

CR TP6 Programmation par contraintes

Sur une balançoire

Julien LETOILE, Romain HUBERT

le 17/03/2021

Table des matières

Table des matières

I. Réponses rédigées

Question 6.1

Question 6.2

Question 6.3

Question 6.4

Version de base

Version 1

Version 2

Version 3

Version 4

Question de compréhension

Choix des stratégies

Prédicats utilitaires

II. Code source

I. Réponses rédigées

Question 6.1

```
1  /* Pose les contraintes sur les données */
2  getData(Poids, Personnes):-
3      Personnes = [(ron, zoe, jim, lou, luc, dan, ted, tom, max,
4                    kim),
5                    Personnes &:: noms,
6                    Poids = [(24, 39, 85, 60, 165, 6, 32, 123, 7, 14)].
7
8  /* Pose les contraintes sur les variables */
9  getVar(Places, Longueur):-
10     dim(Places, [Longueur]),
11     Places #:: [-8 .. -1, 1 .. 8].
12
13 pose_contrainte(Places, Poids, Personnes):-
14     equilibre(Places, Poids),
15     contraindreLouEtTom(Places, Personnes),
16     contraindreDanEtMax(Places, Personnes),
17     cinqDeChaqueCote(Places),
18     enleveSymetrie(Places, Personnes),
19     ic:alldifferent(Places).
20
21 /* Garantit que la balançire est en équilibre */
22 equilibre(Places, Poids):-
23     produitScalaire(Places, Poids, Somme),
24     Somme #= 0.
25
26 /* Place Lou et Tom aux extrémités */
27 contraindreLouEtTom(Places, Personnes):-
28     ic:max(Places, Max),
29     ic:min(Places, Min),
30     indicePersonne(lou, Personnes, IndiceLou),
31     indicePersonne(tom, Personnes, IndiceTom),
32     Places[IndiceLou] #>= Max or Places[IndiceLou] #<= Min,
33     Places[IndiceTom] #>= Max or Places[IndiceTom] #<= Min.
```

```

34
35  /* Place Dan et Max devant Lou et Tom */
36  contraindreDanEtMax(Places, Personnes):-
37      indicePersonne(dan, Personnes, Dan),
38      indicePersonne(max, Personnes, Max),
39      indicePersonne(lou, Personnes, Lou),
40      indicePersonne(tom, Personnes, Tom),
41      (Places[Lou] > 0) => ((Places[Dan] #= Places[Lou] - 1 and
Places[Max] #= Places[Tom] + 1) or (Places[Max] #= Places[Lou] -
1 and Places[Dan] #= Places[Tom] + 1)),
42      (Places[Lou] < 0) => ((Places[Dan] #= Places[Lou] + 1 and
Places[Max] #= Places[Tom] - 1) or (Places[Max] #= Places[Lou] +
1 and Places[Dan] #= Places[Tom] - 1)).
43
44  /* Garantit que 5 personnes seront de part de d'autre de la
balançoire */
45  cinqDeChaqueCote(Places):-
46      (
47          foreachelem(Place, Places),
48          fromto(0, AncienCompteGauche, NouveauCompteGauche, 5)
49          do
50              NouveauCompteGauche #= AncienCompteGauche + (Place #< 0)
51      ).
52
53  /* Recherche l'indice d'une personne */
54  indicePersonne(Personne, Personnes, Indice):-
55      dim(Personnes, [Longueur]),
56      (
57          for(I, 1, Longueur),
58          param(Personne, Personnes, Indice)
59          do
60              X is Personnes[I],
61              /* Toujours mettre un else dans un if (cond -> true;
false) */
62              (Personne &= X -> Indice #= I; true)
63      ).
64
65
66  getVarList(Places, List):-
67      term_variables(Places, List).

```

Question 6.2

```
1  /* Prédicat de résolution du problème */
2  solve:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6      pose_contrainte(Places, Poids, Personnes),
7      norme(Places, Poids, Norme),
8      labeling(Places),
9      affiche(Places, Personnes, Norme).
```

Test:

```
tom  dan  ron          jim    ted | kim          zoe  luc  max  lou
. -8   -7   -6   -5   -4   -3   -2   -1 | 1    2    3    4    5    6    7
8
```

Norme : 2914

Question 6.3

Les personnes peuvent s'asseoir sur le même siège mais de l'autre côté de la balançoire, entraînant ainsi la même solution mais à l'opposé.

