Devoir 3 – Programmation par contraintes

Solutionnaire

Question 1 – À la découverte de contraintes globales

1. `at_most(int: n, array [int] of var int: x, int: v)`

   La contrainte en question impose que, pour une liste x de variables entières, ainsi que deux entiers n et v, la valeur v n'apparaisse pas plus de n fois dans la liste x. Alors, v est une valeur susceptible de se trouver dans la liste x.

   Exemple : Dans un problème d'affectation d'étudiants à un collège ayant une capacité de n, x est une variable égale à 0 si l'étudiant n'est pas affecté à ce collège et à 1 s'il l'est. Pour respecter la capacité, on utilise la contrainte `at_most(n, x, 1)`.

2. `count(array[int] of var int: x, var int: y, var int: c)`

   Soit x une liste de variables entières et y et c des variables entières. y est une valeur susceptible de se trouver dans la liste x. Cette contrainte globale impose que le nombre d'occurrences de y dans la liste x soit égal à c, ce qui peut s'écrire mathématiquement :

   $$c = |\{v \in x \mid v = y\}|$$

   Exemple : Dans un problème de sac à dos, la liste x est indexée par les items que l'on peut emporter et $x[i]$ est la quantité de l'item i que l'on emporte. Si l'on fixe y à 0, la contrainte count impose que la variable c soit le nombre d'items que l'on emporte pas du tout. Si par exemple on souhaite avoir au moins un exemplaire de chaque item dans le sac, on peut imposer que c soit nulle.

3. `global_cardinality(array[int] of var int: x, array[int] of int: cover, array[int] of var int: counts)`

   x, *cover* et *counts* sont des listes de variables entières. Les éléments de *counts* sont des valeurs susceptibles de se trouver dans la liste x. Cette contrainte impose que pour chaque élément de *cover* se trouvant en position i dans cette liste, le nombre d'occurrences de cet élément dans la liste x soit égal au $i^{\text{ème}}$ élément de *counts*.

   On peut l'écrire mathématiquement sous la forme :

   $$\forall i \in \{1, \ldots, |cover|\}, \quad count_i = |\{v \in x \mid v = cover_i\}|$$

   Exemple : Dans l'exercice 2, on se sert de cette contrainte : x est la liste des chiffres d'un code, *cover* est l'ensemble des valeurs qu'ils peuvent prendre et *counts* le nombre d'occurrences de chaque valeur dans le code. Cette contrainte permet d'imposer que chaque valeur apparaît au plus 2 fois grâce à la définition que l'on fait préalablement de *counts*.

4. `cumulative(array[int] of var int: s,array[int] of var int: d, array[int] of var int: r, var int: b)`

On considère que l'on a un ensemble de tâches à effectuer. s, d et r sont des listes de variables entières donnant respectivement la date de début de chaque tâche, sa durée et la quantité de ressource qu'elle nécessite. b est une variable entière. Cette contrainte impose qu'à tout moment dans le processus, la somme des demandes en ressource des tâches en train d'être effectuées n'excède pas b.

Exemple : On souhaite préparer un très grand gâteau au chocolat. Les tâches sont les étapes de la recette. Certaines tâches peuvent être faites en même temps (couper le chocolat et battre les blancs en neige), mais il y a un ordre à respecter (préchauffer le four avant d'enfourner le gâteau). Les tâches sont donc ordonnées et se superposent par moments. Notre ressource est la main d'œuvre. Comme c'est un très grand gâteau, il faut souvent plusieurs personnes sur chaque tâche. s contient les dates de début de chaque tâche, d leur durée et r le nombre de personnes nécessaires pour les effectuer. On a un nombre fixe de cuisiniers et cuisinières égal à b. La contrainte cumulative permet de s'assurer que notre recette ne prévoit à aucun moment d'avoir besoin de plus de main d'œuvre que ce dont on dispose.

