Métaheuristiques

M2 Informatique - Parcours Intelligence Artificielle
Année 2023 - 2024

Timothée Afonso

Afonso Timothée

1. Description du problème

Le problème est celui de la planification de rencontres sportives.

Nous considérons les spécifications suivantes :

- un tournoi regroupe un ensemble T d'équipes ($|T|$ pair)
- chaque équipe rencontre toutes les autres exactement une fois. La notion de matches aller/retour n'est pas traité dans ce problème
- le tournoi se déroule sur $|T| - 1$ semaines. Chaque équipe joue une et une seule fois chaque semaine
- La programmation d'une rencontre se déroule sur un terrain, à une certaine date. Nous emploierons le terme générique de période, au nombre de $|T|/2$. A chaque période de chaque semaine correspond un match. Aucune équipe ne peut jouer plus de deux fois sur une période.

Le problème est de déterminer une programmation qui respecte l'ensemble de ces contraintes.
Le Tableau 1 montre un exemple de planification vérifiant l'ensemble des contraintes pour $|T| = 8$ équipes numérotées de 0 à 7 (il y a donc 7 semaines et 4 périodes).

	Période 0	Période 1	Période 2	Période 3
Semaine 0	0 vs 1	2 vs 3	4 vs 5	6 vs 7
Semaine 1	0 vs 2	1 vs 7	3 vs 5	4 vs 6
Semaine 2	4 vs 7	0 vs 3	1 vs 6	2 vs 5
Semaine 3	3 vs 6	5 vs 7	0 vs 4	1 vs 2
Semaine 4	3 vs 7	1 vs 4	2 vs 6	0 vs 5
Semaine 5	1 vs 5	0 vs 6	2 vs 7	3 vs 4
Semaine 6	2 vs 4	5 vs 6	0 vs 7	1 vs 3

Figure 1 - Exemple de tableau pour 8 équipes

Comme dans le tableau précédent, une configuration peut être représentée sous la forme d'un tableau à deux dimensions avec les semaines en lignes et les périodes en colonnes. Chaque colonne vérifie la contrainte de cardinalité pour laquelle aucune équipe n'apparaît plus de deux fois. Sur chaque ligne, chaque équipe apparaît exactement une fois. Ce qui est équivalent à dire que toutes les équipes d'une ligne sont différentes. Enfin, il existe une contrainte globale sur tout le tableau : chaque match n'y apparaît qu'une seule fois (tous les matches sont différents). Dans la suite du rapport les semaines, les périodes et les numéros d'équipe commencent à 0.

On retient 3 contraintes principales:

- Contrainte n°1: chaque équipe joue une et une seule fois chaque semaine
- Contrainte n°2: aucune équipe ne peut jouer plus de deux fois sur une période
- Contrainte n°3: chaque équipe rencontre toutes les autres exactement une fois (pas de match aller/retour)

2. Modélisation

Le problème de planification de rencontre sportive peut être modélisé de la manière suivante.

Variables:

T, nombre d'équipe
$S = T - 1$, nombre de semaine
$P = T/2$, nombre de période
$C = [(i,j), ..], \forall_{i,j} \in \{0 .. T-1\}, i < j$, ensemble des rencontres à planifier avec i et j des équipes numérotées de 0 à T-1.
$X_{i,j,s,p} \in \{0,1\}, \forall_{(i,j)} \in C, \forall_s \in \{0 .. S-1\}, \forall_p \in \{0 .. P-1\}$, $X_{i,j,s,p}$ est égal à 1 si la rencontre entre l'équipe i et j à lieu la semaine s à la période p, 0 sinon

Contraintes:

$$c1 = \forall_i \in \{0 .. T-1\}, \forall_s \in \{0 .. S-1\}, \left(\sum_{j=0}^{T-1} \sum_{p=0}^{P-1} , i < j , X_{i,j,s,p} + \sum_{j=0}^{T-1} \sum_{p=0}^{P-1} , i > j , X_{j,i,s,p} \right) = 1$$

$$c2 = \forall_i \in \{0 .. T-1\}, \forall_p \in \{0 .. P-1\}, \left(\sum_{j=0}^{T-1} \sum_{s=0}^{S-1} , i < j , X_{i,j,s,p} + \sum_{j=0}^{T-1} \sum_{s=0}^{S-1} , i > j , X_{j,i,s,p} \right) \leq 2$$

$$c3 = \forall_{(i,j)} \in C, \left(\sum_{s=0}^{S-1} \sum_{p=0}^{P-1} X_{i,j,s,p} \right) = 1$$

