Métaheuristique

—

Sport Tournament Scheduling

Cyril Grelier

Encadrant :
Jin-Kao Hao

12 février 2020

Table des matières

1 Introduction **1**

2 Sport Tournament Scheduling **2**

3 État de l'art **2**

4 Représentation **3**

5 Gestion des contraintes **3**

6 Méthodes de résolution **4**

7 Tests et résultats **7**

8 Conclusion **7**

1 Introduction

Dans le cadre du cours de métaheuristiques, nous découvrons différentes méthodes de résolutions de problèmes, telles que les recherches locales, les algorithmes génétiques, méthodes gloutonnes, Branch & Bound, etc. Nous sommes ensuite amenés à implémenter ces méthodes afin de résoudre des problèmes d'optimisation. Le problème analysé ici sera celui du Sport Tournament Scheduling (STS), le problème 26 de CSPLib[3].

Dans un premier temps, nous vous présenterons le problème du STS et quelques méthodes utilisées pour résoudre ce problème. Ensuite, nous vous présenterons les méthodes implémentées suivies des tests et résultats. Enfin, nous conclurons ce rapport.

Le projet est développé en Python. Le code est disponible sur github à l'adresse `https://github.com/Cyril-Grl/Sport_Tournament_Sheduling_metaheuristique`.

2 Sport Tournament Scheduling

2.1 Description du problème

Le Sport Tournament Scheduling est un problème de planification de rencontres sportives lors d'un tournoi.

Nous considérons les spécifications suivantes :

— Un tournoi regroupe un ensemble n équipes (n pair).

— Chaque équipe rencontre toutes les autres exactement une fois. La notion de matches aller/retour n'est pas traitée dans ce problème.

— Le tournoi se déroule sur $n-1$ semaines. Chaque équipe joue une seule fois chaque semaine.

— Chaque semaine est divisée en $n/2$ période. Chaque équipe joue deux fois au maximum sur une même période.

Le problème est de déterminer une programmation qui respecte l'ensemble de ces contraintes.

	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 planification vérifiant l'ensemble des contraintes pour $n = 8$ équipes numérotées de 0 à 7 (il y a donc 7 semaines et 4 périodes).

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 ou inversement.

3 État de l'art

Depuis plus de trente ans, beaucoup de méthodes on été mises en place pour résoudre le problème du STS. Il existe cependant beaucoup de déclinaisons de celui-ci sous différents noms, sport league tournament, round-robin tournament,... dont les contraintes sont généralement légèrement différentes.

Dans l'article "Sports League Scheduling : Enumerative Search for Prob026 from CSPLib" [2], Jean-Philippe Hamiez et Jin-Kao Hao en résument certaines :

— programmation linéaire en nombres entiers (jusqu'à 12 équipes),

— différentes versions de recherche locale (jusqu'à 20 équipes),

— programmation par contraintes (jusqu'à 30 équipes),

— combinaison de recherche locale avec tabou et programmation par contrainte (jusqu'à 40 équipes),

— algorithme de réparation (jusqu'à 40 équipes).

Dans l'article en question, jusqu'à 70 équipes ont été testées en utilisant un algorithme énumératif.

Il est aussi précisé qu'il existe des solutions pour tout $n \neq 4$ et des solutions directes ont étés proposées pour $(n-1)\%3 \neq 0$ ou $n/2$ impair.

Il existe de nombreuses représentation du problème, beaucoup sous forme de matrice, mais il est aussi possible de le représenter sous forme de graphe où chaque point représente les équipes et chaque arrête les rencontres entre les différentes équipes, une partie du problème devient alors un problème de coloration de graphe[1].

