Projet IAMSI

Organisation d'un championnat

Hocine Kadem 21309534

Neil Benahmed 21200977

Date : [14/02/2024]

Table des matières

1 Introduction

Ce rapport présente le projet de développement d'un générateur de championnat en utilisant le langage de programmation Python. L'objectif principal de ce projet est de créer un programme capable de générer un planning de matchs pour un championnat en respectant différentes contraintes. Pour résoudre ce problème complexe, nous avons utilisé la programmation SAT et le solveur Glucose.

Le processus de développement consistait à modéliser le problème en termes de variables propositionnelles, de clauses et de contraintes. Nous avons ensuite traduit cette modélisation en fichiers de format DIMACS, qui contiennent les clauses représentant les contraintes du problème. En utilisant Glucose, un solveur SAT performant, nous avons pu tester la satisfiabilité des clauses générées et obtenir les modèles correspondants.

Dans ce rapport, nous détaillerons les différentes étapes de développement, en mettant l'accent sur la modélisation du problème, la génération des fichiers DIMACS et l'utilisation du solveur Glucose. De plus, nous répondrons en détail aux questions et exercices posés, en fournissant des explications approfondies sur les solutions proposées. Nous présenterons également les résultats obtenus pour diverses instances de test, en analysant la satisfiabilité des clauses et les modèles générés.

2 Modélisation

2.1 Question 1

Pour exprimer le nombre de variables propositionnelles utilisées en fonction de ne (le nombre d'équipes) et nj (le nombre de jours de match), nous devons prendre en compte la représentation des équipes et des jours de match, ainsi que les contraintes spécifiées.

Chaque équipe est représentée par un numéro entre 0 et ne-1, et chaque jour de match est représenté par un numéro entre 0 et nj-1. Selon la proposition de codage présentée, chaque variable propositionnelle mj,x,y représente le fait qu'il y ait (ou non) un match entre l'équipe x, jouant à domicile, et l'équipe y au jour j.

Le nombre total de variables propositionnelles utilisées dépendra du nombre de combinaisons possibles pour les jours de match (nj) et les équipes (ne). Si l'on considère que chaque équipe ne peut pas jouer contre elle-même il y aura **nj * (ne - 1) * ne** variables propositionnelles.

2.2 Question 2

Pour répondre à la question 2, nous devons écrire une fonction de codage.

Voici une implémentation possible de cette fonction de codage en Python :

```
def codage(ne, nj, j, x, y):
    k = ne**2 * j + x * ne + y + 1
    return k
```

La fonction codage prend les arguments ne (nombre d'équipes), nj (nombre de jours de match), j (jour de match), x (équipe jouant à domicile) et y (équipe adverse), et renvoie la valeur de k correspondante.

2.3 Question 3

Pour répondre à la question 3, nous devons écrire une fonction de décodage qui retrouve les valeurs de j, x et y à partir de k et ne.

Voici une implémentation possible de cette fonction de décodage en Python :

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

La fonction decodage prend les arguments k (valeur de la variable propositionnelle codée) et ne (nombre d'équipes), et renvoie les valeurs correspondantes de j (jour de match), x (équipe jouant à domicile) et y (équipe adverse).

Ces fonctions de codage et de décodage permettent de représenter et de manipuler les variables propositionnelles de manière pratique et efficace dans le contexte de la modélisation du générateur de championnat.

3 Génération d'un planning de matchs

3.1 Contraintes de cardinalité

Dans le contexte de l'encodage du problème avec des contraintes de cardinalité "au plus 1" et "au moins 1", nous proposons ici quelques fonctions pour faciliter cet encodage.

3.1.1 Contraintes de cardinalité "au plus 1" et "au moins 1"

Pour répondre à cette contrainte, nous avons écrit une fonction qui prend en argument une liste d'entiers représentant des variables propositionnelles. Cette fonction génère une clause, au format DIMACS, correspondant à la contrainte "au moins une de ces variables est vraie". La clause générée

est de la forme "v1 v2 ... vn 0", où v1, v2, ..., vn sont les variables propositionnelles de la liste. Si la liste est vide, la fonction renvoie une clause vide.

Voici notre implémentation de cette fonction en Python :

```
def cnf_au_moins(liste):
    clause = ""
    for v in liste:
        clause += str(v) + " "
    if clause != "":
        clause += str(0)
    return clause
```

3.1.2 Contraintes de cardinalité "au plus une de ces variables est vraie"

Pour répondre à cette contrainte, nous avons écrit une fonction qui prend en argument une liste d'entiers représentant des variables propositionnelles. Cette fonction génère les clauses, au format DIMACS, correspondant à la contrainte "au plus une de ces variables est vraie". Les clauses générées sont de la forme "-v1 -v2 0", "-v1 -v3 0", ..., "-vn-1 -vn 0", où v1, v2, ..., vn sont les variables propositionnelles de la liste. Chaque clause représente une paire de variables propositionnelles qui ne peuvent pas être toutes les deux vraies. La fonction renvoie toutes les clauses générées sous forme d'une chaîne de caractères.