On a alors une symétrie par rapport au milieu. Ainsi, on aura 2 fois moins de solutions, et on ne gardera que les solutions non triviales.

```
1  /* Pour enlever la symetrie, il suffit de choisir un côté pour
   soit Lou, soit Tom vu qu'on sait qu'ils sont de part et d'autres
   de la balançoire */
2  enleveSymetrie(Places, Personnes):-
3      indicePersonne(lou, Personnes, IndiceLou),
4      Places[IndiceLou] #> 0. /* Choisit de mettre la place de Lou
   à droite */
```

Question 6.4

Version de base

```
1  /* Prédicat de résolution avec Branch and Bound */
2  solve_minimize_v0:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6
7      pose_contrainte(Places, Poids, Personnes),
8      norme(Places, Poids, Norme),
9
10     /* Recherche par le plus contraint, dans l'ordre croissant
11        */
11     minimize(labeling(Places), Norme),
12     affiche(Places, Personnes, Norme).
```

Test:

	tom	dan	ted	kim	zoe		luc	jim	ron		max	lou				
-8	-7	-6	-5	-4	-3	-2	-1		1	2	3	4	5	6	7	8

Norme : 802

P = [(3, -1, 2, 6, 1, -4, -3, -5, 5, -2)

Yes (2.13s cpu)

Version 1

```

1  /* On essaie d'échouer le plus tôt possible pour éviter de
   développer des branches inutiles */
2  solve_minimize_v1:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6
7      pose_contrainte(Places, Poids, Personnes),
8      norme(Places, Poids, Norme),
9
10     /* Recherche par le plus contraint, dans l'ordre croissant
    */
11     minimize(search(Places, 0, most_constrained, indomain_min,
    complete, []), Norme),
12     affiche(Places, Personnes, Norme).

```

Test:

				tom	dan	ted	kim	zoe		luc	jim	ron		max	lou		
-8	-7	-6	-5	-4	-3	-2	-1	-1		1	2	3	4	5	6	7	8

Norme : 802

$P = [(3, -1, 2, 6, 1, -4, -3, -5, 5, -2)]$

Yes (0.23s cpu)

Version 2

```

1  /* Les variables sont triées pour correspondre au problème */
2  solve_minimize_v2:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6
7      pose_contrainte(Places, Poids, Personnes),
8      norme(Places, Poids, Norme),
9
10     minimize(search(Places, 0, input_order, indomain_middle,
11                    complete, []), Norme),
12     affiche(Places, Personnes, Norme).

```

Test:

Found a solution with cost 1277

Found a solution with cost 1176

Found a solution with cost 1138

Found a solution with cost 1053

Found a solution with cost 972

Found a solution with cost 933

Found a solution with cost 875

Found a solution with cost 852

Found a solution with cost 802

Found no solution with cost 30.0 .. 801.0

ron : 3

zoe : -1

jim : 2

lou : 6

luc : 1

dan : -4

ted : -3

tom : -5

max : 5

kim : -2

				tom	dan	ted	kim	zoe		luc	jim	ron		max	lou		
-8	-7	-6	-5	-4	-3	-2	-1	-1		1	2	3	4	5	6	7	8

Norme : 802

Yes (1.35s cpu)

Version 3

```
1  /* Combinaison des versions 1 et 2 */
2  solve_minimize_v3:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6
7      pose_contrainte(Places, Poids, Personnes),
8      norme(Places, Poids, Norme),
9
10     minimize(search(Places, 0, most_constrained,
11                    indomain_middle, complete, []), Norme),
12     affiche(Places, Personnes, Norme).
```

Test:

Found a solution with cost 945

Found a solution with cost 802

Found no solution with cost 30.0 .. 801.0

ron : 3

zoe : -1

jim : 2

lou : 6

luc : 1

dan : -4

ted : -3

tom : -5

max : 5

kim : -2

				tom	dan	ted	kim	zoe		luc	jim	ron		max	lou	
-8	-7	-6	-5	-4	-3	-2	-1		1	2	3	4	5	6	7	8

Norme : 802

Yes (0.14s cpu)

