização):
    - *bab* (usa *branch-and-bound*; valor por omissão), *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

Ordenação de Variáveis

- Como selecionar a próxima variável?
  - ***leftmost*** (opção por omissão): variável mais à esquerda
  - ***min***: variável com menor valor mínimo do seu domínio
  - ***max***: variável com maior valor máximo do seu domínio
  - ***ff***: variável mais à esquerda, de entre as que têm o menor domínio
  - ***anti_first_fail***: variável mais à esquerda das que têm o maior domínio
  - ***occurrence***: variável mais à esquerda com mais restrições suspensas
  - ***ffc***: variável com menor domínio, desempatando com a escolha da que tem mais restrições suspensas (*most constrained*), e em caso de novo empate, escolhendo a mais à esquerda
  - ***max_regret***: variável com maior diferença entre os dois primeiros valores de domínio, desempatando com a escolha da mais à esquerda

Ordenação de Variáveis

- Como selecionar a próxima variável? (cont)
  - ***variable(Sel)***:
    - *Sel* é um predicado para selecionar a próxima variável com a assinatura *Sel(Vars, Selected, Rest)*
    - Deve suceder deterministicamente unificando *Selected* com a variável selecionada e *Rest* com a lista de variáveis remanescentes
    - Exemplo:

```
...
labeling( [ variable(selRandom) ], Vars).

% seleciona uma variável de forma aleatória
selRandom(ListOfVars, Var, Rest):-
        random_select(Var, ListOfVars, Rest).        % da library(random)
```

Seleção de Valores

- Como selecionar valores para uma variável?
  - ***step*** (opção por omissão): 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 (média entre valores mínimo e máximo do domínio de X, com arredondamento para baixo)
  - ***median*** / ***middle***: escolha binária entre X #= M e X #\= M, onde M é a mediana / média do domínio de X

Seleção de Valores

- Como selecionar valores para uma variável?
  - ***value(Enum)***:
    - *Enum* é um predicado que deve reduzir o domínio de X com a assinatura *Enum(X, Rest, BB0, BB)*
      - *Rest* é a lista de variáveis que necessitam de *labeling* com exceção de *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
    - Exemplo:

```
labeling( [ value(selRandom) ], Vars).

selRandom(Var, Rest, BB0, BB1):-          % seleciona valor de forma aleatória
    fd_set(Var, Set), fdset_to_list(Set, List),
    random_member(Value, List),           % da library(random)
    ( first_bound(BB0, BB1), Var #= Value  ;
      later_bound(BB0, BB1), Var #\= Value ).
```

Ordenação de Valores

- Como selecionar um valor para uma variável?
  - (sem utilidade com a opção *value(Enum)*)
  - ***up*** (opção por omissão): 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 omissão): 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*
- Opções apenas com sentido para problemas de otimização:
  - ***best*** (opção por omissão): obtém a solução ótima
  - ***all***: obtém, por *backtracking*, soluções cada vez melhores

Tempo limite para a pesquisa

- Possível definir um limite temporal para a pesquisa, com opção ***time_out(Time, Flag)***
  - ***Time*** é tempo máximo de execução (em milissegundos)
  - Se provar não existir solução para o problema em ***Time*** ms, o predicado falha
  - Se for atingido o tempo limite, ou for encontrada a solução ótima, ***Flag*** é unificada com um dos seguintes valores:
    - ***optimality*** – foi encontrada a solução ótima para o problema (caso tenha sido usada *flag* ***best***) dentro do tempo limite; as variáveis são unificadas com os valores correspondentes à melhor solução
    - ***success*** – foi encontrada pelo menos uma solução para o problema (mas não atingida a prova de otimalidade) dentro do tempo limite; as variáveis são unificadas com a melhor solução encontrada até ao momento
    - ***time_out*** – foi atingido o tempo limite, sem ter sido encontrada uma solução para o problema; as variáveis ficam por instanciar

Pesquisa

- ***solve(:Options, :Searches)***
  - ***Options*** é uma lista de opções de pesquisa (semelhante às usadas em *labeling/2*)
  - ***Searches*** é uma lista com um ou mais objetivos *labeling/2* ou *indomain/1*
  - Usado principalmente para problemas de otimização, permite definir heurísticas de pesquisa distintas para [conjuntos de] variáveis diferentes
  - Algumas opções são globais, enquanto maioria são locais
    - Opções globais sobrepõem-se às opções indicadas nos objetivos *labeling/2* presentes em ***Searches***
    - Opções locais indicadas em ***Options*** definem opção por omissão caso não seja indicada nos objetivos *labeling/2* em ***Searches***

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 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 omissão): retorna solução ótima após prova de otimalidade
      - *all*: enumera soluções cada vez melhores até provar otimalidade

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).

Exemplos

- Minimizar o custo, usando duas estratégias de pesquisa diferentes para dois subconjuntos de variáveis:
  - **| ?- constraints(A, B, C, D, E, F),**
    solve([minimize(Cost)],
    [labeling([ffc, bisect], [A, C, E]),
    labeling([max_regret, median], [B, D, F])]).

PLR no SICStus Prolog

5. PREDICADOS DE ESTATÍSTICAS

Predicados de Estatísticas

- Estatísticas de execução específicas do *solver* clp(fd):
  - ***fd_statistics(?Key, ?Value)***: para cada possível chave ***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/0***: mostra um resumo das estatísticas acima (valores desde a última chamada ao predicado)

Predicados de Estatísticas

- Outras estatísticas relativas a tempo de CPU, consumo de memória e outras podem ser obtidas com os predicados:
  - ***statistics(?Keyword, ?List))***: para cada possível chave ***Keyword***, ***List*** é unificado com o valor atual de um contador. Exemplos:
    - ***runtime*** / ***total_runtime*** / ***walltime***: tempo de execução (em ms) excluindo gestão de memória e chamadas de sistema / tempo total de execução / tempo absoluto. O primeiro elemento da lista refere-se ao tempo desde o início da sessão, e o segundo refere-se ao tempo desde a última chamada ao predicado *statistics*.
    - ***memory_used***: memória usada (em bytes)
    - Várias outras opções descritas na secção 4.10.1.2 do manual do SICStus
  - ***statistics/0*** mostra resumo de estatísticas relativas a tempo de execução, memória, garbage collection, ...

Exemplo

```
testStats(Vars):-
        declareVars(Vars),
        reset_timer,
        postConstraints(Vars),
        print_time('Posting Constraints: '),
        labeling([], Vars),
        print_time('Labeling Time: '),
        fd_statistics,
        statistics.
```

```
reset_timer:-
        statistics(total_runtime, _).

print_time(Msg):-
        statistics(total_runtime,[_,T]),
        TS is ((T//10)*10)/1000, nl,
        write(Msg), write(TS), write('s'), nl, nl.
```

PLR no SICStus Prolog

Q & A