Cependant le CSP est un problème d'optimisation combinatoire complexe. En effet, il est NP-complet. J'ai implémenter un algorithme de métaheuristique afin de trouver des solutions de bonne qualité en un temps raisonnable. J'ai choisi l'algorithme de recherche locale avec la méthode de descente. Afin de ne pas s'arrêter au premier optimum local rencontré j'ai intégrer des relances multiple semi-aléatoire qui acceptent des voisins moins performants

L'algorithme et son implémentation seront détaillés dans la suite du rapport. L'initialisation du problème joue aussi un rôle important sur la méthode et la vitesse de résolution du problème et sera détaillée dans la quatrième partie.

3. Structure du code

Le langage de programmation utilisé est le python. Le problème est modélisé par une classe "*Tournoi*" comportant différents attributs:

- un tableau à deux dimensions appelé "*tableau*". On retrouve les semaines en lignes et les périodes en colonne. Chaque élément du tableau est un tuple (i, j) correspondant à la rencontre entre l'équipe i et l'équipe j:
- des variables entières $nbEquipe$, $nbSemaine$, $nbPeriode$ correspondant respectivement aux nombre d'équipes, de semaine et de période.
- une liste de rencontres correspondant aux matchs qui ne respecte pas la contrainte n°2 appelé "*match_en_erreur*".
- une liste de rencontres correspondant aux matchs qui respecte la contrainte n°2 appelé "*match_bon*".

Cette classe possède les méthodes suivantes:

- une méthode de recherche locale appliquant la descente.
- une méthode utilisant un solveur python pour trouver une solution initiale
- Une méthode retournant la liste des voisin par rapport à un match en erreur
- une méthode permettant de mettre à jour les listes "*match_en_erreur*" et "*match_bon*"
- des méthodes permettant de trier des listes de solution voisine ou de rencontre

Une fonction permettant de calculer la pénalité d'un match, d'une période ou d'un tableau existe également.

```
class Tournoi:
    def __init__(self, nbEquipe):

        self.nbEquipe = nbEquipe
        self.nbSemaine = nbEquipe - 1
        self.nbPeriode = nbEquipe // 2
        self.tableau = [[(None,None) for i in range(self.nbPeriode)] for j in range(self.nbSemaine)]
        self.match_en_erreur = []
        self.match_bon = []

    def __str__(self):···

    def solve(self,sDebut,sFin):···

    def generer_solutions_voisines(self,all_match,match_en_erreur_i,match_switch):···

    def recherche_locale_descente(self):···

    def set_type_match(self):···

    def sort_by_per(self,un_match):···

    def get_nb_err_periode(self):···

    def sort_by_pena(self,un_match):···

    def sort_voisin(self,un_voisin):···

def verifTableau(tournoi,justif = False):···

def verifMatch(tournoi,sem=None,per=None,a_calculer=None):···
```

Figure 2 - Structure du code

4. Initialisation du problème

La solution initiale choisi doit respecter uniquement deux contraintes:

- Contrainte n°1: chaque équipe joue une et une seule fois chaque semaine
- Contrainte n°3: chaque équipe rencontre toutes les autres exactement une fois (pas de match aller/retour)

La liste de toutes les rencontres (match) est générée lors du lancement du programme. Ce sont les rencontres de cette liste que l'on doit placer dans chaque emplacement du tableau. Dès la création de la liste on supprime la contrainte n°2 car toutes les rencontres sont différentes et pour toutes les rencontres, l'équipe n°2 est supérieure à l'équipe n°1 (donc pas de match aller retour).

$$combinaisons \ = \ [(i,j), \ \forall \, 0 \leq i < \ j \leq T \ - \ 1 \,], \quad avec \, T \, le \, nombre \, d'équipe$$

Les rencontres de la première semaine du tableau sont insérés automatiquement après la création de la liste des rencontres. En effet, il existe une symétrie. Si il existe une solution, alors peut importe l'ordre d'apparition des équipes dans la première semaine, il y aura toujours une solution.

