APLAI
Optimisation with Active Constraints in ECLiPSe

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

Constraint (Logic) Programming

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

5. Optimisation with active constraints

1. The `minimize/2` built-in
2. The knapsack problem
3. The coins problem
4. The currency design problem
5. The `bb_min/3` built-in
6. When the number of variables is unknown

5.1 With `lib(ic)` and `lib(branch_and_bound)`

- Finite constrained optimisation problems (COP): combine constraint propagation with branch and bound search

```
:- lib(ic).
:- lib(branch_and_bound).
solveOpt(List) :-
   declareDomains(List),
   generateConstraints_and_Cost(List, Cost),
   minimize(search(List), Cost).
```

computes a solution to the CSP with minimal value for the cost function defined in Cost

What is CSP doing??

- What does constraint propagation?
  - systematic exclusion of non-solutions from the search space
- What does search?
  - the heuristic partitioning of the search space into smaller, more manageable subspaces
- Aim?
  - finding all solutions that satisfy the constraints

What is COP doing??

- applying a bounding method on top of the all-solutions method,
- incrementallly looking for solutions that are better than a previously found one

- ```
  [eclipse 1]:
  minimize(member(X,[5,6,5,3,4,2]), X).
     % X is the cost!
  ```
  ```

Example minimize/2

```
[eclipse 1]: minimize(member(X,[5,6,5,3,4,2]), X).
Found a solution with cost 5
Found a solution with cost 3
Found a solution with cost 2
Found no solution with cost -1.0Inf .. 1

X = 2
Yes

% Cost #= X+1, minimize(member(X,[5,6,5,3,4,2]),Cost).
Found a solution with cost 6 ...
Cost = 3   X = 2
```

minimize(Goal,Cost)

- A solution of the goal *Goal* is found that minimizes the value of *Cost*. *Cost* should be a variable that is affected, and eventually instantiated, by *Goal*. Usually, *Goal* is the search procedure of a constraint problem and *Cost* is the variable representing the cost.
- The solution is found using the branch and bound method: as soon as a solution is found, it gets remembered and the search is continued with an additional constraint on the *Cost* variable which requires the next solution to be better than the previous one. Iterating this process yields an optimal solution in the end.

Launching complete search

- Find a solution to the equation $x^3 + y^3 = z^3$ such that x, y, z ∈ [100..500], with minimal value of z – x – y.

```
find(X,Y,Z) :-
  [X,Y,Z] :: [100..500],
  X*X*X  + Y*Y*Y #= Z*Z*Z,
  Cost #= Z - X - Y,
```

- How to launch search???

Example

```
find(X,Y,Z) :-
   [X,Y,Z] :: [100..500],
   X*X*X  + Y*Y*Y #= Z*Z*Z,
   Cost #= Z - X - Y,
   minimize(labeling([X,Y,Z]), Cost).

[eclipse 3]: find(X,Y,Z).
Found a solution with cost -180
Found a solution with cost -384
Found no solution with cost -1.0Inf .. -385

X = 110   Y = 388   Z = 114
```

5.2 The knapsack problem

- Combinatorial optimization problem.
- We have n objects with volumes a1, … , an and values b1, … , bn and the knapsack has volume v.
- Find a collection of the objects with maximal total value that fits in the knapsack.
- N decision variables xi with domain [0..1]
  xi has value 1 if the object i is put in the knapsack
- sum of volumes ; minimization of ???

Knapsack code

N decision variables xi with domain [0..1]

xi has value 1 if the object i is put in the knapsack

```
Xs :: [0..1],
```

We have n objects with volumes a1, … , an and the knapsack volume is v.