Voici une implémentation possible de cette fonction en Python :

```
def cnf_au_plus(liste):
    clause = ""
    for i in range(len(liste)):
        for j in range(i + 1, len(liste)):
            clause += "-" + str(liste[i]) + " -" + str(liste[j]) + " 0 \n"
    return clause[:-1]
```

3.2 Traduction du problème

3.2.1 Traduction de la contrainte C1 "chaque équipe ne peut jouer plus d'un match par jour" en un ensemble de contraintes de cardinalité

Pour traduire cette contrainte, nous utilisons des contraintes de cardinalité "au plus un vrai" pour chaque paire de joueurs. Ainsi, pour chaque jour ji et chaque équipe xi donnés, nous avons la traduction suivante :

$$\bigwedge_{j \text{ : jour}; x,y,z \text{ : équipes avec } x \neq y \neq z} \neg(\text{match}(x, y, j) \wedge \text{match}(x, z, j))$$

Cette traduction permet de garantir qu'une équipe ne joue pas plus d'un match par jour.

3.2.2 Fonction encoderC1 pour générer ces contraintes pour ne et nj donnés

Nous avons écrit une fonction qui prend en argument le nombre d'équipes (ne) et le nombre de jours (nj). Cette fonction génère les contraintes correspondant à C1 en utilisant des contraintes de cardinalité "au plus un vrai". Les contraintes sont générées pour chaque équipe et chaque jour. La fonction retourne toutes les clauses générées sous forme d'une chaîne de caractères en éliminant les doublons éventuels en utilisant la fonction eliminer_doublons qu'on a écrit.

Voici une implémentation possible de cette fonction en Python :

```
def encoder_c1(ne, nj):
    clauses = ""
    for xi in range(ne):
        for ji in range(nj):
            liste_x_j = []
            for yi in range(ne):
                if yi != xi:
                    liste_x_j.append(codage(ne, nj, ji, xi, yi))
                    liste_x_j.append(codage(ne, nj, ji, yi, xi))
            clauses += cnf_au_plus(liste_x_j) + "\n"
    return eliminer_doublons(clauses)
```

3.2.3 Nombre de contraintes et de clauses générées par C1 pour 3 équipes sur 4 jours, et explication de ces contraintes

Pour 3 équipes sur 4 jours, nous avons le nombre total de contraintes générées qui est de 60 clauses. Cela inclut les contraintes de cardinalité "au plus un vrai" pour chaque paire de joueurs, garantissant qu'une équipe ne joue pas plus d'un match par jour. Les contraintes sont générées pour chaque équipe et chaque jour.

Chaque jour, nous générons 6 clauses pour chaque équipe, ce qui donne un total de $3 \times 4 \times 6 = 72$ clauses. Cependant, il y a des doublons dus à la symétrie des contraintes, ce qui ajoute 12 doublons. Par conséquent, nous avons finalement 60 clauses uniques qui représentent les contraintes nécessaires pour traduire la contrainte C1 dans le contexte donné.

Ces contraintes de cardinalité permettent de respecter la contrainte C1 du problème, garantissant qu'aucune équipe ne joue plus d'un match par jour.

3.2.4 Traduction de la contrainte C2 : "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" en un ensemble de contraintes de cardinalité

Pour traduire la contrainte C2 en un ensemble de contraintes de cardinalité, nous pouvons utiliser des contraintes de cardinalité "au moins deux vrais" et "au plus deux vrais" pour chaque paire de joueurs.

La traduction se fait de la manière suivante :

$$\bigwedge_{\substack{x,y \text{ : équipes} \\ x \neq y}} ((\sum_{j \text{ : jours}} \text{match}(x,y,j) \geq 1 \wedge (\sum_{j \text{ : jours}} \text{match}(x,y,j) \leq 1 \wedge (\sum_{j \text{ : jours}} \text{match}(y,x,j) \geq 1 \wedge (\sum_{j \text{ : jours}} \text{match}(y,x,j) \leq 1))$$

3.2.5 Fonction encoderC2 pour générer ces contraintes pour ne et nj donnés

Voici une fonction possible pour générer ces contraintes pour un nombre d'équipes (ne) et un nombre de jours (nj) donnés :

```
def encoder_c2(ne, nj):
    clauses = ""
    for xi in range(ne):
        for yi in range(xi + 1, ne):
            matchs_aller = []   # Tous les matchs    domicile de xi avec yi
            matchs_retour = []  # Tous les matchs    l'extrieur de xi avec yi
            for ji in range(nj):
                matchs_aller.append(codage(ne, nj, ji, xi, yi))
                matchs_retour.append(codage(ne, nj, ji, yi, xi))
            # Ajout des clauses
            clauses += cnf_au_moins(matchs_aller) + '\n' + cnf_au_plus(matchs_aller) +
    return eliminer_doublons(clauses)
```

3.2.6 Nombre de contraintes et de clauses générées par C2 pour 3 équipes sur 4 jours, et explication de ces contraintes

Pour 3 équipes sur 4 jours, le nombre de contraintes et de clauses générées est le suivant :

Nous avons 3 équipes (2 parmi 3) xi, yi :

Nous calculons les contraintes "au moins deux vrais" : 1 contrainte par xi, yi.

Nous calculons les contraintes "au plus deux vrais" : Nous avons 4 jours, donc 2 parmi $4 = 6$ contraintes par xi, yi.

Nous multiplions par 2 car nous le faisons pour les matchs à domicile et à l'extérieur. Donc, le nombre total de contraintes (clauses) C2 avec 3 équipes sur 4 jours est :

$3 \times (1 + 6) \times 2 = 42$ contraintes.

