



Organisation d'un championnat

Projet 01 d'Intelligence Artificielle et Manipulation Symbolique de l'Information

15 février 2024

Table des matières

1	MODÉLISATION	3
1.1	Notations utilisées	3
1.2	Encodage et Décodage	3
2	Génération d'un planning de matchs	4
2.1	Contraintes de cardinalité	4
2.2	Contrainte C_1	4
2.3	Contrainte C_2	5
2.4	Résolution du programme	5
3	OPTIMISATION DU NOMBRE DE JOURS	7
4	ÉQUILIBRER LES DÉPLACEMENTS ET LES WEEK-ENDS	7
4.1	Contrainte C_3	7
4.2	Contrainte C_4	7
4.3	Résolution du programme étendu	7
5	CONCLUSION	7

INTRODUCTION

L'objectif de ce projet est d'organiser la grille de match d'un championnat, entre n_e équipes participantes, se déroulant sur au plus $n_s = \frac{n_j}{2}$ semaines où n_j est le nombre de jours de matchs.

On suppose de plus qu'une semaine de matchs ne contient que deux jours de matchs : le mercredi et le dimanche, et qu'un championnat commence systématiquement le mercredi.

Par ailleurs, les noms des équipes participantes sont présents dans le fichier `equipes.txt`.

1 MODÉLISATION

1.1 Notations utilisées

Pour résoudre le problème, nous utilisons le solveur SAT Glucose, basé sur Minisat 2.2, qui utilise le format DIMACS où toutes les variables sont numérotées. C'est pourquoi, dans la suite, on numérote une équipe x par un nombre compris entre 0 et $n_e - 1$. De même, un jour est compris entre 0 et $n_j - 1$.

On note $m_{j,x,y}$ la variable booléenne permettant de déterminer s'il y a un match le jour j entre l'équipe x à domicile, et l'équipe y à l'extérieur. Par conséquent, il y a $(n_j - 1) \times n_e^2 + (n_e - 1) \times n_e + n_e - 1 + 1 = n_j \times n_e^2$ variables différentes.

1.2 Encodage et Décodage

On définit la suite $(v_i) \in \{0, 1\}^{\mathbf{N}^*}$ afin d'indicer séquentiellement les $m_{j,x,y}$. Pour ce faire, on code les triplets (j, x, y) en base n_e . Ainsi, v_k correspond à $m_{j,x,y}$ si et seulement si $k = j \times n_e^2 + x \times n_e + y + 1$.

```
1 def codage(ne, nj, j, x, y):
2     assert(ne > 0)
3     assert(j <= nj-1 and j >= 0)
4     assert(x <= ne-1 and x >= 0)
5     assert(y <= ne-1 and y >= 0)
6     return j*ne**2 + x*ne + y + 1
```

Pour s'assurer de la robustesse de l'algorithme, dans cette fonction comme dans les prochaines, on s'assure également de respecter les bonnes plages de valeurs.

En ce qui concerne le décodage, on peut obtenir (j, x, y) à partir des divisions successives de $k - 1$ par n_e . En effet, on a :

$$\begin{cases} y \equiv (k - 1) \pmod{n_e} \\ x \equiv \lfloor \frac{(k-1)}{n_e} \rfloor \pmod{n_e} \\ j \equiv \lfloor \frac{(k-1)}{n_e^2} \rfloor \end{cases}$$

```
1 def decodage(k, ne):
2     assert(ne > 0)
3     y = (k-1) % ne
4     x = ((k-1) // ne) % ne
5     j = (k-1) // ne**2
6     return j, x, y
```

Afin de vérifier si les fonctions étaient correctes, nous avons commencé par comparer les valeurs de codage avec celles que l'on obtenait à la main :

```
1 assert(codage(3, 2, 0, 0, 0) == 1)
2 assert(codage(3, 2, 1, 2, 1) == 17)
3 assert(codage(3, 2, 1, 0, 0) == 10)
```

Nous avons ensuite testé la fonction de décodage grâce à celle de codage en comparant le triplet en sortie, de celui mis en entrée de la fonction de codage.

```

1 assert(decodage(codage(3, 3, 2, 0, 0), 3) == (2, 0, 0))
2 assert(decodage(codage(3, 6, 5, 2, 1), 3) == (5, 2, 1))