Version 4

```
1  /* Les variables sont triées selon une heuristique indéfinie our
   le moment */
2  solve_minimize_v4:-
3      getData(Poids, Personnes),
4      dim(Poids, [Longueur]),
5      getVar(Places, Longueur),
6
7      pose_contrainte(Places, Poids, Personnes),
8      norme(Places, Poids, Norme),
9
10     /* Recherche par le plus contraint, dans l'ordre croissant
    */
11     getVarListOpti(Places, Liste),
12
13     minimize(search(Liste, 0, most_constrained, indomain_middle,
    complete, []), Norme),
14     affiche(Places, Personnes, Norme).
15
16 /* Tri des données par ordre décroissant des poids */
17 getVarListOpti(Places, Liste):-
18     PlaceRon is Places[1],
19     PlaceZoe is Places[2],
20     PlaceJim is Places[3],
21     PlaceLou is Places[4],
22     PlaceLuc is Places[5],
23     PlaceDan is Places[6],
24     PlaceTed is Places[7],
25     PlaceTom is Places[8],
26     PlaceMax is Places[9],
27     PlaceKim is Places[10],
28     Liste = [PlaceLuc, PlaceTom, PlaceJim, PlaceLou, PlaceZoe,
    PlaceTed, PlaceRon, PlaceKim, PlaceMax, PlaceDan].
```

Test:

Found a solution with cost 945

Found a solution with cost 848

Found a solution with cost 802

Found no solution with cost 30.0 .. 801.0

ron : 3

zoe : -1

jim : 2

lou : 6

luc : 1

dan : -4

ted : -3

tom : -5

max : 5

kim : -2

.				tom	dan	ted	kim	zoe		luc	jim	ron		max	lou		
-8	-7	-6	-5	-4	-3	-2	-1	-1		1	2	3	4	5	6	7	8

Norme : 802

Yes (0.14s cpu)

Question de compréhension

Le labeling original prend les variables dans l'ordre de la liste en prenant les valeurs croissantes. En faisant cette technique, on risque de commencer par des variables avec un grand domaine, et donc aggrandi le temps de trouver une solution. Il n'est donc pas efficace puisque, dans notre cas, on commence avec une variable à grand domaine en partant de faibles valeurs. Ainsi, le temps de trouver une solution est plus important qu'avec notre propre labeling.

Choix des stratégies

Version 2 : Nous avons choisi de chercher les solutions vers le centre des domaines parce qu'il est fort probable qu'une des solutions se trouve vers le milieu.

Version 4 : Nous avons placé les personnes plus lourdes en avant pour augmenter leur priorité. En effet, une personne plus lourde affecte plus le moment qu'une personne légère donc il a moins de choix pour se placer sur la balançoire. On réduit ainsi le domaine pour obtenir plus rapidement une solution.

Prédicats utilitaires

```
1  /* Calcule la norme d'un vecteur */
2  norme(Places, Poids, Norme):-
3      absVecteur(Places, AbsVecteur),
4      produitScalaire(AbsVecteur, Poids, Scalaire),
5      Norme #= Scalaire / 2.
6
7  /* Calcule le produit cartésien de deux vecteurs */
8  produitVecteur(Vect1, Vect2, Res):-
9      dim(Vect1, [Longueur]),
10     dim(Vect2, [Longueur]),
11     dim(Res, [Longueur]),
12     (
13         for(I, 1, Longueur),
14         param(Vect1, Vect2, Res)
15     do
16         Res[I] #= Vect1[I] * Vect2[I]
17     ).
18
19  /* Calcule le produit scalaire de deux vecteurs */
20  produitScalaire(Vect1, Vect2, Res):-
21      produitVecteur(Vect1, Vect2, Vect3),
22      sommeVecteur(Vect3, Res).
23
24  sommeVecteur(Vect, Res):-
25      (
26          foreachelem(X, Vect),
27          fromto(0, AncienneValeur, NouvelleValeur, Res)
28      do
29          NouvelleValeur #= AncienneValeur + X
30      ).
31
32  /* Calcule la valeur absolue de chaque élément du vecteur Vect
   */
```

```

33  absVecteur(Vect, Res):-
34      dim(Vect, [Longueur]),
35      dim(Res, [Longueur]),
36      (
37          for(I, 1, Longueur),
38          param(Vect, Res)
39      do
40          X is Vect[I],
41          Res[I] #= abs(X)
42      ).
43
44  /* Trouve l'indice d'un élément dans un vecteur */
45  indiceDe(Element, Vecteur, Indice):-
46      dim(Vecteur, [Longueur]),
47      (
48          for(I, 1, Longueur),
49          fromto(0, AncienneValeur, NouvelleValeur, Indice),
50          param(Element, Vecteur)
51      do
52          X is Vecteur[I],
53          NouvelleValeur #= (Element #= X) * I + AncienneValeur *  

54          (Element #\= X)
55      ).
56  /* Affichage des resultats */
57  affiche(Places, Personnes, Norme):-
58      dim(Places, [Longueur]),
59
60      /* Affiche personne : place */
61      (
62          for(I, 1, Longueur),
63          param(Places, Personnes)
64      do
65          Place is Places[I],
66          Personne is Personnes[I],
67          write(Personne),
68          write(" : "),
69          writeln(Place)
70      ),
71      writeln(""),
72      writeln(""),