3.2.7 Ecriture d'un programme encoder qui encode toutes les contraintes C1 et C2 pour ne et nj donnée

Pour encoder les contraintes C1 et C2 nous avons écrit la fonction suivante :

```
def encoder(ne, nj):
    return eliminer_doublons(encoder_c1(ne, nj) + encoder_c2(ne, nj))
```

Ce programme utilise les fonctions encoder_c1 et encoder_c2 pour générer les contraintes C1 et C2 respectivement. Ensuite, il concatène les contraintes C1 et C2, et utilise la fonction eliminer_doublons pour supprimer les doublons éventuels.

3.2.8 Execution des fonctions et affichage du nombre de contraintes

Afin de faciliter l'analyse et la compréhension des contraintes générées pour un certain nombre d'équipes et de jours, nous avons conçu la fonction affichage_contrainte. Cette fonction a pour objectif d'afficher les informations relatives aux contraintes générées par les fonctions encoder_c1, encoder_c2 et encoder. Elle prend en compte les paramètres ne (nombre d'équipes), nj (nombre de jours) et affichager_contrainte (indicateur pour afficher ou non les clauses générées).

En exécutant cette fonction avec les paramètres ne=3, nj=4 et affichager_contrainte=False, nous avons obtenu les résultats suivants :

```
Pour 3 équipes sur 4 jours :
La contrainte c1 génére : 60 clauses
Pour 3 équipes sur 4 jours :
La contrainte c2 génére : 42 clauses
Pour 3 équipes sur 4 jours :
Les contraites c1 et c2 génèrent : 102 clauses
```

Ces résultats correspondent bien aux calculs qu'on a effectués précédemment.

4 Utilisation du solveur

4.1 Vérifier la solution proposée pour 3 équipes sur 4 jours

Afin d'utiliser le solveur on a écrit la fonction générer_fichier_Cnf qui génère un fichier au format DIMACS à partir de la liste de clause qu'on génère avec la fonction encodée qu'on a expliquée précédemment.

Voici le code de la fonction :

```
def generer_fichier_cnf(clauses):
    clauses_list = transforme_liste(clauses)
    contenu = []

    # Ajout du Header
    contenu.append("c Fichier CNF g n r ")
    contenu.append("c")
    contenu.append("p cnf {} {}".format(max(map(abs, [var for clause in clauses_list fo

    # Ajout des clauses
    for clause in clauses_list:
        contenu.append(" ".join(map(str, clause)) + " 0")

    fichier_cnf = "\n".join(contenu)

    with open(nom_fichier_cnf, "w") as f:
        f.write(fichier_cnf)
```

Après avoir généré le fichier DIMACS à l'aide de la fonction generer_fichier_cnf pour le problème de planification de matchs avec 3 équipes sur 4 jours, nous avons exécuté le solveur SAT Glucose en utilisant le fichier CNF généré. Le résultat de l'exécution du solveur SAT a indiqué que le problème était UNSAT (insatisfiable).

Cela signifie que les contraintes spécifiées dans le fichier CNF ne peuvent pas être satisfaites simultanément. Le solveur SAT a analysé les clauses et a conclu qu'il n'existe aucune assignation de valeurs aux variables qui satisferait toutes les contraintes en même temps.

Le résultat UNSAT (insatisfiable) obtenu pour le problème de planification de matchs avec 3 équipes sur 4 jours est logique et attendu. En effet, pour que chaque équipe joue contre toutes les autres équipes, il faut au minimum 6 jours, car chaque équipe doit jouer deux matchs contre chaque autre équipe.

Dans le cas présent, avec seulement 4 jours disponibles, il est impossible de satisfaire toutes les contraintes requises pour que chaque équipe joue contre toutes les autres équipes. Par conséquent, le résultat UNSAT indique que le problème est insoluble dans les conditions actuelles.

Pour résoudre le problème, il serait nécessaire de disposer d'au moins 6 jours pour permettre à chaque équipe de jouer contre toutes les autres équipes. Si davantage de jours sont disponibles, il serait possible de générer des contraintes satisfaisables pour satisfaire toutes les exigences du problème.

4.2 Ajout d'une nouvelle contrainte C3

Nous avons décidé d'ajouter une nouvelle contrainte C3. Cette contrainte a été introduite pour résoudre le problème de la présence de matchs où une équipe se retrouve à jouer contre elle-même, ce qui est contradictoire et non souhaité.

La fonction encoder_c3 a été créée pour encoder cette nouvelle contrainte. Elle parcourt toutes les combinaisons possibles d'équipes et de jours de matchs, et ajoute une clause négative pour chaque situation où une équipe se retrouve à jouer contre elle-même. Ces clauses négatives sont ensuite ajoutées aux autres clauses existantes dans le fichier DIMACS.

Voici le code de la fonction encoder_c3 :

```
def encoder_c3(ne, nj):
    clauses = ""
    for xi in range(ne):
        for ji in range(nj):
            clauses += "-" + str(codage(ne, nj, ji, xi, xi)) + " 0\n"
    return clauses
```

En outre, nous avons également ajouté une nouvelle fonction appelée encoder_bis pour prendre en compte cette nouvelle contrainte encoder_c3. La fonction encoder_bis combine les anciennes contraintes avec la nouvelle contrainte encoder_c3

5 Décodage et Assemblage final

Afin d'afficher lisiblement le modèle de la solution de notre problème, nous avons créé plusieurs fonctions :

La fonction **decoder** : Elle prend en entrée un fichier contenant la sortie de l'appel à Glucose et un fichier externe (tel que **equipes.txt**) contenant les noms des équipes. Cette fonction permet de traduire le modèle généré par Glucose en une solution lisible pour le problème de planification des matchs.