1 Question 2 – Un code secret

1. **Modélisez ce problème en utilisant la programmation par contraintes. Formalisez mathématiquement les données, les variables de décision, la fonction objectif ainsi que les contraintes.**

Données :
— $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$: ensemble des valeurs possibles pour les chiffres du code
— $C = \{N, B, V\}$: ensemble des couleurs possibles pour les chiffres du code
— $n = 5$: nombre de chiffres dans le code

Variables de décision :
— $x_i \in X$, $1 \leq i \leq n$: valeur du chiffre en position i dans le code
— $c_i \in C$, $1 \leq i \leq n$: couleur du chiffre en position i dans le code

Contraintes :

$$x_i \in X \qquad \forall i \in \{1, \ldots, n\} \tag{1}$$
$$c_i \in C \qquad \forall i \in \{1, \ldots, n\} \tag{2}$$
$$x_i \leq x_{i+1} \qquad \forall i \in \{1, \ldots, n-1\} \tag{3}$$
$$x_i \neq 5 \Rightarrow c_i \in \{N, B\} \qquad \forall i \in \{1, \ldots, n\} \tag{4}$$
$$x_i = 5 \Rightarrow c_i = V \qquad \forall i \in \{1, \ldots, n\} \tag{5}$$
$$x_i = x_{i+1} \text{ et } x_i \neq 5 \Rightarrow c_i = N \text{ et } c_{i+1} = B \qquad \forall i \in \{1, \ldots, n-1\} \tag{6}$$
$$|\{i \in \{1, \ldots, n\} \mid x_i = k\}| \leq 2 \qquad \forall k \in X \tag{7}$$
$$|\{i \in \{1, \ldots, n-1\} \mid x_i = x_{i+1}\}| \geq 1 \tag{8}$$
$$c_1 = c_2 = c_3 \tag{9}$$
$$c_i = c_{i+1} \qquad \forall i \in \{3, \ldots, n-1\} \tag{10}$$
$$x_{n-2} + x_{n-1} + x_n = 7 \tag{11}$$

— Contraintes (1) : la valeur d'un chiffre du code est dans l'ensemble X
— Contraintes (2) : la couleur d'un chiffre du code est dans l'ensemble C
— Contraintes (3) : les valeurs sont en ordre croissant

— Contraintes (4) : la couleur d'un chiffre autre que 5 est noir ou blanc
— Contraintes (5) : la couleur d'un 5 est vert
— Contraintes (6) : si un chiffre d'une valeur autre que 5 est présent deux fois, le premier est noir et le second est blanc
— Contraintes (7) : chaque valeur apparaît au plus deux fois
— Contrainte (8) : deux chiffres du code ont la même valeur
— Contraintes (9) et (10) : les trois premiers chiffres du code sont les seuls chiffres adjacents ayant la même couleur
— Contrainte (11) : la somme des valeurs des trois derniers chiffres vaut 7

Fonction objectif : Ici, il n'y a pas de fonction objectif ; on cherche simplement une solution qui satisfasse toutes les contraintes.

2. **Modélisez ce problème sur MiniZinc. Pouvez-vous désamorcer la bombe à coup sûr ? Si non, quelle est votre chance de succès ?**

   Il n'y a qu'une seule solution possible, c'est le code 0N,1N,2N,2B,3N. On peut donc désamorcer la bombe à coup sûr.

3. **Vous réalisez qu'il y a une seconde bombe à désamorcer. Cette deuxième bombe suit également la logique expliquée dans les points 1 à 4. De plus, vous obtenez cette fois-ci les renseignements suivants. Pouvez-vous désamorcer la bombe à coup sûr ? Si non, quelle est votre chance de succès ?**

   Contraintes :

$$(1) - (7)$$

$$x_2 = 2 \tag{12}$$

$$\sum_{i \in \{1,\dots,n\}: c_i = N} x_i = 3 \tag{13}$$

$$\sum_{i \in \{1,\dots,n\}: c_i = B} x_i = 11 \tag{14}$$

$$(x_i \neq 0 \vee c_i \neq N) \wedge (x_i \neq 1 \vee c_i \neq B) \wedge (x_i \neq 7 \vee c_i \neq N) \wedge (x_i \neq 9) \quad \forall i \in \{1, \dots, n\} \tag{15}$$