```

```

73
74     /* Affiche les noms des personnes au bon endroit sur la
    balancoire */
75     (
76         for(I, -8, 8),
77         param(Places, Personnes)
78     do
79         indiceDe(I, Places, Indice),
80         (I #= 0 ->
81             write("|")
82         ;
83             (Indice #> 0 ->
84                 Personne is Personnes[Indice],
85                 write("  "),
86                 write(Personne),
87                 write("  ")
88             ;
89                 write("      ")
90             )
91         )
92     ),
93     writeln(""),
94
95     /* Affiche la barre de la balancoire */
96     (
97         for(I, -8, 8)
98     do
99         (I #= 0 ->
100             write(" ")
101         ;
102             write(" ----- ")
103         )
104     ),
105     writeln(""),
106
107     /* Affiche la place de la balancoire en dessous */
108     (
109         for(I, -8, 8)
110     do
111         (I #> 0 -> write("  "), write(I), write("  "); true),
112         (I #< 0 -> write("  "), write(I), write("  "); true),

```

```
13         (I != 0 -> write("|"); true)
14     ),
15     writeln(""),
16     writeln(""),
17     write("Norme : "),
18     writeln(Norme),
19     writeln("").
```

II. Code source

On le posera quand il sera fini.

```
1  :-lib(ic).
2  :-lib(ic_symbolic).
3  :-lib(branch_and_bound).
4
5
6  ?- local domain(noms(ron, zoe, jim, lou, luc, dan, ted, tom,
   max, kim)).
7
8
9  /* minimize(+Goal, ?Cost, +Options) : A solution of the goal
   Goal is found that minimizes the value of Cost using branch and
   bound */
10 /* search(+L, ++Arg, ++Select, +Choice, ++Method, +Option) : */
11
12 /* Q 6.2 */
13 /* Prédicat de résolution du problème */
14 solve:-
15     getData(Poids, Personnes),
16     dim(Poids, [Longueur]),
17     getVar(Places, Longueur),
18     pose_contrainte(Places, Poids, Personnes),
19     norme(Places, Poids, Norme),
20     labeling(Places),
21     affiche(Places, Personnes, Norme).
22
23 /* Prédicats de résolution avec Branch and Bound */
24 solve_minimize_v0:-
25     getData(Poids, Personnes),
26     dim(Poids, [Longueur]),
27     getVar(Places, Longueur),
28
29     pose_contrainte(Places, Poids, Personnes),
30     norme(Places, Poids, Norme),
31
```



```

32      /* Recherche par le plus contraint, dans l'ordre croissant
33      */
34      minimize(labeling(Places), Norme),
35      affiche(Places, Personnes, Norme).
36
37 solve_minimize_v1:-
38     getData(Poids, Personnes),
39     dim(Poids, [Longueur]),
40     getVar(Places, Longueur),
41
42     pose_contrainte(Places, Poids, Personnes),
43     norme(Places, Poids, Norme),
44
45     /* Recherche par le plus contraint, dans l'ordre croissant
46     */
47     minimize(search(Places, 0, most_constrained, indomain_min,
48     complete, []), Norme),
49     affiche(Places, Personnes, Norme).
50
51 solve_minimize_v2:-
52     getData(Poids, Personnes),
53     dim(Poids, [Longueur]),
54     getVar(Places, Longueur),
55
56     pose_contrainte(Places, Poids, Personnes),
57     norme(Places, Poids, Norme),
58
59     minimize(search(Places, 0, input_order, indomain_middle,
60     complete, []), Norme),
61     affiche(Places, Personnes, Norme).
62
63 solve_minimize_v3:-
64     getData(Poids, Personnes),
65     dim(Poids, [Longueur]),
66     getVar(Places, Longueur),
67
68     pose_contrainte(Places, Poids, Personnes),
69     norme(Places, Poids, Norme),
70
71     minimize(search(Places, 0, most_constrained,
72     indomain_middle, complete, []), Norme),