4 Représentation

Nous avons représenté ce problème sous forme de tableau en deux versions. Dans les deux cas, chaque case correspond à un couple de deux valeurs correspondant aux deux équipes se rencontrant (tuples Python). Dans la première version, nous utilisons une matrice de taille $s * p$ où s correspond au nombre de semaines et p le nombre de périodes et leurs transposées. Dans la seconde version la matrice est aplatie afin de faciliter les opérations de crossover et de mutation pour l'utilisation d'un algorithme génétique. Voici un exemple pour 6 équipes, chaque ligne représente une semaine et chaque colonne une période :

$$[[(0,1),(2,3),(4,5)],$$
$$[(3,5),(1,4),(0,2)],$$
$$[(3,4),(0,5),(1,2)],$$
$$[(2,5),(0,4),(1,3)],$$
$$[(2,4),(1,5),(0,3)]]$$

5 Gestion des contraintes

Les méthodes implémentées auront pour but de satisfaire toutes les contraintes du STS :
— tournoi simple : chaque équipe joue contre toutes les autres exactement une fois,
— unicité, contrainte semaine : chaque équipe joue exactement 1 fois / semaine,
— double, contrainte période : aucune équipe ne peut jouer plus de 2 fois / période.

Dans tous les cas, la contrainte du tournoi simple est satisfaite en générant toutes les rencontres et en les plaçant aléatoirement sur le planning. Les méthodes devront donc satisfaire les contraintes sur les semaines et celles des périodes.

Avec le planning représenté sous forme de matrice, en analysant chaque ligne et les matchs présents nous pouvons obtenir le nombre d'incohérences vis-à-vis de la contrainte sur les semaines. En générant une matrice contenant les positions des incohérences nous pouvons donc repérer les zones du planning les plus impératives à modifier. Après une transposition de la matrice du planning et de la matrice d'incohérences, nous pouvons y ajouter facilement les incohérences concernant la contrainte sur les périodes.

Voici un exemple. À gauche, le planning présentant des incohérences, par exemple lors de la première ligne (semaine) l'équipe 1 et 3 jouent deux fois et sur la première colonne (période) l'équipe 3 joue trois fois. Au milieu, vous pouvez trouver la matrice d'incohérences des semaines de la matrice de gauche, ainsi la première ligne, première colonne est à deux car l'équipe 1 et 3 jouent trop de fois dans la semaine. À droite, nous avons ajouté les incohérences concernant les périodes à la matrice précédente, ainsi la première rencontre possède une incohérence de plus à cause du nombre d'apparition de l'équipe 3 dans la première période.

$$[[(1,3),(0,3),(1,2)], \quad [[2,1,1], \quad [[3,1,1],$$
$$[(3,4),(3,5),(0,5)], \quad [1,2,1], \quad [2,3,2],$$
$$[(2,3),(4,5),(0,2)], \quad [1,0,1], \quad [2,1,2],$$
$$[(2,4),(1,4),(0,4)], \quad [1,1,1], \quad [1,1,2],$$
$$[(0,1),(2,5),(1,5)]] \quad [1,1,2]] \quad [1,2,2]]$$

Nous pouvons aussi compter le nombre d'incohérences dans un planning afin de comparer deux plannings, équivalent d'une fitness pour les algorithmes génétiques.

6 Méthodes de résolution

Trois méthodes ont été mises en place pour la résolution du problème. Tout d'abord une recherche locale classique, ensuite une recherche locale avec tabou puis l'utilisation d'un algorithme génétique.

6.1 Recherche locale

La recherche locale consiste à partir d'une solution aléatoire (ou déjà un minimum performante) à explorer le voisinage de la solution courante afin de trouver une solution améliorante. Dans notre cas, le voisinage correspond à la permutation dans le planning de deux rencontres. La recherche s'arrête lorsqu'elle ne trouve plus de solution améliorante.

Voici la structure simplifiée du fonctionnement implémenté (avec quelques modifications vis-à-vis du fonctionnement de base) :