Voici ce que fait la fonction **decoder** :

— Elle ouvre le fichier contenant les noms des équipes (**equipes.txt**) et les lit.

— Ensuite, elle vérifie si la sortie de Glucose indique que le problème est insatisfiable (**UNSATISFIABLE**). Si c'est le cas, elle renvoie un message correspondant.

— Elle récupère l'indice où se trouve le mot "SATISFIABLE" dans la sortie de Glucose.

— Elle récupère la solution du modèle donnée par Glucose, en ignorant les espaces et les caractères spéciaux.

— Elle divise la solution en une liste de matchs.

— Elle divise la liste des noms d'équipes en une liste d'équipes en utilisant le fichier **equipes.txt**

— Elle parcourt chaque match dans la liste des matchs.

— Si le code du match est positif, c'est-à-dire qu'il est joué, elle décode le jour, l'équipe à domicile et l'équipe à l'extérieur en utilisant la fonction decodage.

— Elle vérifie si le jour du match est déjà présent dans la liste des jours. Si ce n'est pas le cas, elle l'ajoute à la liste des jours.

— Elle détermine l'index du jour dans la liste des jours, qui représente le numéro du jour.

— Elle affiche chaque match avec son numéro, le numéro de la journée, le jour de la semaine, le numéro du jour dans la semaine, l'équipe à domicile et l'équipe à l'extérieur.

— Elle renvoie la solution sous forme de chaîne de caractères.

La fonction **programme** : Cette fonction englobe l'ensemble du processus de résolution du problème de planification des matchs. Elle appelle les autres fonctions dans l'ordre approprié pour générer les clauses, créer le fichier CNF, appeler Glucose, décoder le modèle et afficher la solution.

La fonction **main_1** : Cette fonction sert de point d'entrée du programme. Elle récupère le nom du fichier d'équipes en argument de ligne de commande, puis appelle la fonction **read_ne_nj** pour lire le nombre d'équipes et de jours à partir du fichier. Ensuite, elle appelle la fonction **programme** pour résoudre le problème de planification des matchs.

La fonction **read_ne_nj** : Cette fonction est utilisée pour lire le nombre d'équipes et de jours saisis par l'utilisateur. Elle vérifie également si l'utilisateur a entré des valeurs incorrectes.

Ces fonctions principales travaillent ensemble pour résoudre le problème de planification des matchs en utilisant Glucose comme solveur SAT. Elles orchestrent les différentes étapes du processus et permettent d'obtenir une solution lisible à partir du modèle généré par Glucose.

6 Tests et Affichage

Voici quelques examples qu'on a testé :

Pour ne = 3 et nj = 4

```
Entrez le nombre de jours du championnat (nj) :4
Entrez le nombre d'équipes qu'on prend on compte (ne) : 3
nombre d'equipes = 3
nombre de jours = 4
UNSAT (NON SATISFIABLE)
```

Pour ne = 6 et nj = 3

```
Entrez le nombre de jours du championnat (nj) :6
Entrez le nombre d'équipes qu'on prend on compte (ne) : 3
nombre d'equipes = 3
nombre de jours = 6
Match | Journée |   Jour   | Num jour | Equipe à domicile | Equipe à l'exterieur
 1    |  1      | Mercredi |  1       | Germany           | Japan
 2    |  2      | Dimanche |  2       | Japan             | France
 3    |  3      | Mercredi |  3       | Germany           | France
 4    |  4      | Dimanche |  4       | France            | Germany
 5    |  5      | Mercredi |  5       | Japan             | Germany
 6    |  6      | Dimanche |  6       | France            | Japan
```

Pour ne = 6 et nj = 20

```
Entrez le nombre de jours du championnat (nj) :20
Entrez le nombre d'équipes qu'on prend on compte (ne) : 6
nombre d'equipes = 6
nombre de jours = 20
```

Match	Journée	Jour	Num jour	Equipe à domicile	Equipe à l'exterieur
1	1	Mercredi	1	France	Japan
2	1	Mercredi	1	Egypt	Germany
3	1	Mercredi	1	China	Canada
4	2	Dimanche	2	Japan	Germany
5	2	Dimanche	2	Egypt	China
6	2	Dimanche	2	Canada	France
7	3	Mercredi	3	France	Egypt
8	3	Mercredi	3	Germany	Japan
9	4	Dimanche	4	Germany	Canada
10	4	Dimanche	4	Egypt	Japan
11	4	Dimanche	4	China	France
12	5	Mercredi	5	China	Egypt
13	6	Dimanche	6	Germany	Egypt
14	6	Dimanche	6	China	Japan
15	7	Mercredi	7	Egypt	France
16	7	Mercredi	7	Canada	Germany
17	8	Dimanche	8	France	Canada
18	8	Dimanche	8	Japan	Egypt
19	9	Mercredi	9	Japan	France
20	9	Mercredi	9	Egypt	Canada
21	10	Dimanche	10	France	Germany
22	10	Dimanche	10	Canada	Japan
23	11	Mercredi	11	Japan	Canada
24	12	Mercredi	13	Germany	China
25	13	Mercredi	15	Germany	France
26	13	Mercredi	15	Canada	China
27	14	Mercredi	17	China	Germany
28	15	Dimanche	18	France	China
29	16	Mercredi	19	Japan	China
30	16	Mercredi	19	Canada	Egypt

