UNIVERSITÉ D'ANGERS

Projet Métaheuristique

Étude d'algorithmes pour la résolution d'un problème de planification sportive

Lepinette Cyril

12 février 2020

TABLE DES MATIÈRES

1 INTRODUCTION

Le code source du projet est disponible à l'adresse :

`https://github.com/KyrillosL/tabu_sport_scheduling_tournament`

Pour ce projet, le langage que j'ai choisi est python car il dispose de nombreuses bibliothèques pour l'intelligence artificielle. Le problème proposé à la résolution est le Sports Tournament Scheduling, abrégé "STS".
Il est référencé au numéro 26 sur le site csplib : `http://www.csplib.org/Problems/prob026/`
Le STS consiste à faire rencontrer un nombre *n* d'équipes sur *n-1* semaines, chacune divisée en $\frac{n}{2}$ périodes.

3 contraintes doivent être satisfaites :

1. Chaque équipe joue exactement une fois par semaine : les équipes apparaissent une fois par ligne.
2. Chaque équipe doit se rencontrer : toutes les combinaisons sont dans le tableau
3. Une équipe ne peut pas jouer plus de deux fois par période : une équipe n'apparaît pas plus de deux fois par colonne.

Voici un exemple du problème pour 6 équipes :

		Période 1	Période 2	Période 3
Semaine	1	2-4	1-5	0-3
	2	0-5	3-4	1-2
	3	3-5	0-2	1-4
	4	1-3	0-4	2-5
	5	0-1	2-3	4-5

Pour l'ensemble des algorithmes présentés ci-dessous une configuration initiale constructiviste satisfaisant les deux premières contraintes a été généré grâce à la méthode présenté

dans [1] à la partie : *4.2 Initial Configuration*
Elle consiste à représenter le problème sous forme de graphe dont chaque sommet est une rencontre et chaque arrête est la semaine accueillant le match. Voici un schéma récapitulatif :

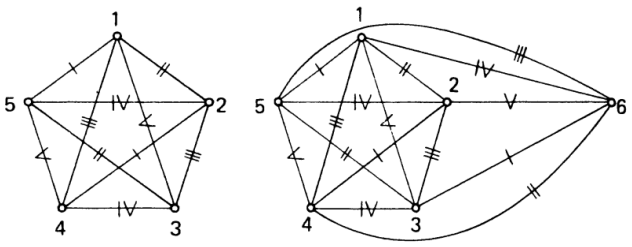


FIGURE 1.1 – Graphe permettant de satisfaire les deux premières contraintes. Ici l'équipe 1 joue contre la 2 durant la semaine 2, et la 5 contre la 6 en semaine 3

Ainsi, il nous suffira de procéder à des permutations de rencontres entre deux périodes dans une même semaine (afin de ne pas casser la première contrainte).

L'évaluation d'une configuration dépendra de l'algorithme utilisé, mais se basera toujours sur celle décrite dans [1] : On compte le nombre d'occurrences d'une équipe dans une période et on incrémente le résultat pour les équipes apparaissant plus de deux fois. Ainsi, une solution au problème est une configuration ayant une évaluation égale à 0. Voici la formule :

$$f(s) = f_p(s) = \sum_{p=1}^{|P|} \sum_{n=1}^{|T|} \chi_P(s,n,p),$$

$$\chi_P(s,n,p) = \begin{cases} 0 & \text{if } \mathrm{Occ}P_S(p,t_n) \leq 2, \\ \mathrm{Occ}P_S(p,t_n) - 2 & \text{otherwise.} \end{cases}$$

FIGURE 1.2 – Fonction d'évaluation

Les différents algorithmes étudiés lors de ce projet s'inscrivent dans des méta-heuristiques de recherches locales et génétiques. Une comparaison avec un modèle SAT est également fournie.

2 Recherches Locales

2.1 Hill Climbing

L'algorithme *Hill Climbing*, relativement simple est une bonne introduction aux méthodes de recherches locales.

En partant d'une configuration initiale *CI*, l'algorithme va chercher dans son voisinage (c'est à dire dans toutes les configurations possibles accessibles à partir de la configuration actuelle) la meilleure configuration. Ceci est permis grâce à une fonction évaluant une configuration. Dans notre cas, c'est celle décrite dans [1].

Le problème de cet algorithme est qu'il n'a pas de phase de diversification : il ne fait qu'exploiter. De ce fait, il s'arrête au premier optimum local rencontré. Notre problème ayant des optimums locaux, cet algorithme n'est donc pas approprié. Dans mon cas, peu importe le nombre d'équipe, il est resté dans des optimums locaux. Par exemple, pour 6 équipe, il est resté bloqué à un score de 3. Dans la suite nous étudierons différentes améliorations afin de pallier à ce problème.

2.2 Random Walk

Une première variation du Hill Climbing est de ne plus prendre la meilleure configuration possible à chaque itération, mais une configuration aléatoire. De ce fait, nous ne sommes plus dans une exploitation seulement, mais dans une exploration seulement. Nous ne tombons plus dans un optimum local et nous trouvons des solutions. En théorie, avec un nombre d'itérations infini nous pourrions tomber sur le bon résultat par hasard, mais cela n'est pas possible en pratique.
J'ai obtenu 100% de succès pour un pour un nombre maximum d'itérations de 1 000 000 (sur 20 essais) jusqu'à 8 équipes, mais aucune solution pour 10 et plus.