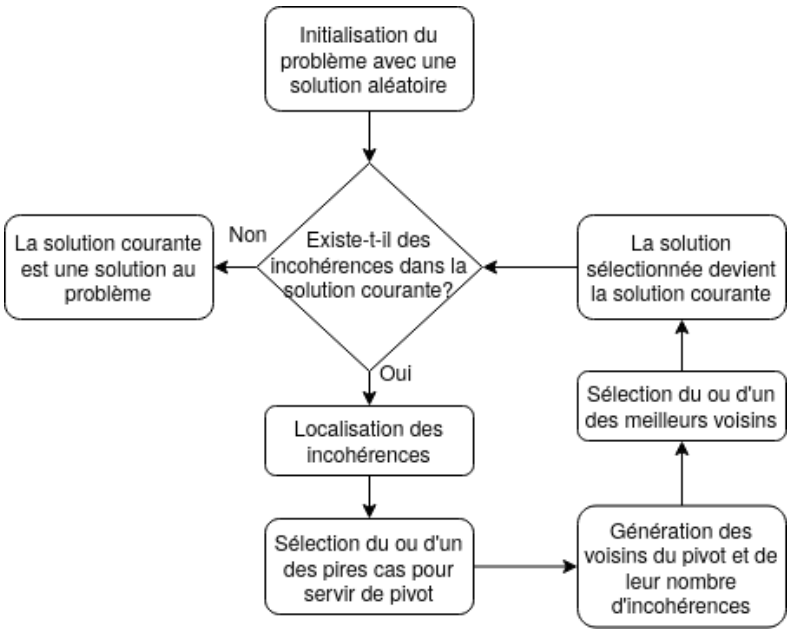

FIGURE 2 – En partant d'une solution aléatoire, tant que la solution possède des incohérences, nous localisons le match en apportant le plus d'incohérences, partant de ce pivot nous générons ces voisins(soit sur la colonne dans la matrice, soit sur toute la matrice) et calculons leur nombre d'incohérences, puis sélectionnons le ou l'un des meilleurs individus qui deviendra la solution courante. La recherche continue ainsi tant que la solution n'est pas parfaite ou qu'un nombre d'itérations prédéfinies n'est pas atteint.

Le problème du STS présentant une grande quantité d'optimum locaux, nous avons modifié le fonctionnement de base en permettant à l'algorithme de continuer même s'il ne possède pas de voisin améliorant. De plus, nous avons modifié le comportement pour ajouter de l'aléatoire dans le choix du voisin (tout de même en choisissant parmi les 3 meilleurs). Afin de réduire le nombre de voisins générés à chaque tour, seuls les voisins partant du plus mauvais sommet/pivot sont générés.

6.2 Recherche locale avec tabou

La recherche locale avec tabou consiste à garder une liste des éléments ayant été permutés dans les tours précédents afin d'éviter de les réutiliser comme pivot pour la permutation durant quelques tours afin de ne pas rester bloqué dans des minimums locaux .

L'algorithme de la recherche locale avec tabou a exactement le même fonctionnement que l'algorithme de recherche locale excepté que lors du choix du plus mauvais sommet/pivot, nous vérifions au préalable que celui ci n'a pas été permuté lors des précédents tours.

6.3 Recherche locale avec tabou modifiée

Étant donné que la recherche locale avec tabou n'arrive pas à résoudre le problème avec plus de 12 équipes (voir partie suivante), nous avons cherché à améliorer son fonctionnement. Le principe est donc de commencer avec une recherche locale avec tabou pour un nombre limité de permutations en satisfaisant toutes les contraintes de manière à résoudre partiellement le problème. Puis à chercher à satisfaire uniquement la contrainte sur les semaines. Une fois la contrainte sur les semaines valide, nous passons à la satisfaction de la contrainte sur les périodes en ne modifiant que les périodes.

6.4 Algorithmes génétiques

Les algorithmes génétiques sont des outils d'optimisation utilisant une population d'individus représentant le problème étudié. Lors de la recherche les individus seront modifiés et croisés jusqu'à atteindre l'objectif recherché ou une version plus optimisée de la solution de base. L'algorithme utilise quatre étapes principales décrites dans le schémas ci-dessous.

