Organisation d'un championnat

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

15 février 2024

LUONG Ethan 28627904
PHAM Louis-Antoine 21304385

Table des matières

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.

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 $k = j \times n_e^2 + x \times n_e + y + 1$.

```python
def codage(ne, nj, j, x, y):
    assert(ne > 0)
    assert(j <= nj-1 and j >= 0)
    assert(x <= ne-1 and x >= 0)
    assert(x <= ne-1 and x >= 0)
    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 :

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

```python
def decodage(k, ne):
    assert(ne > 0)
    y = (k-1) % ne
    x = ((k-1) // ne) % ne
    j = ((k-1) // ne**2) % ne
    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 :

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

```python
assert(decodage(codage(3, 3, 2, 0, 0), 3) == (2, 0, 0))
assert(decodage(codage(3, 3, 1, 2, 1), 3) == (1, 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.

```python
def au_moins(L):
    assert(not any([v <= 0 for v in L]))
    if len(L) == 0:
        return ''
    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 : $\forall i \neq i_0, v_i = 0$.

```python
def au_plus(L):
    assert(not any([v <= 0 for v in L]))
    contrainte = ''
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            contrainte += f'{-L[i]} {-L[j]} 0\n'
    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.

```python
assert(au_moins([]) == '')
assert(au_moins([1, 2, 3]) == '1 2 3 0\n')
assert(au_plus([]) == '')
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 :

$$\forall x \in [\![0, n_e - 1]\!], \forall j \in [\![0, n_j - 1]\!], \sum_{y \neq x} m_{j,x,y} \leq 1$$

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.

```python
def encoderC1(ne, nj):
    assert(ne > 0 and nj > 0)
    C1 = ''
    for j in range(nj):
        for x in range(ne):
            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]
            C1 += au_plus(mjxy)
    return C1
```

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 :

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

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

```python
def encoderC2(ne, nj):
    assert(ne > 0 and nj > 0)
    C2 = ''
    for x in range(ne):
        for y in range(x+1, ne):
            mjxy = [codage(ne, nj, j, x, y) for j in range(nj)]
            mjyx = [codage(ne, nj, j, y, x) for j in range(nj)]
            C2 += au_moins(mjxy) + au_plus(mjxy)
            C2 += au_moins(mjyx) + au_plus(mjyx)
    return C2
```

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

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