```
Volumes = [a1, …, an],
sigma(Volumes, Xs, Volume), Volume $=< v

sigma(L1, L2, Value) :-    % iterates simultaneously
( foreach(V1, L1),         % n is known at run-time
  foreach(V2, L2),
  foreach(Prod, ProdList)
do
  Prod = V1 * V2
),                         % Value $= a1*X1+a2*X2+..
Value $= sum(ProdList).    % built-in sum/1 !!!!
```

Knapsack code

```
sigma(L1, L2, Value) :-      % iterates simultaneously
( foreach(V1, L1),           % n is known at run-time
  foreach(V2, L2),
  foreach(Prod, ProdList)    % ProdList is a list of
do                           % arithmetic expressions
  Prod = V1 * V2
),                           %
Value $= sum(ProdList).      % built-in sum/1 !!!!
```

sum(+ExprList, -Result)

Evaluates the arithmetic expressions in ExprList and unifies their sum with Result. *ExprList* is a list of arithmetic expressions. *Result* is a variable or number.

Thus, `Value $= a1*X1+a2*X2+..`

Knapsack code

```
knapsack(Volumes, Values, Capacity, Xs) :-
  Xs :: [0..1],
  sigma(Volumes, Xs, Volume),
  Volume $=< Capacity,

  sigma(Values, Xs, Value),
  Cost $= -Value,

  minimize(labeling(Xs), Cost).
```

Knapsack run

```
?- knapsack([52,23,35,15,7],[100,60,70,15,15], 60,
   [X1,X2,X3,X4,X5]).
Found a solution with cost 0
Found a solution with cost -15
Found a solution with cost -30
Found a solution with cost -70
Found a solution with cost -85
Found a solution with cost -100
Found a solution with cost -130
Found no solution with cost -260.0 .. -131.0
X1 = 0  X2 = 1  X3 = 1   X4 = 0   X5 = 0
```

Generating Arithmetic Expressions at run-time: eval/1 (and sum/1)

```
sigma(As, Xs, V) :-
   makemysum(As, Xs, Out),       % Out is an arithmetic
   eval(Out) #= V.               % expression (run-time)
makemysum([], [], 0).
makemysum([A|As], [X|Xs], A*X + Y) :-
   makemysum(As, Xs, Y).
```

The eval/1 built-in indicates that its argument is a variable that will be bound to a symbolic expression at run-time.

Used in programs which generate constraints at run-time

See also User Manual, Chapter 8 Arithmetic evaluation

Other Finite Domain Constraints: ic_global

- Chapter 4 of Constraint Library Manual
- Constraints over lists of variables
- E.g. maxlist(+List, ?Max)
  Max is the maximum of the values in List. Operationally: Max gets updated to reflect the current range of the maximum of variables and values in List. Likewise, the list elements get constrained to the maximum given.
- minlist/2, occurrences/3, sumlist/2

5.3 The coins problem

- Find the minimum number of euro cent coins that allow us to pay exactly any amount smaller than one euro.
- six variables: x1, x2, x5, x10, x20, x50 ranging over [0..99]   amount of coins of 1/2/..50 cent
- for each i in [1..99] we have x^i1, x^i2, x^i5, x^i10, x^i20, x^i50 such that   % paying i exactly
  $i = x^i1 + 2\,x^i2 + 5x^i5 + 10x^i10 + 20x^i20 + 50\,x^i50$
  $0 \leq x^ij$ and $x^ij \leq xj$ for $j \in \{1,2,5,10,20,50\}$
- cost function??

The coins problem code

```
solve(Coins, Min) :-
    init_vars(Values, Coins),
    coin_cons(Values, Coins, Pockets),
    Min #= sum(Coins),
    minimize((labeling(Coins), check(Pockets)), Min).

init_vars(Values, Coins) :-
    Values = [1,2,5,10,20,50],
    length(Coins,6),
    Coins :: 0..99.        % Coins=[X1,X2,X5,X10,X20,X50]
```

The coins problem code

```
solve(Coins, Min) :-
   init_vars(Values, Coins),   %Coins=[X1,X2,X5,X10,X20,X50]
   coin_cons(Values, Coins, Pockets),
   Min #= sum(Coins),
   minimize((labeling(Coins), check(Pockets)), Min).

coin_cons(Values, Coins, Pockets) :-
   % Pockets is a list of "coins used for i" (Xi values)
   ( for(I,1,99),
     foreach(CoinsforI, Pockets),
     param(Coins,Values)
   do
     price_cons(I, Coins, Values, CoinsforI)
   ).
```