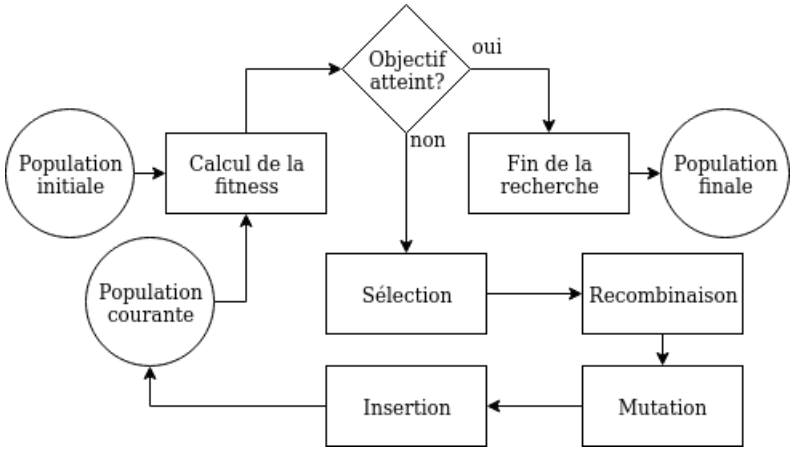

FIGURE 3 – Fonctionnement de l'algorithme génétique : L'algorithme commence avec une population initiale avec des plannings aléatoires. La fitness des individus est calculée une première fois. Tant que l'objectif n'est pas atteint, nous utilisons un opérateur de sélection d'individus sur la population. Les individus choisis seront ensuite croisés (création d'individus "enfants", mélange des caractéristiques d'individus "parents") grâce à des opérateurs de croisement(recombinaison). Les individus créés lors du croisement seront ensuite mutés grâce à des opérateurs de mutation. Ils seront ensuite insérés dans la population en retirant les individus les plus vieux ou les moins bons. Lorsque l'objectif est atteint (limite du nombre de générations ou fitness maximum atteinte) alors la recherche est terminée et nous pouvons analyser les individus obtenus.

Le coeur de l'algorithme génétique n'est pas implémenté dans ce projet mais a été codé dans le cadre du cours d'algorithmes intelligents pour l'aide à la décision, son code est disponible sur github à l'adresse `https://github.com/Cyril-Grl/AlgoGen` et le package est disponible sur `pypi.org` avec la commande `pip install algo-gen`. Vous trouverez cependant l'implémentation des individus utilisés dans le github de ce projet.

Pour les tests, nous avons utilisés les caractéristiques suivantes :

— Taille de la population : 100.

— Nombre de tours maximum : 1000.
— Opérateur de sélection : sélection par tournoi, 2 parmi 5.
— Proportion de sélection : 20% de la population.
— Opérateur de croisement : PMX, Partially-mapped crossover, le croisement d'ordre 1 étant aussi implémenté mais se révélant très inefficace pour ce problème voir graphique ci-dessous.
— Proportion de croisement : 100% des individus sélectionnés.
— Opérateur de mutation : insertion, sélection de deux matchs dans l'individu, le second match devient le suivant du premier ou échange, sélection et inversion de la position de deux matchs.
— Proportion de mutation : 40% des individus sélectionnés.
— Opérateur d'insertion : par âge, les plus anciens individus sont remplacés par les plus récents.
— Lorsque la diversité sera trop faible, des individus aléatoires seront réinsérés dans la population.

Voici un test réalisé avec 8 équipes pour repérer les opérateurs adaptés à la résolution de ce problème (dans ce cas seuls 2 individus étaient sélectionnés à chaque tour, d'où le nombre de tours conséquent, ces courbes sont la moyennes de 20 différentes exécutions). À gauche en utilisant un croisement d'ordre 1, à droite le croisement PMX :

FIGURE 4 – Le croisement d'ordre 1 s'avère inefficace pour résoudre ce problème peu importe l'opérateur de mutation. Pour le PMX, la mutation par insertion et par échange semblent être bien supérieures au deux autres méthodes de mutation implémentées.

6.5 Algorithmes mémétiques

Les algorithmes mémétiques sont nés de l'hybridation des algorithmes génétiques et de la recherche locale. Ceux-ci remplacent l'opérateur de mutation par une recherche locale. Dans notre implémentation, nous avons utilisé un nombre limité de permutations lors de la recherche locale ($n * 20$ permutations).

6.6 Algorithmes génétiques et mémétiques

Pour essayer d'améliorer les deux méthodes ci dessus nous avons tenté de les mélanger, la recherche locale poussant souvent les individus dans des minimum locaux. Nous avons mis en place des solutions pour alterner l'utilisation des opérateurs de mutation par insertion et échange et utiliser moins fréquemment la recherche locale lors de la mutation.