7 Optimisation du nombre de jours

Afin de trouver le nj minimal pour pouvoir planifier tous les matchs de championnat pour un ne donnée, on a écrit une fonction min_nj qui utilise des appels SAT de manière itérative pour trouver la valeur minimale de nj permettant de planifier tous les matchs de championnat pour un nombre donné d'équipes ne. On commence avec nj = 2 et on utilise une boucle while pour augmenter progressivement la valeur de nj. À chaque itération, on génère les clauses correspondantes avec la fonction encoder_bis, génère le fichier CNF et exécute Glucose. On incrémente ensuite nj pour essayer la prochaine valeur. Si le solveur trouve une solution satisfaisante, cela signifie que nj est suffisant pour planifier tous les matchs, donc on retourne la valeur actuelle de nj. Sinon, on continue à augmenter nj jusqu'à trouver une valeur qui satisfait les contraintes.

On a également optimisé cette fonction en utilisant une méthode d'itération par recherche dichotomique. On a écrit une autre fonction min_nj_dichotomique qui utilise une recherche dichotomique pour trouver la valeur minimale de nj. On initialise nj_min à ne (le nombre d'équipes) et nj_max à ne(ne-1)/2 (la valeur maximale possible de nj). Ensuite, on effectue une boucle while où on calcule nj_mid en prenant la moyenne de nj_min et nj_max. On génère les clauses correspondantes avec encoder_bis, génère le fichier CNF et exécute le solveur SAT. Si le solveur trouve une solution

satisfaisante, cela signifie que nj_mid est suffisant pour planifier tous les matchs, donc on met à jour nj_max à nj_mid. Sinon, cela signifie que nj_mid est insuffisant, donc on met à jour nj_min à nj_mid + 1. La boucle continue jusqu'à ce que nj_min soit égal à nj_max, et à ce stade, on retourne nj_min comme la valeur minimale de nj pour laquelle tous les matchs peuvent être planifiés.

On a également ajouté une fonction min_nj_plusieurs_equipes qui utilise la fonction min_nj_dichotomique pour trouver les valeurs minimales de nj pour une plage de nombres d'équipes allant de ne_min à ne_max. On utilise un gestionnaire de timeout pour limiter le temps d'exécution à 10 secondes. Si le calcul pour une valeur de ne prend plus de 10 secondes, on annule la commande et on passe à la valeur suivante. Les valeurs minimales de nj pour chaque nombre d'équipes sont affichées.

En exécutant la fonction min_nj_plusieurs_equipes(), on a pu obtenir des résultats en moins de 10 secondes pour nj = 3, nj = 4 et nj = 6. Cela signifie que pour les cas où le nombre d'équipes était de 3, 4 ou 6, on a pu trouver la valeur minimale de nj permettant de planifier tous les matchs de championnat dans un délai raisonnable. Cependant, pour les autres nombres d'équipes, la résolution du problème a pris plus de 10 secondes, dépassant ainsi le délai imparti.

```
Choisissez une option :
1. Modèle pour la résolution du problème du championnat
2. Nombre de jours minimum
2
Calcule pour ne = 3 et nj = 4
Calcule pour ne = 3 et nj = 5
Nombre minimal de jours pour 3 équipes : 6
Calcule pour ne = 4 et nj = 8
Calcule pour ne = 4 et nj = 6
Calcule pour ne = 4 et nj = 5
Nombre minimal de jours pour 4 équipes : 6
Calcule pour ne = 5 et nj = 12
Calcule pour ne = 5 et nj = 8
Le calcul pour 5 équipes a pris plus de 10 secondes. Annuler la commande.
Calcule pour ne = 6 et nj = 18
Calcule pour ne = 6 et nj = 12
Calcule pour ne = 6 et nj = 9
Calcule pour ne = 6 et nj = 11
Calcule pour ne = 6 et nj = 10
Nombre minimal de jours pour 6 équipes : 10
Calcule pour ne = 7 et nj = 24
Calcule pour ne = 7 et nj = 15
Calcule pour ne = 7 et nj = 11
Le calcul pour 7 équipes a pris plus de 10 secondes. Annuler la commande.
Calcule pour ne = 8 et nj = 32
Le calcul pour 8 équipes a pris plus de 10 secondes. Annuler la commande.
Calcule pour ne = 9 et nj = 40
Le calcul pour 9 équipes a pris plus de 10 secondes. Annuler la commande.
Calcule pour ne = 10 et nj = 50
Le calcul pour 10 équipes a pris plus de 10 secondes. Annuler la commande.
```

8 Equilibrer les déplacements et les week-ends

8.0.1 Encodage de contraintes de cardinalités ”au plus k”

Pour étendre l'encodage par paire pour les contraintes de cardinalité "au plus k", on doit prendre en compte quelles interprétations partielles contrediraient la contrainte de cardinalité afin de générer une clause contradictoire pour chacune d'entre elles.