```
price_cons(I, Coins, Values, CoinsforI) :-
  ( foreach(V, Values),          % constraints for i
    foreach(C, CoinsforI),
    foreach(Coin, Coins),
    foreach(Prod, ProdList)
  do
    Prod = V * C,              % [1 * Xi1, 2 * Xi2, … ]
    0 #=< C,
    C #=< Coin
  ),
  I #= sum(ProdList).
```

The coins problem code

```
solve(Coins, Min) :-
   init_vars(Values, Coins),    %Coins=[X1,X2,X5,X10,X20,X50]
   coin_cons(Values, Coins, Pockets),
   Min #= sum(Coins),
   minimize((labeling(Coins), check(Pockets)), Min).

check(Pockets) :-    % there is a feasible labeling for all i
   ( foreach(CoinsforI, Pockets)
   do
     once(labeling(CoinsforI))
   ).
```

The coins problem run

```
[eclipse 5]: solve(Coins, Min)
Found a solution with cost 8
Found no solution with cost 1.0 .. 7.0

Coins = [1, 2, 1, 1, 2, 1]  Min = 8
Yes
```

5.4 The currency design problem

- freedom of choosing the values of the six coins
- solution with fewer than 8 coins???
- $i = x^i1 + 2\,x^i2 + 5x^i5 + 10x^i10 + 20x^i20 + 50\,x^i50$ now becomes …
- code can be generalised:

```
values = [ V1, V2, V3, V4, V5, V6],
0 #< V1, V1 #< V2, …  , V6 #< 100
```

```
finds solution with 8 coins,
proof for optimality: too long
```

Currency design problem: implied constraints

```
design_currency(Values, Coins) :-
  init_vars(Values, Coins),
  coin_cons(Values, Coins, Pockets),
  clever_cons(Values, Coins),
  Min #= sum(Coins),
  minimize((labeling(Values), labeling(Coins), check(Pockets)),
  Min).

init_vars(Values, Coins) :-
  length(Values, 6), Values :: 1..99, increasing(Values),
  length(Coins,6), Coins :: 0..99.

increasing(List) :-
  ( fromto(List, [This, Next|Rest], [Next|Rest], [_])
  do
    This #< Next
  ).
```

The currency design problem: implied constraints

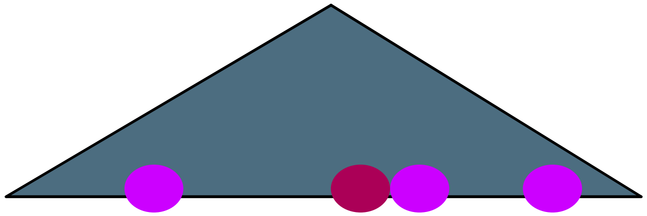
```
design_currency(Values, Coins) :-
   init_vars(Values, Coins),
   coin_cons(Values, Coins, Pockets),
   clever_cons(Values, Coins),
   Min #= sum(Coins),
   minimize((labeling(Values), labeling(Coins), check(Pockets)), Min).

% amount of coins for V1 can be kept < V2
clever_cons(Values, Coins) :-
   ( fromto(Values,[V1 | NV], NV,[]),
     fromto(Coins,[N1 | NN], NN,[])
   do      % use a V2 coin instead of N1 * V1 >= V2
           % note:always enough coins to make up any
   amount up to V1-1 (V2 - (N1*V1) still has to be paid)
     ( NV = [ V2 | _] -> N1 * V1 #< V2;  N1 * V1 #< 100)
   ).
```

Currency design run

```
?- design_currency(K,L).
Found a solution with cost 19
Found a solution with cost 17
…
Found a solution with cost 9
Found a solution with cost 8
Found no solution with cost 1.0 .. 7.0
K = [1, 2, 3, 4, 11, 33]
L = [1, 1, 0, 2, 2, 2]
Yes (297.49s cpu)
```

5.5 Best Solution: Optimization



- Branch-and-bound method

  finding the best of many solutions

  without checking them all

  ```
  :- lib(branch_and_bound).
  ```

- Search code for all-solutions can simply be wrapped into the optimisation primitive:

  ```
  bb_min(labeling(Vars), Cost, Options)
  ```

- Options:

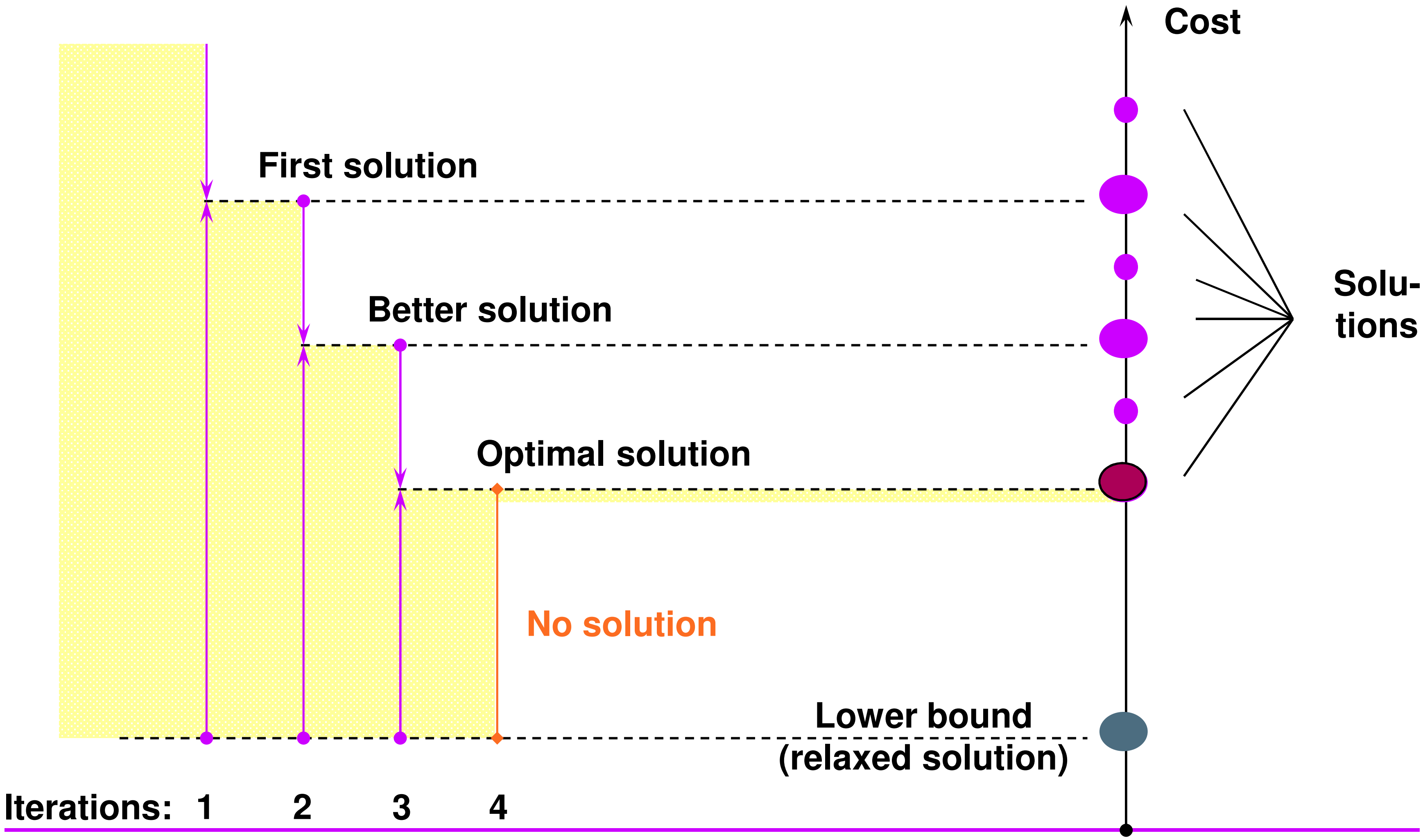
  Strategy: continue, restart, dichotomic

  Initial cost bounds (if known)