```

```

68     affiche(Places, Personnes, Norme).
69
70 solve_minimize_v4:-
71     getData(Poids, Personnes),
72     dim(Poids, [Longueur]),
73     getVar(Places, Longueur),
74
75     pose_contrainte(Places, Poids, Personnes),
76     norme(Places, Poids, Norme),
77
78     /* Recherche par le plus contraint, dans l'ordre croissant
79     */
80     getVarListOpti(Places, Liste),
81
82     minimize(search(Liste, 0, most_constrained, indomain_middle,
83     complete, []), Norme),
84     affiche(Places, Personnes, Norme).
85
86 /* Q 6.1 */
87 /* Pose les contraintes sur les données */
88 getData(Poids, Personnes):-
89     Personnes = [(ron, zoe, jim, lou, luc, dan, ted, tom, max,
90     kim),
91     Personnes &:: noms,
92     Poids = [(24, 39, 85, 60, 165, 6, 32, 123, 7, 14)].
93
94 /* Pose les contraintes sur les variables */
95 getVar(Places, Longueur):-
96     dim(Places, [Longueur]),
97     Places #:: [-8 .. -1, 1 .. 8].
98
99 pose_contrainte(Places, Poids, Personnes):-
100     equilibre(Places, Poids),
101     contraindreLouEtTom(Places, Personnes),
102     contraindreDanEtMax(Places, Personnes),
103     cinqDeChaqueCote(Places),
104     enleveSymetrie(Places, Personnes),
105     ic:alldifferent(Places).

```

```

06  /* Garantit que la balançire est en équilibre */
07  equilibre(Places, Poids):-
08      produitScalaire(Places, Poids, Somme),
09      Somme #= 0.
10
11  /* Place Lou et Tom aux extrémités */
12  contraindreLouEtTom(Places, Personnes):-
13      ic:max(Places, Max),
14      ic:min(Places, Min),
15      indicePersonne(lou, Personnes, IndiceLou),
16      indicePersonne(tom, Personnes, IndiceTom),
17      Places[IndiceLou] #>= Max or Places[IndiceLou] #<= Min,
18      Places[IndiceTom] #>= Max or Places[IndiceTom] #<= Min.
19
20  /* Place Dan et Max devant Lou et Tom */
21  contraindreDanEtMax(Places, Personnes):-
22      indicePersonne(dan, Personnes, Dan),
23      indicePersonne(max, Personnes, Max),
24      indicePersonne(lou, Personnes, Lou),
25      indicePersonne(tom, Personnes, Tom),
26      (Places[Lou] > 0) => ((Places[Dan] #= Places[Lou] - 1 and
Places[Max] #= Places[Tom] + 1) or (Places[Max] #= Places[Lou] -
1 and Places[Dan] #= Places[Tom] + 1)),
27      (Places[Lou] < 0) => ((Places[Dan] #= Places[Lou] + 1 and
Places[Max] #= Places[Tom] - 1) or (Places[Max] #= Places[Lou] +
1 and Places[Dan] #= Places[Tom] - 1)).
28
29  /* Garantit que 5 personnes seront de part de d'autre de la
balançoire */
30  cinqDeChaqueCote(Places):-
31      (
32          foreach(Place, Places),
33          fromto(0, AncienCompteGauche, NouveauCompteGauche, 5)
34          do
35              NouveauCompteGauche #= AncienCompteGauche + (Place #< 0)
36      ).
37
38  /* Pour enlever la symetrie, il suffit de choisir un côté pour
soit Lou, soit Tom vu qu'on sait qu'ils sont de part et d'autres
de la balançoire */
39  enleveSymetrie(Places, Personnes):-

```

```

40     indicePersonne(lou, Personnes, IndiceLou),
41     Places[IndiceLou] #> 0. /* Choisi de mettre la place de Lou
    à droite */
42
43 /* Calcule la norme d'un vecteur */
44 norme(Places, Poids, Norme):-
45     absVecteur(Places, AbsVecteur),
46     produitScalaire(AbsVecteur, Poids, Scalaire),
47     Norme #= Scalaire / 2.
48
49
50 /* Toujours mettre un else dans un if (cond -> true; false) */
51 /* Recherche l'indice d'une personne */
52 indicePersonne(Personne, Personnes, Indice):-
53     dim(Personnes, [Longueur]),
54     (
55         for(I, 1, Longueur),
56         param(Personne, Personnes, Indice)
57     do
58         X is Personnes[I],
59         (Personne &= X -> Indice #= I; true)
60     ).
61
62
63 getVarList(Places, List):-
64     term_variables(Places, List).
65
66
67 /* Tri des données par ordre décroissant des poids */
68 getVarListOpti(Places, Liste):-
69     PlaceRon is Places[1],
70     PlaceZoe is Places[2],
71     PlaceJim is Places[3],
72     PlaceLou is Places[4],
73     PlaceLuc is Places[5],
74     PlaceDan is Places[6],
75     PlaceTed is Places[7],
76     PlaceTom is Places[8],
77     PlaceMax is Places[9],
78     PlaceKim is Places[10],