Pour cela, on a écrit la fonction suivante :

```
def au_plus_k(variables, k):
    clauses = []
    n = len(variables)
    for i in range(k + 1, n + 1):
        for c in combinations(variables, i):
            clause = [str(-x) for x in c] + ["0"]
            clauses.append(" ".join(clause))

    return clauses
```

Cette fonction prend en entrée une liste de variables et un entier k. Elle génère et retourne une liste de clauses (au format DIMACS) correspondant à la contrainte "au plus k" de ces variables. Chaque clause représente une interprétation partielle contradictoire à la contrainte de cardinalité.

La fonction initialise une liste vide appelée clauses, qui va contenir les clauses contradictoires à la contrainte de cardinalité.

La variable n est initialisée avec la longueur de la liste variables, c'est-à-dire le nombre total de variables.

La boucle for itère de k + 1 à n + 1. Cela permet de générer des combinaisons de variables de toutes les tailles supérieures à k. Par exemple, si k vaut 2 et il y a 5 variables au total, la boucle générera des combinaisons de 3, 4 et 5 variables.

À chaque itération de la boucle externe, la boucle interne utilise la fonction combinations du module itertools pour générer toutes les combinaisons de variables de taille i (où i est l'itérateur de la boucle externe).

Pour chaque combinaison de variables, une clause contradictoire est créée. Cela est fait en convertissant chaque variable de la combinaison en une chaîne de caractères négative (précédée d'un signe moins) et en les concaténant avec l'opérateur logique "or" ("|") entre elles. Enfin, un "0" est ajouté à la fin pour indiquer la fin de la clause.

La clause contradictoire ainsi formée est ensuite ajoutée à la liste clauses.

Une fois que toutes les combinaisons de variables ont été traitées, la fonction retourne la liste

clauses, qui contient toutes les clauses contradictoires à la contrainte de cardinalité "au plus k" des variables.

8.1 Traduction en contraintes de cardinalité

8.1.1 contrainte sur les matchs le dimanche

Nous avons deux contraintes à prendre en compte : (a) pour éviter de trop perturber les études des joueurs et (b) pour permettre aux supporters d'assister à suffisamment de matchs.

(a) Pour chaque équipe i, nous souhaitons nous assurer qu'au moins pext% de ses matchs se déroulent à l'extérieur des dimanches.

(b) Pour chaque équipe i, nous souhaitons nous assurer qu'au moins pdom% de ses matchs se déroulent à domicile les dimanches.

Voici une traduction pour (a) et (b)

$$\bigwedge_{\substack{x \text{ : équipes} \\ y \text{ : équipe}}} \left(\sum_{j \text{ : jour impair}} \left(\text{match}(x, y, j) \geq \left\lfloor \frac{(ne-1) \cdot p_{\text{ext}}}{100} \right\rfloor \right) \right)$$

$$\wedge \left(\sum_{j \text{ : jour impair}} \left(\text{match}(y, x, j) \geq \left\lfloor \frac{(ne-1) \cdot p_{\text{dom}}}{100} \right\rfloor \right) \right)$$

8.1.2 contrainte sur les matchs consécutifs à domicile et à l'extérieur

Pour la contrainte sur les matchs consécutifs, nous souhaitons alterner les matchs à domicile et à l'extérieur. Nous avons donc les deux sous-contraintes suivantes :

(a) Aucune équipe ne joue (strictement) plus de deux matchs consécutifs à l'extérieur.

(b) Aucune équipe ne joue (strictement) plus de deux matchs consécutifs à domicile.

On propose la traduction suivante pour (a) et (b) :

$$\bigwedge_{\substack{x \text{ : équipes} \\ x \neq y \\ j \text{ : jours}}} \left(\sum_{y \text{ : équipe}} (\text{matchs}(x, y, j) + \text{matchs}(x, y, j+1) + \text{matchs}(x, y, j+2)) \leq 2 \right)$$

$$\wedge \left(\sum_{y \,:\, \text{équipe}} (\text{matchs}(y, x, j) + \text{matchs}(y, x, j+1) + \text{matchs}(y, x, j+2)) \leq 2 \right)$$

8.1.3 encondage des contraintes sur les matchs le dimanche

Dans cette section nous discuterons de l'encodage des contrainte 8.1.1 (a) et 8.1.1 (b) concernant les % des matchs exterieurs et domiciles joués les dimanches. Pour cela nous avons écrit la fonction suivante :

```
def encoder_c4(ne, nj, pext=50, pdom=40):

    kext = int((ne - 1) * pext / 100)
    kdom = int((ne - 1) * pdom / 100)

    clauses = []
    str_clauses = ""

    for x in range(ne):
        domiciles = []
        exterieurs = []
        for y in range(ne):
            if x != y:
                domicile = [codage(ne, nj, j, x, y) for j in range(1, nj, 2)]
                domiciles.append(domicile)

                exterieur = [codage(ne, nj, j, y, x) for j in range(1, nj, 2)]
                exterieurs.append(exterieur)

        flattened_domicile = [v for domicile in domiciles for v in domicile]
        flattened_exterieur = [v for exterieur in exterieurs for v in exterieur]
        clauses.extend(au_moins_k(flattened_domicile, kdom))
        clauses.extend(au_moins_k(flattened_exterieur, kext))

    for cl in clauses:
        str_clauses += cl + '\n'
    return str_clauses
```

La fonction encoder_c4 génère des contraintes logiques au format DIMACS pour représenter les contraintes des matchs joués le dimanche. Elle prend en compte le nombre d'équipes (ne), le nombre de jours (nj), ainsi que deux paramètres optionnels (pext et pdom) qui déterminent le pourcentage de matchs à l'extérieur et à domicile respectivement.