  Minimum improvement (absolute/percentage) between solutions

  Timeout

Branch-and-bound (incremental)



Impact of search strategy on b&b

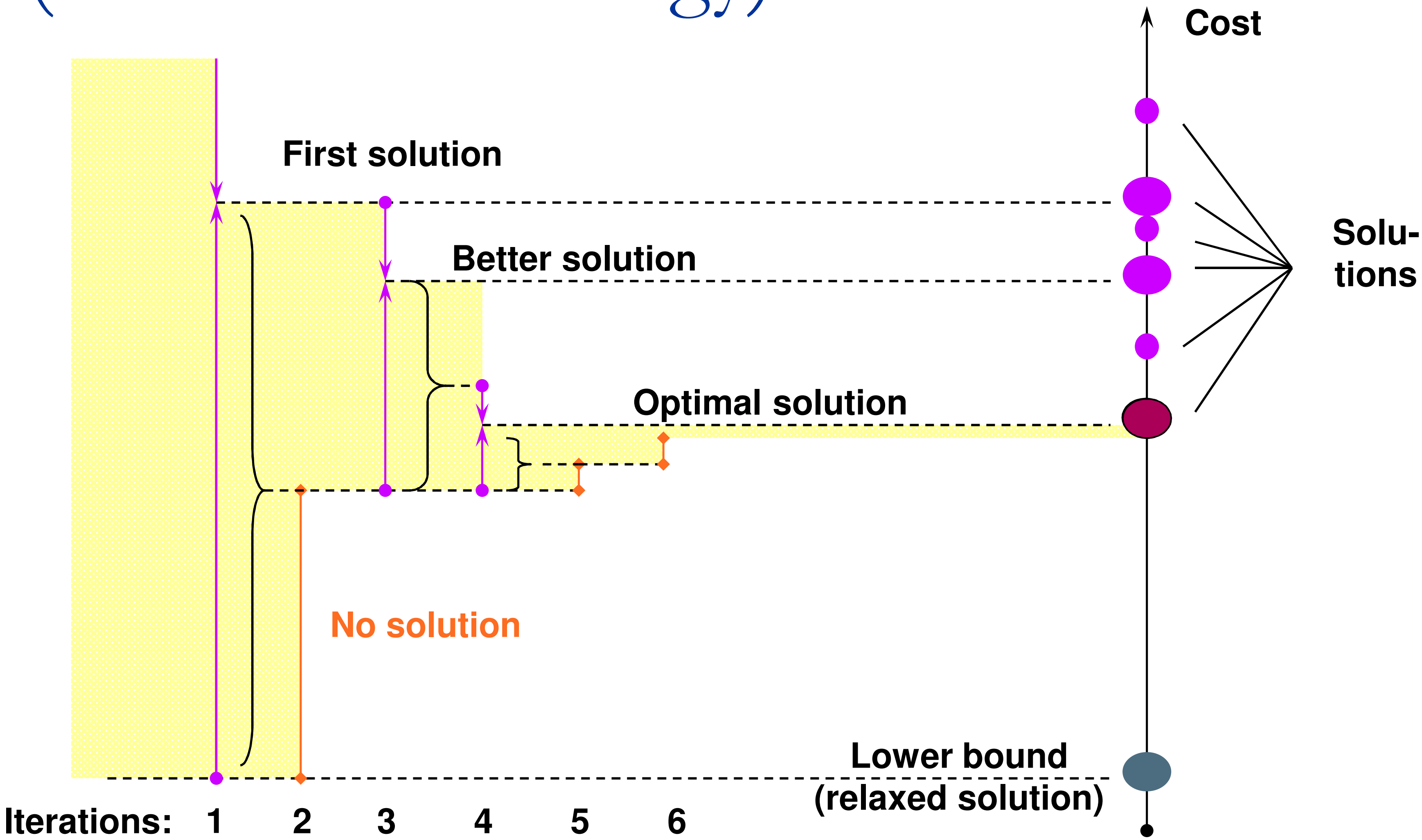
- Search space size 5, unlucky