La première semaine est fixée comme suit pour un tournoi à 8 équipes. Ce schéma est répété selon le nombre d'équipes.

	Période 0	Période 1	Période 2	Période 3
Semaine 0	0 vs 1	2 vs 3	4 vs 5	6 vs 7

Figure 3 - Planification de la première semaine pour un tournoi à 8 équipes

A la modélisation de la partie 2, on peut ajouter la contrainte suivante qui permet de fixer la première semaine.

$$c4_{symétrie} \ = \forall_{(i,j)} \in C, \ i == 2p, \ j == i + 1 \ , (\ X_{i,j,0,p}\ ) = 1$$

J'ai ensuite utilisé la bibliothèque PuLP de python qui utilise un solveur permettant de résoudre des problèmes linéaires. Le solveur permet de placer les matches restants pour les semaines restantes tout en respectant la contrainte n°1 et la contrainte n°3.

Cette solution initiale permet de réduire grandement l'espace de recherche. En effet, pour chaque algorithme de recherche local, les permutations se feront sur les rencontres des semaines 1 à $S \ - \ 1 \ (S \, le \, nombre \, de \, semaine)$, la première semaine étant fixe. De plus, il ne reste plus que la contrainte n°2 à obtenir. Ainsi, chacune des permutations se fera uniquement entre deux rencontres de la même semaine (afin de ne pas briser la contrainte n°1).

5. Recherche local - Descente

A partir de la solution initiale, on cherche des voisins plus performants en réalisant des permutations entre un match en erreur et un second match de la même semaine (pas forcément en erreur). Le but étant de trouver une solution optimale. On s'arrête à la première amélioration trouvée. Pour résumer, on sélectionne le premier match en erreur, on génère sa liste de voisin (un voisin correspond à une permutation entre un match en erreur et un second match de la même semaine) puis on sélectionne le premier voisin améliorant, enfin on met à jour la liste des match en erreur et on recommence jusqu'à atteindre une solution optimal ou un nombre d'itération données.

L'ordre de sélection des matchs en erreur ainsi que l'ordre de parcours des voisins est important car à chaque itération on s'arrête au premier voisin améliorant trouver. Plusieurs méthodes de trie ont été testées. La plus performante est la suivante:
On trie d'abord les périodes selon leur nombre de match en erreur dans l'ordre décroissant dans une liste "*liste_periode"*. On trie ensuite la liste des matchs en erreur selon la période à laquelle appartient chaque match et dans l'ordre de "*liste_periode"*. Pour finir on parcourt la liste des match en erreur dans le nouvel ordre.

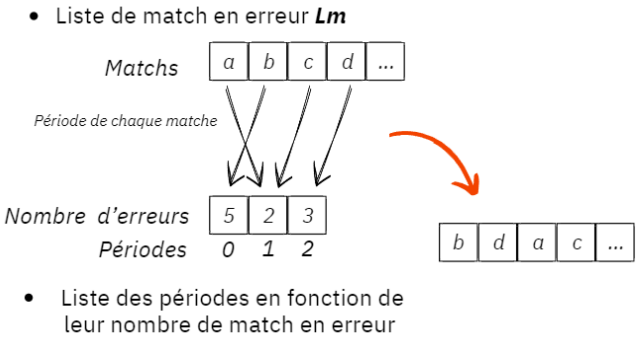

Figure 4 - Méthode de trie appliqué à la liste de match en erreur

La liste des voisins retournée par chaque match en erreur est triée dans l'ordre croissant des gains de pénalités obtenues. Les pénalités sont obtenue de la façon suivante:

- Pour une permutation de deux matchs "*m1"* et "*m2"*, on calcul avant permutation l'ancienne pénalité totale "*ancienne_penalite"* puis après permutation la nouvelle pénalité totale "*nouvelle_penalite"* .
- Pour chaque match *"m"*, une pénalité correspond au nombre d'équipes qui ne respectent pas la contrainte sur la période de *"m"*.
- La pénalité totale correspond donc à la somme des pénalités de *"m1"* et "*m2"*.
- Le gain de pénalité obtenu correspond à la différence entre "*ancienne_penalite"* et "*nouvelle_penalite"*.