```

2 Génération d'un planning de matchs

2.1 Contraintes de cardinalité

Dans le cas de la contrainte `au_moins`, la traduction est simple puisqu'il s'agit d'une simple disjonction. Sous le format DIMACS, cela revient à écrire une ligne où tous les indices des variables concernées sont présents. Nous avons néanmoins fait attention à ce que toutes les indices de la liste considérée soient strictement positifs.

```

1 def au_moins(L):
2     assert(not any([v <= 0 for v in L]))
3     if len(L) == 0:
4         return ''
5     return ' '.join([str(v) for v in L]) + ' 0\n'

```

Le cas de la contrainte `au_plus` est un peu plus compliqué. Pour le traiter, nous avons utilisé l'encodage par paires : le principe est que pour toutes les paires possibles (v_i, v_j) , on ait une contrainte `-i -j 0`, c'est-à-dire que $v_i = 0$ ou $v_j = 0$. Ainsi, le cas $\forall i, v_i = 0$ est satisfait, et s'il existe i_0 tel que $v_{i_0} = 1$ alors pour satisfaire les autres contraintes, on a : $\forall i \neq i_0, v_i = 0$.

```

1 def au_plus(L):
2     assert(not any([v <= 0 for v in L]))
3     contrainte = ''
4     for i in range(len(L)):
5         for j in range(i+1, len(L)):
6             contrainte += f'{-L[i]} {-L[j]} 0\n'
7     return contrainte

```

Comme il peut y avoir rapidement de nombreuses contraintes, notamment dans le cas de `au_plus`, nos tests ont été assez succincts, comparant la sortie des fonctions à celle que nous attendions.

```

1 assert(au_moins([]) == '')
2 assert(au_moins([1, 2, 3]) == '1 2 3 0\n')
3 assert(au_plus([]) == '')
4 assert(au_plus([1, 2, 3]) == '-1 -2 0\n-1 -3 0\n-2 -3 0\n')

```

2.2 Contrainte C_1

Mathématiquement, la contrainte C_1 : "Chaque équipe ne peut jouer plus d'un match par jour" correspond à :

$$\forall x \in \llbracket 0, n_e - 1 \rrbracket, \forall j \in \llbracket 0, n_j - 1 \rrbracket, \sum_{y \neq x} m_{j,x,y} \leq 1$$

Pour l'implémenter, il s'agit donc d'appliquer la règle `au_plus` sur les bons triplets. Néanmoins, dans le cas de C_1 , quelques contraintes apparaissent en doublon car entre deux équipes x_1 et x_2 différentes, les paires (x_1, x_2) et (x_2, x_1) sont communes.

```

1 def encoderC1(ne, nj):
2     assert(ne > 0 and nj > 0)
3     C1 = ''
4     for j in range(nj):
5         for x in range(ne):
6             mjxy = [codage(ne, nj, j, x, y) for y in range(ne) if y != x] + [codage(ne, nj, j, y, x) for y in range(ne) if y != x]
7             C1 += au_plus(mjxy)
8     return C1, len(C1.split('\n'))-1

```

En particulier, pour $n_e = 3, n_j = 4$, notre programme génère 72 contraintes.

2.3 Contrainte C_2

La contrainte C_2 : "Sur la durée du championnat, chaque équipe doit rencontrer l'ensemble des autres équipes une fois à domicile et une fois à l'extérieur, soit exactement 2 matchs par équipe adverse" correspond à :

$$\forall x \in \llbracket 0, n_e - 1 \rrbracket, \forall y > x, \sum_j m_{j,x,y} = 1 \text{ et } \sum_j m_{j,y,x} = 1$$

Cela correspond donc à quatre contraintes de cardinalité (une égalité correspond à `au_moins` et `au_plus`) pour chaque paire d'équipe (x, y) .

```
1 def encoderC2(ne, nj):
2     assert(ne > 0 and nj > 0)
3     C2 = ''
4     for x in range(ne):
5         for y in range(x+1, ne):
6             mjxy = [codage(ne, nj, j, x, y) for j in range(nj)]
7             mjyx = [codage(ne, nj, j, y, x) for j in range(nj)]
8             C2 += au_moins(mjxy) + au_plus(mjxy)
9             C2 += au_moins(mjyx) + au_plus(mjyx)
10    return C2, len(C2.split('\n'))-1
```

Cette implémentation produit 42 contraintes dans le cas $n_e = 3, n_j = 4$, desquelles il ne devrait pas y avoir de doublons puisque les paires (x, y) sont disjointes pour $y > x$.