```
?- X::1..5, Cost #= 6-X, minimize(labeling([X]),
   Cost).
Found a solution with cost 5
Found a solution with cost 4
Found a solution with cost 3
Found a solution with cost 2
Found a solution with cost 1
X=5
Cost=1
```

- Search space size 5, lucky

```
?- X::1..5, Cost #= X, minimize(labeling([X]), Cost).
Found a solution with cost 1
X=1
Cost=1
```

Branch-and-bound (dichotomic strategy)



Using dichotomic b&b strategy

- Part of search space (solution 4) skipped:

```
?- X::1..5, Cost #= 6-X, bb_min(labeling([X]), Cost,
           bb_options with strategy:dichotomic).
Found a solution with cost 5
Found a solution with cost 3
Found a solution with cost 2
Found a solution with cost 1
X = 5
Cost = 1
```

- after finding a solution, split the remaining cost range and restart search to find a solution in the lower sub-range. If that fails, assume the upper sub-range as the remaining cost range and split again.

The `bb_min/3` built-in

- the process of finding successively better solutions
- the proof of optimality: search for an even better solution and ultimately failing
- cost of next solution has to be 3 better

```
bb_min(member(X,[5,6,5,3,4,2]), X, bb_options{delta:3}).
                                                    % delta 4???
Found a solution with cost 5
Found a solution with cost 2
Found no solution with cost -1.0Inf .. -1

X = 2
Yes (0.00s cpu)
```

The `bb_min/3` built-in: improvement factor

- How much better the next solution should be than the last; number between 0 an 1 which relates the improvement to the current cost upper bound and lower bound.
- 1 : used by minimize/2 ; puts the new upper bound at the last found best cost
- 0.01: sets new upper bound almost to the cost lower bound (it is assumed to be easy to prove that there is no such solution).

Typical factor values

- 0.9: 10% improvement in each step (scheduling)
- 2/3: for currency design problem
  - Faster!!!

```
?- design_currency(K, L).
Found a solution with cost 19
Found a solution with cost 12
Found a solution with cost 8
Found no solution with cost 1.0 .. 5.662
K = [1, 2, 3, 4, 11, 33]
L = [1, 1, 0, 2, 2, 2]
Yes (30.25s cpu)       % before about 300s
```

Strategy options used in branch and bound

- minimize uses strategy:continue
- could be better to restart: restarts the search whenever a new optimum is found. Can be more efficient if the tightened cost focusses the variable choice heuristic on the right variables

```
hardy([X1,X2,Y1,Y2], Z) :-
   X1 #> 0, X2 #> 0, Y1 #> 0 , Y2 #> 0,
   X1 #\= Y1, X1 #\= Y2,
   X1^3 + X2^3 #= Z,  Y1^3 + Y2^3 #= Z,
   bb_min(labeling([X1,X2,Y1,Y2]), Z,
             bb_options{strategy:restart}).
```

Restart strategy run

```
?- hardy(L, Z).
Found a solution with cost 1729
Found no solution with cost 2.0 .. 1728.0
L = [1, 12, 9, 10]
Z = 1729
There are 4 delayed goals.
Yes (0.00s cpu)


% with continue
% takes too long
```

5.6 When the number of variables is unknown

- Given m, check whether it can be written as a sum of at least two different cubes. If yes, produce the smallest solution in the number of cubes.
- Even if m is fixed, it is not clear what the number of variables of the CSP is.
- Solution: systematically pick a candidate number n between 2 and m, and try to find a solution for n (a CSP with n variables!!)
- use customary backtracking combined with constraint propagation.
- Additional constraints???

Cubes code

```
cubes(M,Qs) :-    % fix converts real to an integer
  K is fix(round(sqrt(M))), N :: [2..K],
  indomain(N),    % choicepoint: start with small Ns
  length(Qs,N), Qs :: [1..K],
  increasing(Qs),
  ( foreach(Q,Qs),
    foreach(Expr, Exprs)
  do
    Expr = Q*Q*Q
  ),
  sum(Exprs) #= M,
  labeling(Qs), !.
```

Cubes run

```
?- cubes(33, Os).
No (0.00s cpu)
?- cubes(100, Os).
Os = [1, 2, 3, 4]
There is 1 delayed goal.
Yes (0.00s cpu)
```