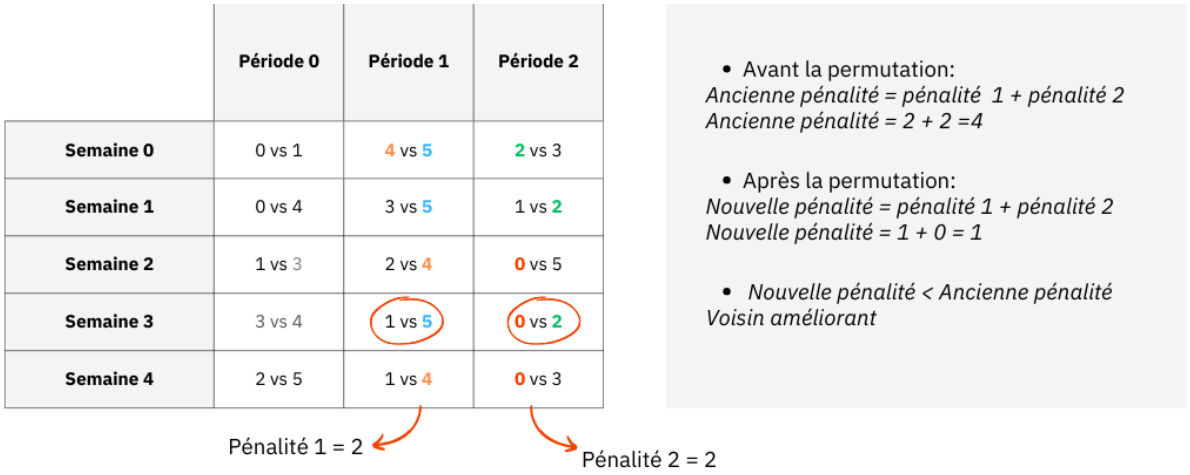

Figure 5 - Méthode de calcul des pénalités

On trie ensuite les voisins en fonction de leur pénalité.

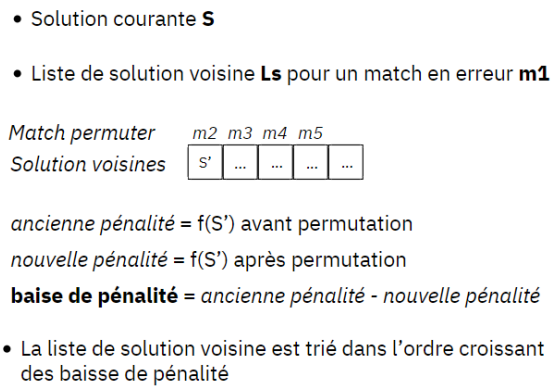

Figure 5 - Méthode de trie appliqué à la liste des solution voisines

Afin de ne pas permettre des rencontres qui ont déjà été permutées entre elles et ne pas créer de boucle on met en place une liste "*match_switch*". Cette liste enregistre,une fois un voisin améliorant trouvé, les deux matchs permutés. Lorsque l'on générera une nouvelle liste de voisin, on vérifiera pour chaque voisin que les match permuté ne sont pas dans la liste "*match_switch*" . Cette liste est vidé lorsqu'un optimum local est rencontré.

Après avoir parcouru tous les match en erreur, il est possible qu'il n'y ait pas de voisin améliorant et que l'on ait atteint un optimum local. Il faut donc mettre en place une relance semi-aléatoire. Pour cela, on sélectionne aléatoirement un voisin qui n'est pas améliorant. Le match en erreur est choisi aléatoirement parmi les matchs à pénalité la plus élevé et dont la période est celle avec le plus de match en erreur. Pour chaque voisin du match en erreur choisi on en sélectionne un aléatoirement parmi ceux dont la pénalité est inchangée (si il n'en existe pas on en choisit un avec une pénalité légèrement supérieur).

L'algorithme implémenté pour la descente est le suivant.