La fonction commence par calculer les valeurs de kext et kdom en utilisant les formules $(ne - 1) * pext/100 et (ne - 1) * pdom/100$ respectivement. Ces valeurs représentent le nombre minimal de matchs à l'extérieur et à domicile que devront jouer chaque équipe les dimanches[2].

Ensuite, une liste vide appelée clauses est initialisée pour stocker les contraintes logiques générées ainsi la version en string str_clauses aussi.

Une boucle for x in range(ne) : est utilisée pour itérer sur chaque équipe, représentée par la variable x.

Ensuite pour representer les adversaire des équipes x, La boucle interne for y in range(ne) : represente les équipes adverse (en excluant x == y).

À l'intérieur de la boucle for y, deux listes domicile et exterieur sont générées, Ces listes représentent les contraintes logiques pour les matchs joués à domicile et à l'extérieur respectivement entre les équipes x et y. Les matchs sont limités au dimanche (jours impairs).

On reprend le meme processus pour touts les équipes y adversaire de x et on concatene les listes domicile et exterieur à chaque fois dans des variables representant tout les matchs de chaque équipe x exterieurs et domiciles

On passe les variables résultante pour chaque x (de maniére séparée) dans la fonction "au_moins_k" avec k étant les variables kext et kdom respectivement pour les contrainte des matchs joués à l'exterieur et les contraintes des matchs joués à domicile,

Enfin, la chaîne de caractères str_clauses contenant toutes les contraintes générés pour chaque équipe x est retournée par la fonction.

8.1.4 encondage des contraintes sur les matchs consécutifs à domicile et à l'extérieur

Dans cette section nous discuterons de l'encodage des contraintes sur les matchs consécutifs à domicile et à l'extérieur 8.1.2 (a) et 8.1.2 (b). Pour cela nous avons écrit la fonction suivante :

```
def encoder_c5(ne, nj):

    clauses = []
    str_clauses = ""

    for x in range(ne):
        for j in range(nj-1):

            matchs_domicile = []
            matchs_exterieur = []

            for y in range(ne):
                matchs_domicile.append(codage(ne,nj,j,x,y))
                matchs_domicile.append(codage(ne,nj,j+1,x,y))
                matchs_domicile.append(codage(ne,nj,j+2,x,y))

                matchs_exterieur.append(codage(ne,nj,j,y,x))
                matchs_exterieur.append(codage(ne,nj,j+1,y,x))
                matchs_exterieur.append(codage(ne,nj,j+2,y,x))

            clauses.extend(au_plus_k(matchs_domicile, 1))
```

```
            clauses.extend(au_plus_k(matchs_exterieur, 1))

    for cl in clauses:
        str_clauses += cl + '\n'
    return str_clauses
```

La fonction encoder_c5 génère des contraintes logiques au format DIMACS pour représenter les contraintes des matchs joués le dimanche. Elle prend en compte le nombre d'équipes (ne), le nombre de jours (nj),

Tout d'abord La liste clauses sera utilisée pour stocker les contraintes logiques générées.

Une chaîne de caractères vide appelée str_clauses est également initialisée pour stocker les contraintes logiques sous forme de texte.

Une boucle for x in range(ne) : est utilisée pour itérer sur chaque équipe, représentée par la variable x.

À l'intérieur de la boucle for x, une autre boucle for j in range(nj-1) : est utilisée pour represenété chaque jour de la semaine par la variable j,

l'intérieur de la boucle for j, les listes matchs_domicile et matchs_exterieur sont réinitialisées comme des listes vides pour chaque jour.

Une autre boucle for y in range(ne) : est utilisée pour itérer sur les équipes adverses, représentées par la variable y.

À l'intérieur de la boucle for y, trois appels à la fonction codage() sont effectués pour générer les variables logique necessaire pour les contraintes des matchs consécutifs à domicile et à l'extérieur entre les équipes x et y. Les appels à la fonction codage() sont effectués pour les jours j, j+1 et j+2, ce qui représente trois matchs consécutifs.

Donc pour chaque jour on envoie les variables domiciles et exterieurs crées dans la fonction au_plus_k et k égale à 2 ce qui rerepresente sur toute les séquence de 3 matchs joués au plus 2 matchs peuvent etre joués à la fois (pour domicile d'un coté et exterieur de l'autre)

Enfin, la chaîne de caractères str_clauses est retournée par la fonction.

On peut noter que certains cas dans ces deux fonction ne sont pas pris en compte mais considéré comme aqcuis grace aux contraintes de début, On rappelle notamment que les équipes ne peuvent pas jouer plus d'un match par jour (si ce n'etait pas le cas il aurait fallu changer les fonctions encoder_c5 et encoder_c4 en conséquence).