```

```

79     Liste = [PlaceLuc, PlaceTom, PlaceJim, PlaceLou, PlaceZoe,
PlaceTed, PlaceRon, PlaceKim, PlaceMax, PlaceDan].

80

81  /* Affichage des resultats */
82  affiche(Places, Personnes, Norme):-
83      dim(Places, [Longueur]),
84
85      /* Affiche personne : place */
86      (
87          for(I, 1, Longueur),
88          param(Places, Personnes)
89      do
90          Place is Places[I],
91          Personne is Personnes[I],
92          write(Personne),
93          write(" : "),
94          writeln(Place)
95      ),
96      writeln(""),
97      writeln(""),
98
99      /* Affiche les noms des personnes au bon endroit sur la
balancoire */
100     (
101         for(I, -8, 8),
102         param(Places, Personnes)
103     do
104         indiceDe(I, Places, Indice),
105         (I #= 0 ->
106             write("|")
107         ;
108             (Indice #> 0 ->
109                 Personne is Personnes[Indice],
110                 write("  "),
111                 write(Personne),
112                 write("  ")
113             ;
114                 write("      ")
115             )
116         )
117     ),

```

```

18     writeln(""),
19
20     /* Affiche la barre de la balançoire */
21     (
22         for(I, -8, 8)
23     do
24         (I #= 0 ->
25             write(" ")
26         ;
27             write(" ----- ")
28         )
29     ),
30     writeln(""),
31
32     /* Affiche la place de la balançoire en dessous */
33     (
34         for(I, -8, 8)
35     do
36         (I #> 0 -> write("  "), write(I), write("  "); true),
37         (I #< 0 -> write("  "), write(I), write("  "); true),
38         (I #= 0 -> write("|"); true)
39     ),
40     writeln(""),
41     writeln(""),
42     write("Norme : "),
43     writeln(Norme),
44     writeln("").
45
46
47     /* Q 6.3 */
48
49
50     /****** FONCTIONS UTILITAIRES *****/
51
52     /* Calcule le produit cartésien de deux vecteurs */
53     produitVecteur(Vect1, Vect2, Res):-
54         dim(Vect1, [Longueur]),
55         dim(Vect2, [Longueur]),
56         dim(Res, [Longueur]),
57         (
58             for(I, 1, Longueur),

```

```

59         param(Vect1, Vect2, Res)
60     do
61         Res[I] #= Vect1[I] * Vect2[I]
62     ).
63
64 /* Calcule le produit scalaire de deux vecteurs */
65 produitScalaire(Vect1, Vect2, Res):-
66     produitVecteur(Vect1, Vect2, Vect3),
67     sommeVecteur(Vect3, Res).
68
69 sommeVecteur(Vect, Res):-
70     (
71         foreachelem(X, Vect),
72         fromto(0, AncienneValeur, NouvelleValeur, Res)
73     do
74         NouvelleValeur #= AncienneValeur + X
75     ).
76
77 /* Calcule la valeur absolue de chaque élément du vecteur Vect */
78 absVecteur(Vect, Res):-
79     dim(Vect, [Longueur]),
80     dim(Res, [Longueur]),
81     (
82         for(I, 1, Longueur),
83         param(Vect, Res)
84     do
85         X is Vect[I],
86         Res[I] #= abs(X)
87     ).
88
89 /* Trouve l'indice d'un élément dans un vecteur */
90 indiceDe(Element, Vecteur, Indice):-
91     dim(Vecteur, [Longueur]),
92     (
93         for(I, 1, Longueur),
94         fromto(0, AncienneValeur, NouvelleValeur, Indice),
95         param(Element, Vecteur)
96     do
97         X is Vecteur[I],

```

```
98         NouvelleValeur #= (Element #= X) * I + AncienneValeur *  
          (Element #\= X)  
99     ).
```