Algorithme 1 : Algorithme de recherche local avec la méthode de descente et relance multiple semi-aléatoire

input : *S*, solution initial, tableau à deux dimension avec les semaines en lignes et les périodes en colonne. Chaque élément du tableau est un tuple (i,j) correspondant à la rencontre entre l'équipe i et l'équipe j
*S**, meilleur solution trouvé
opti_trouver, boolean définissant si une solution voisine est de meilleur qualité que la solution courante
matchs_switch, liste de rencontre déjà permuté
optimal, nombre de rencontres ne respectant pas la contrainte sur une période
matchs_en_erreur, liste des rencontres ne respectant pas la contrainte sur une période
nb_semaine, nombre de semaine
nb_periode, nombre de période
iter, nombre d'itération

output : *S**, meilleur solution trouvé

générer_match_en_erreur() ← génère la liste des rencontres ne respectant pas la contrainte sur une période. La liste est trié en fonction des rencontres dont la période possède le plus de rencontre en erreur
générer_solution_voisine(match_en_erreur , match_switch) ← génère la liste des voisins après permutation de *match_en_erreur* avec les autres matchs de la semaine. Ne crée pas un voisin dont les match permuté sont dans *match_switch*
générer_solution_initiale() ← génère la solution initiale
S ← *générer_solution_initiale()*
*S** ← *S*
match_en_erreur ← *générer_match_en_erreur()*
optimal ← taille de *matchs_en_erreur*
matchs_switch ← Ø
iter ← 0
tant que *optimal != 0* et *iter != 30000* //*On choisit un maximum de 30000 itérations*

- *opti_trouver* ← *False*
- **pour chaque** *match* **dans** *matchs_en_erreur*
  - **si** *opti_trouver == False* // *s'arrêter à la première amélioration trouvé*
    - *voisins* ← *générer_solution_voisine(match_en_erreur , match_switch)*
    - **pour chaque** *S'* **dans** *voisins*
      - *match_1, match_2* ← *match permuté de S'*
      - **si** *opti_trouver == False* // *s'arrêter à la première amélioration trouvé*
        - **si** *f(S') < f(S)*
          - *S* ← *S'*
          - *insérer (match_1,match_2) dans match_switch*
          - *iter* ← *iter + 1*
- **si** *opti_trouver == False*
  - *matchs_en_erreur'* ← *génère la liste des matchs avec la pénalité la plus forte dont la période est celle ou il y a le plus de match en erreur*
  - *match_1* ← *random(matchs_en_erreur')* // *sélectionne un match aléatoirement dans matchs_en_erreur'*
  - *list_match_2* ← *génère une liste de match m qui sont dans la même semaine que match_1 et dont (m,match_1) ∉ match_switch*
  - *meilleur_penalite* ← 0
  - *list_match_2_meilleur_penalite* ← Ø
  - **pour chaque** *match_2* **dans** *list_match_2*
    - *S'* ← *S*
    - *S'* ← *permuter(match_1,match_2)*
    - **si** *f(S') > meilleur_penalite ou list_match_2_meilleur_penalite == Ø*
      - *meilleur_penalite = f(S')*
      - *list_match_2_meilleur_penalite* ← Ø
      - *insérer match_2 dans list_match_2_meilleur_penalite*
    - **sinon si** *f(S') == meilleur_penalite*
      - *insérer match_2 dans list_match_2_meilleur_penalite*
  - *match_2* ← *random(list_match_2_meilleur_penalite)*
  - *S* ← *permuter(match_1,match_2)*
  - *match_switch* ← Ø
  - *insérer (match_1,match_2) dans match_switch*
  - *iter* ← *iter + 1*
- *match_en_erreur* ← *générer_match_en_erreur()*
- *nouvelle_optimal* ← taille de *matchs_en_erreur*
- **Si** *nouvelle_optimal < optimal*
  - *optimal* ← *nouvelle_optimal*
  - *S** ← *S*

Figure 6 - Algorithme utilisé pour implémenter la recherche local avec la méthode de la descente et relance multiple semi-aléatoire

6. Utilisation

Le code fourni permet d'appliquer la recherche locale avec la méthode de descente au problème de la planification de rencontres sportives. Le fichier s'appelle "*tournament.py*". Pour exécuter le fichier dans un environnement linux voici les commande a suivre:

- Il est nécessaire d' installer python et PuLP. Pour PuLP exécuter la commande suivante.
  pip install pulp
- Ce placer dans le repertoire du fichier "*tournament.py*"
- Lancer la commande: *python3 tournament.py <nombre d'éuquipe>*
  Exemple pour un tournoi de 6 équipes:
  python3 tournament.py 6
- Si l'on souhaite ajouter le temps de résolution exécuter la commande suivante:
  time python3 tournament.py 6

Attention le nombre d'équipes doit être pair.