9 Tests et Affichage avec les contraintes C4 et C5

Voici quelques examples qu'on a testé :

Pour ne = 3 et nj = 6

```
Choisissez une option :
1. Modèle pour la résolution du problème du championnat
2. Nombre de jours minimum
1
Voulez-vous prendre en compte C4 et C5 ? (1. oui/2. non)
1
Entrez le nombre de jours du championnat (nj) :6
Entrez le nombre d'équipes qu'on prend on compte (ne) : 3
nombre d'equipes = 3
nombre de jours = 6
Match | Journée |   Jour    | Num jour | Equipe à domicile | Equipe à l'exterieur
 1    |  1      | Mercredi  |  1       | France            | Japan
 2    |  2      | Dimanche  |  2       | Japan             | France
 3    |  3      | Mercredi  |  3       | Germany           | Japan
 4    |  4      | Dimanche  |  4       | Japan             | Germany
 5    |  5      | Mercredi  |  5       | Germany           | France
 6    |  6      | Dimanche  |  6       | France            | Germany
```

Pour ne = 5 et nj = 8

```
Choisissez une option :
1. Modèle pour la résolution du problème du championnat
2. Nombre de jours minimum
1
Voulez-vous prendre en compte C4 et C5 ? (1. oui/2. non)
1
Entrez le nombre de jours du championnat (nj) :8
Entrez le nombre d'équipes qu'on prend on compte (ne) : 5
nombre d'equipes = 5
nombre de jours = 8
UNSAT (NON SATISFIABLE)
```

Pour ne =24 et nj = 8

```
Voulez-vous prendre en compte C4 et C5 ? (1. oui/2. non)
1
Entrez le nombre de jours du championnat (nj) :24
Entrez le nombre d'équipes qu'on prend on compte (ne) : 8
nombre d'equipes = 8
nombre de jours = 24
```

Match	Journée	Jour	Num jour	Equipe à domicile	Equipe à l'exterieur
1	1	Mercredi	1	France	Egypt
2	1	Mercredi	1	Germany	China
3	1	Mercredi	1	Saudi Arabia	Japan
4	2	Dimanche	2	France	Japan
5	2	Dimanche	2	Germany	Canada
6	2	Dimanche	2	Mali	Egypt
7	2	Dimanche	2	Saudi Arabia	China
8	3	Mercredi	3	Japan	France
9	3	Mercredi	3	China	Canada
10	3	Mercredi	3	Saudi Arabia	Mali
11	4	Dimanche	4	France	Canada
12	4	Dimanche	4	China	Germany
13	4	Dimanche	4	Saudi Arabia	Egypt
14	5	Mercredi	5	France	Germany
15	5	Mercredi	5	Egypt	Saudi Arabia
16	6	Dimanche	6	Japan	Saudi Arabia
17	6	Dimanche	6	Canada	France
18	7	Mercredi	7	France	China
19	7	Mercredi	7	Germany	Japan
20	8	Mercredi	9	Japan	China
21	8	Mercredi	9	Egypt	Mali
22	9	Dimanche	10	Japan	Canada
23	9	Dimanche	10	China	Saudi Arabia
24	9	Dimanche	10	Mali	Germany
25	10	Mercredi	11	Germany	Egypt
26	10	Mercredi	11	Mali	France
27	11	Dimanche	12	Canada	China
28	12	Mercredi	13	Japan	Mali
29	12	Mercredi	13	Egypt	Canada
30	13	Dimanche	14	France	Mali
31	14	Mercredi	17	Egypt	Germany
32	14	Mercredi	17	Canada	Japan
33	14	Mercredi	17	Mali	China
34	15	Dimanche	18	Egypt	France
35	15	Dimanche	18	China	Japan
36	15	Dimanche	18	Mali	Canada
37	15	Dimanche	18	Saudi Arabia	Germany
38	16	Mercredi	19	Germany	France
39	16	Mercredi	19	Egypt	China
40	16	Mercredi	19	Mali	Japan
41	16	Mercredi	19	Saudi Arabia	Canada
42	17	Dimanche	20	Japan	Germany
43	17	Dimanche	20	Canada	Egypt
44	17	Dimanche	20	China	France
45	17	Dimanche	20	Mali	Saudi Arabia
46	18	Mercredi	21	Canada	Mali
47	18	Mercredi	21	China	Egypt
48	18	Mercredi	21	Saudi Arabia	France
49	19	Dimanche	22	Japan	Egypt
50	19	Dimanche	22	Germany	Saudi Arabia
51	20	Mercredi	23	Germany	Mali
52	20	Mercredi	23	Egypt	Japan
53	20	Mercredi	23	Canada	Saudi Arabia
54	21	Dimanche	24	France	Saudi Arabia
55	21	Dimanche	24	Canada	Germany
56	21	Dimanche	24	China	Mali

10 Conclusion

En conclusion, ce rapport a présenté le projet IAMSI, qui consiste en la création d'un générateur de championnat en utilisant le langage de programmation Python et la programmation SAT. Nous avons réussi à modéliser le problème, traduire la modélisation en fichiers DIMACS et utiliser le solveur Glucose pour tester la satisfiabilité des clauses générées. Les différentes étapes de développement ont été détaillées, en mettant l'accent sur la modélisation, la génération des fichiers DIMACS et l'utilisation du solveur Glucose. Les résultats obtenus pour diverses instances de test ont été présentés et analysés. En utilisant les fonctions de codage et de décodage, nous avons pu représenter et manipuler les variables propositionnelles de manière pratique et efficace. Ce projet a permis d'aborder de manière approfondie la résolution d'un problème complexe en utilisant des techniques de modélisation avancées.