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.

L'ensemble du code est présent dans le fichier `projet.py`, et les tests peuvent être réalisés pas-à-pas dans `notebook.ipynb`. Par ailleurs, les noms des équipes participantes sont présents dans le fichier `equipes.txt` et font références aux équipes de basket-ball de la NBA.

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

```
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 :

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

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

```
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.

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

```
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 : $\forall i \neq i_0, v_i = 0$.

```
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.

```
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.

```
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, 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 :

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

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

```
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, 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" :

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

```
def encoder(ne, nj):
    assert(ne > 0 and nj > 0)
    C0, n0 = encoderC0(ne, nj)
    C1, n1 = encoderC1(ne, nj)
    C2, n2 = encoderC2(ne, nj)
    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 :

```python
def dico_equipes(filename='equipes.txt'):
    dico = {}
    with open(filename, 'r') as f:
        lignes = f.readlines()
        for i, ligne in enumerate(lignes):
            dico[i] = ligne
        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).

```python
def decoder(ne, equipes):
    equipes = dico_equipes(equipes)
    solution = ''
    with open('resultats.txt', 'r') as f:
        lignes = f.readlines()
        for ligne in lignes:
            tokens = ligne.split()
            if len(tokens) == 0:
                continue
            if tokens[0] == 's' and tokens[1] == 'UNSATISFIABLE':
                return 'UNSAT'
            elif tokens[0] == 'v':
                for v in tokens[1:]:
                    if int(v) > 0:
                        j, x, y = decodage(int(v), ne)
                        solution += f'Jour {j} : {equipes[x][:-1]} (dom.) vs {equipes[
    y][:-1]} (ext.)\n'
    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.

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

Si le problème est *UNSAT* pour $n_e = 3, n_j = 4$ (car on a 3 équipes qui s'affrontent, soit 6 matchs au total, mais comme une équipe ne peut participer qu'à un match par jour, on a au plus un match par jour), 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

L'objectif de cette section est de déterminer le nombre de jours minimum $n_{j,min}$ pour lequel le programme est résoluble (ou déclenche un timeout), pour $n_e \in [\![3, 10]\!]$. Pour gérer la notion de timeout, on utilise le package `signal`.

Pour éviter d'itérer en vain, et ainsi gagner en temps d'exécution, on peut également restreindre la plage des valeurs possibles de n_j. Dans un premier temps, en vertu de la contrainte C_1, on sait que le nombre maximum de matchs en une journée est $\lfloor \frac{n_e}{2} \rfloor$. Le nombre de matchs total étant de $n_e \times (n_e - 1)$, on en déduit :

$$\frac{n_j \times n_e}{2} \geq n_e \times (n_e - 1) \Rightarrow n_{j,min} = 2 \times (n_e - 1)$$

On peut aussi donner une valeur de n_j telle que le programme est satisfait. En effet, si l'on planifie un match entre deux équipes distinctes par jour de championnat, on arrive finalement à respecter à la fois C_1 et C_2. D'où :

$$n_j \leq n_e \times (n_e - 1)$$

Pour gagner en temps d'exécution, on procède donc par dichotomie entre ces deux bornes. On en arrive à la fonction suivante :

```python
def optimiser_nj_selon_ne(equipes, ne_min=3, ne_max=10, timeout=10):
    for ne in range(ne_min, ne_max+1):
        nj_min = 2*(ne-1)
        nj_max = ne*(ne-1)
        nj_inf = nj_min
        while nj_min <= nj_max:
            nj = (nj_max + nj_min) // 2
            signal.signal(signal.SIGALRM, signal_handler)
            signal.alarm(timeout)
            try:
                solution = programme(ne, nj, equipes)
            except TimeoutError:
                solution = 'Timeout'
            finally:
                signal.alarm(0)
                if solution != 'UNSAT':
                    has_timeout = solution == 'Timeout'
                    nj_max = nj - 1
                    nj_inf = nj
                else:
                    nj_min = nj + 1
        print(f'Pour ne = {ne}, on a nj_inf = {nj_inf} ({has_timeout and "Timeout" or "OK"})')
```

Cette fonction nous permet par exemple de déterminer $n_j = 4$ pour $n_e = 3$ ou $n_e = 4$.