2.4 Résolution du programme

On remarque cependant que les contraintes C_1 et C_2 ne suffisent pas puisque les équipes les variables de la forme $m_{j,x,x}$ sont manquantes. On ajoute donc une contrainte C_0 : "une équipe ne peut pas s'affronter elle-même" :

```
1 def encoderC0(ne, nj):
2     assert(ne > 0 and nj > 0)
3     L = [codage(ne, nj, j, x, x) for x in range(ne) for j in range(nj)]
4     C0 = ' 0\n'.join([str(-v) for v in L]) + ' 0\n'
5     return C0, len(C0.split('\n'))-1
```

Il faut ensuite encoder les contraintes C_0, C_1, C_2 ensemble au format DIMACS. Pour cela, on écrit la fonction `encoder` dont la première ligne est `p cnf {nb_variables} {nb_contraintes}`, et dont les lignes suivantes sont les lignes des contraintes à encoder.

```
1 def encoder(ne, nj):
2     assert(ne > 0 and nj > 0)
3     C0, n0 = encoderC0(ne, nj)
4     C1, n1 = encoderC1(ne, nj)
5     C2, n2 = encoderC2(ne, nj)
6     return f'p cnf {nj*ne**2} {n0+n1+n2}\n{C0}{C1}{C2}'
```

Pour pouvoir par la suite décoder la solution retournée par glucose, il nous faut pouvoir retrouver le nom des équipes à partir de leur indice. On forme pour cela un dictionnaire traduisant les noms et encodages :

```
1 def dico_equipements(filename='equipements.txt'):
2     dico = {}
3     with open(filename, 'r') as f:
4         lignes = f.readlines()
5         for i, ligne in enumerate(lignes):
6             dico[i] = ligne
7     return dico
```

Pour traduire le résultat de glucose, il nous reste à écrire une fonction parcourant les lignes de ce résultat. On sait si l'algorithme est *SAT* ou *UNSAT* grâce à la ligne débutant par *s*. S'il est *UNSAT*, nul besoin de continuer. Sinon, on poursuit jusqu'à la ligne débutant par *v* qui donne pour chaque variable son interprétation booléenne (négatif : 0, positif : 1).

En utilisant notre dictionnaire précédemment construit et notre fonction de décodage, on obtient ainsi un jour de match et les deux équipes qui s'affrontent (dans l'ordre : équipe à domicile, équipe à l'extérieur).

```

1 def decoder(ne, equipes):
2     equipes = dico_equipements(equipements)
3     solution = ''
4     with open('resultats.txt', 'r') as f:
5         lignes = f.readlines()
6         for ligne in lignes:
7             tokens = ligne.split()
8             if len(tokens) == 0:
9                 continue
10            if tokens[0] == 's' and tokens[1] == 'UNSATISFIABLE':
11                return 'UNSAT'
12            elif tokens[0] == 'v':
13                for v in tokens[1:]:
14                    if int(v) > 0:
15                        j, x, y = decodage(int(v), ne)
16                        solution += f'Jour {j} : {equipes[x][: -1]} (dom.) vs {equipes[
17y][: -1]} (ext.)\n'
18            return solution

```

Il nous reste à assembler le tout : en entrée de `./glucose -model` – le `-model` est important pour obtenir une solution du problème – on passe la sortie de la fonction `encoder`, avant d'écrire la sortie de glucose dans un fichier `resultats.txt`, en l'écrasant si nécessaire.

```

1 def programme(ne, nj, equipes):
2     os.system(f'echo "{encoder(ne, nj)}" | ../glucose/simp/glucose -model > resultats.
3     txt')
4     return decoder(ne, equipes)

```

Si le problème est *UNSAT* pour $n_e = 3, n_j = 4$, on obtient une solution pour $n_j = 6$:

- Jour 0 : Boston Celtics (dom.) vs Atlanta Hawks (ext.)
- Jour 1 : Brooklyn Nets (dom.) vs Boston Celtics (ext.)
- Jour 2 : Atlanta Hawks (dom.) vs Boston Celtics (ext.)
- Jour 3 : Boston Celtics (dom.) vs Brooklyn Nets (ext.)
- Jour 4 : Brooklyn Nets (dom.) vs Atlanta Hawks (ext.)
- Jour 5 : Atlanta Hawks (dom.) vs Brooklyn Nets (ext.)

3 OPTIMISATION DU NOMBRE DE JOURS

4 ÉQUILIBRER LES DÉPLACEMENTS ET LES WEEK-ENDS

4.1 Contrainte C_3

4.2 Contrainte C_4

4.3 Résolution du programme étendu

5 CONCLUSION