L'affichage dans la console pour un tournoi de 6 équipes est le suivant.

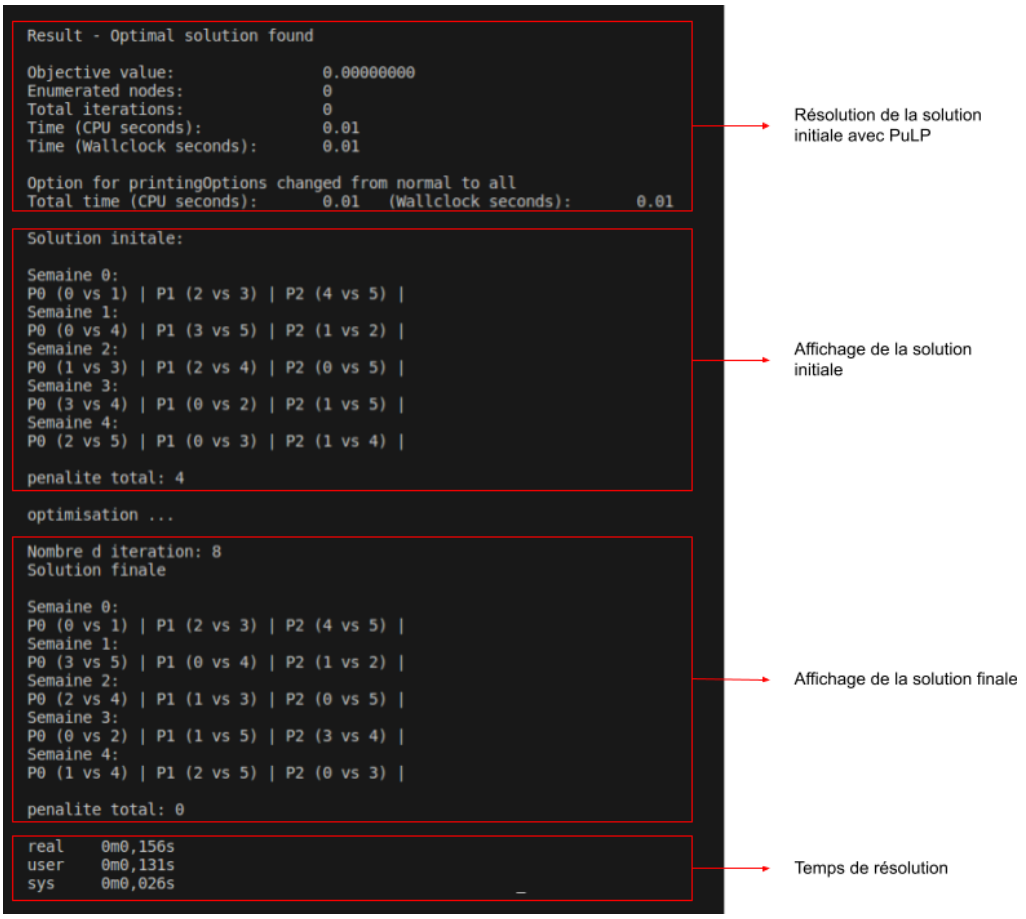

Figure 7 - Affichage de la console après l'exécution du programme avec 6 équipes

7. Résultats

La recherche local avec la méthode permet de résoudre des tournois jusqu'à 14 équipes dans un temps raisonnable. Il est possible de résoudre 16 équipes mais le temps de résolution est important.

Nombre d'équipe	Solution optimale trouvé	Temps de résolution (en seconde)	Nombre d'itération	Nombre de rencontre mal placé
6	Oui	0,15	10	0
8	Oui	0,40	50	0
10	Oui	5	200	0
12	Oui	50	1000	0
14	Oui	520	10 000	0
16	Oui	900	25 000	0
18	Non	-	Limite	15

Figure 8 - Tableau des résultats