4 Équilibrer les déplacements et les week-ends

4.1 Contraintes de cardinalité étendues

Pour les contraintes de type "au plus" : $\sum_{v \in L} v \leq k$, plutôt que d'encoder par paires, on encode par combinaisons de k élements parmi $|L|$, notées C_n^k.

```python
def au_plus_etendu(L, k):
    contrainte = ''
    combinaisons = list(combinations(L, k+1))
    for sous_liste in combinaisons:
        contrainte += ' '.join([str(-v) for v in sous_liste]) + ' 0\n'
    return contrainte
```

Dans le cas des contraintes de type "au moins" : $\sum_{v \in L} v \geq k$, il suffit de se ramener à une contrainte de type "au plus" : $\sum_{v \in L} -v \leq k$. On a alors :

```python
def au_moins_etendu(L, k):
    N = len(L)
    return au_plus_etendu([-v for v in L], N-k)
```

4.2 Contrainte C_3

La contrainte C_3 donnée dans l'énoncé correspond à deux sous-contraintes :

$$\begin{cases} \forall x, \forall y \neq x, \sum_{j \text{ impair}} m_{j,y,x} \geq \lceil p_{ext} n_j n_e^2 \rceil \\ \forall x, \forall y \neq x, \sum_{j \text{ impair}} m_{j,x,y} \geq \lceil p_{dom} n_j n_e^2 \rceil \end{cases}$$

On l'encode de la façon suivante :

```python
def encoderC3(ne, nj, p_ext=0.5, p_dom=0.4):
    assert(ne > 0 and nj > 0)
    assert(p_ext >= 0 and p_ext <= 1)
    assert(p_dom >= 0 and p_dom <= 1)
    C3 = ''
    for x in range(ne):
        L_ext = [codage(ne, nj, dimanche, y, x) for y in range(ne) if y != x for
    dimanche in range(1, nj, 2)]
        C3 += au_moins_etendu(L_ext, math.ceil(p_ext*(ne-1)))
        L_dom = [codage(ne, nj, dimanche, x, y) for y in range(ne) if y != x for
    dimanche in range(1, nj, 2)]
        C3 += au_moins_etendu(L_dom, math.ceil(p_dom*(ne-1)))
```

4.3 Contrainte C_4

Du côté de la contrainte C_4, cette dernière équivaut à deux sous-contraintes :

$$\begin{cases} \forall x, \forall y \neq x, \forall j \in [\![0, n_j - 2]\!], m_{j,y,x} + m_{j+1,y,x} + m_{j+2,y,x} \leq 2 \\ \forall x, \forall y \neq x, \forall j \in [\![0, n_j - 2]\!], m_{j,x,y} + m_{j+1,x,y} + m_{j+2,x,y} \leq 2 \end{cases}$$

Cela revient à faire passer une fenêtre de taille 3 sur les matchs d'une équipe x sur des jours consécutifs, et de s'assurer qu'au plus 2 sont à 1. On l'encode de la façon suivante :

```python
def encoderC4(ne, nj):
    assert(ne > 0 and nj > 0)
    C4 = ''
    for x in range(ne):
        for j in range(nj-2):
            L_ext = [codage(ne, nj, j, y, x) for y in range(ne) if y != x] + [codage(
    ne, nj, j+1, y, x) for y in range(ne) if y != x] + [codage(ne, nj, j+2, y, x) for y
     in range(ne) if y != x]
            C4 += au_plus_etendu(L_ext, 2)
            L_dom = [codage(ne, nj, j, x, y) for y in range(ne) if y != x] + [codage(
    ne, nj, j+1, x, y) for y in range(ne) if y != x] + [codage(ne, nj, j+2, x, y) for y
     in range(ne) if y != x]
            C4 += au_plus_etendu(L_dom, 2)
    return C4, len(C4.split('\n'))-1
```

4.4 Résolution du programme étendu

On écrit la fonction `programme_etendu` en procédant de la même façon que pour `programme` :

```python
def programme_etendu(ne, nj, equipes):
    os.system(f'echo "{encoder_etendu(ne, nj)}" | ../glucose/simp/glucose -model >
    resultats.txt')
    return decoder(ne, equipes)
```

où `encoder_etendu` est identique à `encoder` si ce n'est qu'il rajoute les contraintes C_3 et C_4. Par ailleurs, pour $n_e = 3$ et $n_j = 9$, et les valeurs par défaut de p_{dom} et p_{ext}, le programme nous propose la solution suivante :

- Jour 0 : Boston Celtics (dom.) vs Atlanta Hawks (ext.)
- Jour 1 : Boston Celtics (dom.) vs Brooklyn Nets (ext.)
- Jour 3 : Brooklyn Nets (dom.) vs Atlanta Hawks (ext.)
- Jour 4 : Atlanta Hawks (dom.) vs Brooklyn Nets (ext.)
- Jour 5 : Atlanta Hawks (dom.) vs Boston Celtics (ext.)
- Jour 8 : Brooklyn Nets (dom.) vs Boston Celtics (ext.)

5 Conclusion

Pour conclure sur l'organisation du championnat, nous avons constaté que si le problème est résoluble pour n_j suffisamment élevé, l'enjeu principal est de trouver un nombre de jours minimal durant lesquels organiser le championnat. Dans cette continuité, l'ajout de contraintes peut s'avérer décisif et allonger la durée globale du championnat, quitte à ne pas avoir de matchs sur certains jours.