7 Tests et résultats

Pour tester nos méthodes, nous avons réalisé 100 lancés avec différentes graines du générateur aléatoire pour voir le taux de réussite et le temps demandé pour arriver à ce résultat. Pour chaque méthode, le nombre de gauche correspond au nombre de fois où l'algorithme à réussi à résoudre le problème, celui de droite, quant à lui, le temps utilisé pour réaliser les 100 différentes exécutions.

Pour les algorithmes mémétiques, le temps d'exécution devenant trop long, nous sommes passé à simplement 2 exécutions pour 14 et 16 équipes. Pour les mémétiques modifiés, seulement 10.

nombre d'équipes	Recherche locale		Recherche locale avec tabou		Recherche locale modifiée		Algorithmes génetiques		Algorithmes mémétiques		Algorithmes mémétiques modifiés	
6	97	>1s	95	>1s	100	>1s	75	3min	100	6s	100	3s
8	89	7s	100	2s	97	2s	18	9min	100	40s	100	11s
10	28	68s	86	35s	98	22s	0	15min	100	4min	100	159s
12	3	4min	21	183s	24	86s	-	-	100	2h	100	1h30
14	0	16min	0	6min30	1	193s	-	-	2/2	1h15	5/10	1h45
16	-	-	-	-	1	6min	-	-	0/2	15h	0/10	18h45

Comme vous pouvez le constater, les algorithmes de recherche locale sont beaucoup plus rapides que les algorithmes génétiques, cependant ils sont moins fiables pour trouver de bonnes solutions.

Passées 12 équipes, la recherche locale classique et avec tabou n'arrivent pas à satisfaire le problème, bien que la recherche locale avec tabou soit plus performante. La modification apportée à ces algorithmes (chercher à résoudre le problème de manière générale, puis satisfaire la contrainte sur les semaines, puis sur les périodes) permet à la fois d'accélérer le processus de résolution et d'obtenir des solutions jusqu'à 16 équipes.

Les algorithmes génétiques seuls ne sont pas capables avec les opérateurs implémentés de dépasser les 8 équipes. Cependant, l'utilisation de la recherche locale à la place de la mutation classique donne de bons résultats plus rapidement et de manière fiable.

8 Conclusion

Ce projet m'a permis de découvrir les algorithmes de recherche locale et mémétiques mais aussi de chercher à améliorer un processus de résolution de problèmes pour résoudre des instances plus grandes et plus rapidement.

Les tests effectués ont pu montrer que les modifications apportées ont contribué à une meilleure résolution du problème. Il reste cependant beaucoup à modifier. À la fois pour réduire le temps de recherche, en particulier pour les algorithmes génétiques, mais aussi pour trouver des méthodes permettant d'éviter les optimums locaux en particulier pour la recherche locale. Nous avons remarqué au cours des tests qu'il était très difficile d'arriver à une solution correcte en partant d'un planning satisfaisant les contraintes sur les semaines et en utilisant uniquement des permutations affectant les périodes pour satisfaire la contrainte sur celle-ci. D'où cette question, toute configuration du planning satisfaisant la contrainte sur les semaines peut-elle permettre, en modifiant uniquement l'ordre des matchs dans la semaine, d'atteindre une configuration satisfaisant toutes les contraintes ?

Les algorithmes génétiques pourraient aussi être modifiés. Nous avons montré que le croisement d'ordre 1 est inefficace pour résoudre ce problème alors que le PMX se montre plus adapté. Il existe probablement des croisements tout aussi adaptés pour ce genre de problème. La question d'opérateurs de mutation plus adaptés serait aussi envisageable.

Références

[1] Jean-Philippe Hamiez and Jin-Kao Hao. A linear-time algorithm to solve the sports league scheduling problem (prob026 of csplib). *Discrete Applied Mathematics*, 143 :252–265, 09 2004.

[2] Jean-Philippe Hamiez and Jin-Kao Hao. Sports league scheduling : Enumerative search for prob026 from csplib. In *12th International Conference on Principles and Practice of Constraint Programming (CP)*, Nantes, 25-29/9/2006 2006.

[3] Toby Walsh. CSPLib problem 026 : Sports tournament scheduling. `http://www.csplib.org/Problems/prob026`.