   — Contrainte (12) : il y a un 2 en deuxième position du code
   — Contrainte (13) : la somme des valeurs des chiffres noirs du code vaut 3
   — Contrainte (14) : la somme des valeurs des chiffres blancs du code vaut 11
   — Contraintes (15) : il n'y a pas de 0 noir, de 1 blanc, de 7 noir ni de 9 dans le code

   Cette fois il y a 4 solutions possibles : 1N,2N,2B,3B,6B, 1N,2N,3B,5V,8B, 1N,2N,4B,5V,7B, et 0B,2B,3N,3B,6B. On a donc 1 chance sur 4 de réussir à désamorcer la bombe.

Question 3 – Ordonnancement avec ressources multiples

1. **Modélisez ce problème en utilisant la programmation par contraintes. Formalisez mathématiquement les données, les variables de décisions, la fonction objectif ainsi que les contraintes.**

Données :
— $\mathcal{R}$: ensemble des ressources (r)
— $\mathcal{I}$: ensemble des tâches (i ou j)
— c_r : capacité de la ressource r
— d_i : durée de la tâche i
— a_i^r : quantité de ressource r demandée par la tâche i
— $\mathcal{S}_i \subseteq \mathcal{I}$: ensemble de tâches qui doivent être faites après la fin de la tâche i
— $\mathcal{T} = \{0, \ldots, \sum_{i \in \mathcal{I}} d_i\}$: périodes de temps

Variables :
— $s_i \in \mathcal{T}$: date de début de la tâche i
— $z \in \mathcal{T}$: makespan

Contraintes :

$$s_i + d_i \leq s_j \qquad \forall i \in \mathcal{I}, j \in \mathcal{S}_i \tag{16}$$

$$\sum_{i \in \mathcal{I}} (a_i^r \cdot (t \geq s_i \wedge t < s_i + d_i)) \leq c_r \qquad \forall r \in \mathcal{R}, t \in \mathcal{T} \tag{17}$$

$$z \geq s_i + d_i \qquad \forall i \in \mathcal{I} \tag{18}$$

$$s_i \in \mathcal{T} \qquad \forall i \in \mathcal{I} \tag{19}$$

$$z \in \mathcal{T} \tag{20}$$

Fonction objectif :

$$\min z \tag{21}$$

2. **Modélisez ce problème sur MiniZinc, en utilisant au moins une contrainte globale. Définissez les données comme indiqué dans les fichiers `boulangerie1.dzn`, `boulangerie2.dzn` et `boulangerie3.dzn` fournis. Chaque fichier représente une instance différente du problème.**

   Regardez le fichier `Devoir3_Q3.mzn`.

3. **Résolvez les trois instances en utilisant les solveurs *Gecode* ou *Chuffed*. Reportez les solutions des trois situations dans le rapport (temps de début et coût optimal). Pour vous aider à corriger vos modèles, la solution optimale de la première situation vous est donnée (première ligne du fichier `boulangerie1.dzn`).**

   Makespan Boulangerie 1 = 63

   Makespan Boulangerie 2 = 60

   Makespan Boulangerie 3 = 120

   Les dates de début pour chaque instance sont montrées dans les tableaux 1, 2 et 3.

Tâche	Début	Tâche	Début	Tâche	Début
1	0	11	43	21	6
2	10	12	25	22	38
3	0	13	32	23	42
4	14	14	32	24	56
5	1	15	15	25	50
6	17	16	35	26	50
7	14	17	38	27	50
8	38	18	14	28	57
9	17	19	36	29	23
10	17	20	25	30	57

TABLE 1 – Date de début de chaque tâche : Boulangerie 1

Tâche	Début	Tâche	Début	Tâche	Début
1	0	11	23	21	8
2	0	12	2	22	30
3	16	13	21	23	30
4	9	14	21	24	20
5	21	15	0	25	29
6	0	16	27	26	16
7	15	17	15	27	33
8	3	18	15	28	22
9	9	19	0	29	27
10	15	20	8	30	8
31	29	41	34	51	48
32	32	42	15	52	50
33	0	43	38	53	39
34	30	44	32	54	31
35	16	45	41	55	41
36	13	46	42	56	47
37	17	47	15	57	45
38	38	48	38	58	47
39	31	49	41	59	53
40	33	50	10	60	51

TABLE 2 – Date de début de chaque tâche : Boulangerie 2

4. **Finalement, reportez les statistiques de recherche (nodes, failures, solveTime et peakDepth) des trois instances pour les solveurs *Gecode* et *Chuffed*. Comparez les performances de *Chuffed* et *Gecode*. Cette question a pour objectif de vous sensibiliser aux différences qu'il existe entre les solveurs. (Les statistiques de résolution sont obtenues avec *Show configuration editor* > *Options* > *Output* > *Output solving statistics*).**

   — Les résultats se trouvent dans les tableaux 4, 5 et 6.
   — En considérant la Boulangerie 1, on constate que le solveur *Chuffed* présente une meilleure performance que le solveur *Gecode*. En effet, le temps de résolution de *Chuffed*, ainsi que le nombre de *failures* et le nombre de *nodes*, est considérablement inférieur à celui de *Gecode*.

Tâche	Début	Tâche	Début	Tâche	Début
1	0	31	37	61	67
2	0	32	44	62	98
3	17	33	9	63	61
4	9	34	45	64	40
5	23	35	34	65	73
6	14	36	22	66	70
7	0	37	35	67	46
8	2	38	44	68	89
9	3	39	22	69	82
10	22	40	15	70	98
11	4	41	44	71	82
12	12	42	54	72	67
13	26	43	82	73	67
14	17	44	45	74	101
15	24	45	64	75	73
16	38	46	54	76	81
17	34	47	35	77	102
18	32	48	50	78	90
19	53	49	23	79	62
20	2	50	54	80	98
21	34	51	62	81	103
22	24	52	61	82	64
23	14	53	4	83	109
24	32	54	89	84	103
25	24	55	40	85	70
26	44	56	80	86	72
27	31	57	63	87	118
28	33	58	64	88	119
29	73	59	76	89	108
30	44	60	54	90	111

TABLE 3 – Date de début de chaque tâche : Boulangerie 3

De plus, la profondeur est également plus faible, ce qui renforce cette analyse.

— En ce qui concerne la Boulangerie 2, le solveur *Gecode* surpasse le solveur *Chuffed*. Le temps de résolution de Gecode, ainsi que le nombre de *nodes* et l3 *peakDepth*, sont considérablement inférieurs à ceux de *Chuffed*.

— Pour la Boulangerie 3, les deux solveurs présentent des performances similaires. Bien que *Chuffed* ait un nombre de *failures* et de *nodes* inférieur, le *peakDepth* est supérieure à celle de *Gecode*. Le temps d'exécution est très comparable, avec une différence d' d'approximativement seulement 1 seconde.

— Nous pouvons voir que les performances des solveurs varient en fonction de l'instance.

	Gecode	*Chuffed*
failures	3,147,694	808
nodes	6,295,432	5,776
peakDepth	71	67
solveTime	46.2595	0.201

TABLE 4 – Performances Boulangerie 1

	Gecode	*Chuffed*
failures	6	0
nodes	35	20,400
peakDepth	33	76
solveTime	0.0012167	1.893

TABLE 5 – Performances Boulangerie 2

	Gecode	*Chuffed*
failures	72,147	2,230
nodes	144,361	80,484
peakDepth	83	1,078
solveTime	1.44047	2.653

TABLE 6 – Performances Boulangerie 3