TABLE 2.1 – Algorithme Random Walk, pour kmax = 100000

Nombre d'équipes	Temps
6	0.01
8	5.87

2.3 RECHERCHE TABOU

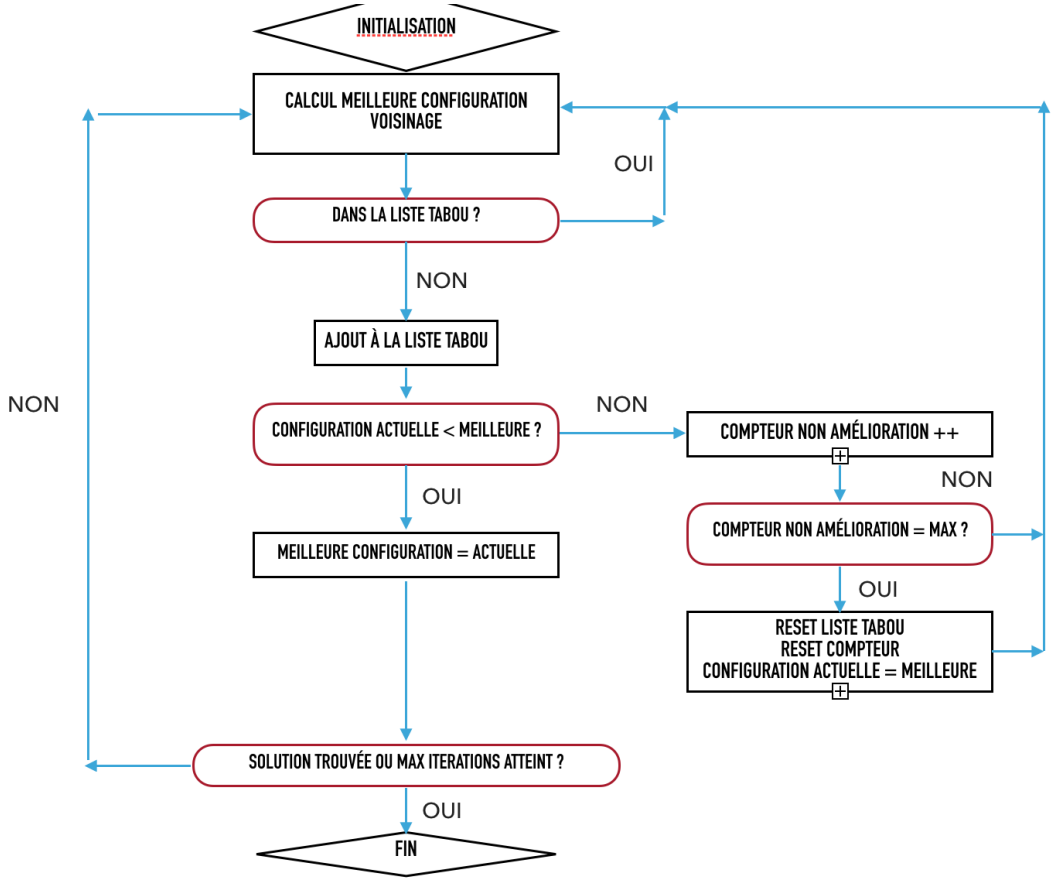
2.3.1 Présentation et choix des paramètres

Une autre approche de recherche locale développée pour ce projet est *tabou*. C'est une implémentation de l'algorithme décrit dans [1]. Contrairement au *Hill Climbing*, il introduit une mémoire permettant deux choses :

1. Ne pas retourner sur les configurations déjà visités pendant un certain temps.
2. Relancer la diversification en cas d'exploitation non productive.

Voici son fonctionnement : tant qu'un nombre maximum d'itérations n'a pas été effectué et qu'une solution n'a pas été trouvée :

1. Calcul de la meilleure configuration dans le voisinage
2. Tant que la meilleure configuration est dans la liste tabou, alors l'enlever du voisinage et prendre la suivante
3. Ajouter la configuration à la liste Tabou
4. Si la nouvelle configuration est améliorante alors elle devient la meilleure, sinon incrémenter le compteur de non amélioration
5. Si le compteur de non amélioration est atteint, alors vider la liste tabou, lui assigner une nouvelle taille puis la configuration actuelle prend la meilleure configuration



Ainsi, si la recherche tombe dans un optimum local (non amélioré depuis trop longtemps), alors on revient à la meilleure solution précédente trouvée (diversification) pour tenter de l'exploiter.

Plusieurs hyper-paramètres peuvent être définis, et notamment le temps nécessaire avant de réinitialiser la liste tabou.

Pour ce problème, j'ai choisi :

$$reset\_time = \min\left(taille\_voisinage * \frac{nombre\_semaines}{6}, 2000\right)$$

De ce fait, plus le problème est petit (et donc l'espace de recherche), plus nous autorisons une exploitation profonde, car peu coûteuse. Par contre, si le problème est grand, alors nous n'exploiterons qu'une partie de l'espace de recherche. Cependant, pour économiser des coûts de calculs nous bornons à 2000 la taille maximale de la liste.

Un autre paramètre pouvant être défini est la taille de la liste tabou. Le papier [1] préconise une valeur aléatoire entre 4 et 100 mais j'ai remarqué que pour des équipes plus grande (> 14), une liste tabou de taille > 100 donnait de meilleurs résultats dans l'ensemble. Pour un nombre d'équipe > 6 j'ai donc décidé de choisir une taille relative aux nombre d'équipes. Cette taille aléatoire est recalculée et mise à jour à chaque fois que l'on vide la liste tabou.

$$size\_tabu\_list = random(10, nombre\_equipes * 50)$$

Concernant l'évaluation d'une configuration, c'est la formule décrite dans l'introduction.

2.3.2 Résultats

Voici les résultats obtenus, pour une taille de la liste tabou fixé à 10, moyenné sur 20 itérations :

TABLE 2.2

Nombre d'équipes	Temps (s)
6	0.31
8	0.10
10	1.75
12	124.69

Voici les résultats obtenus, pour une taille de la liste tabou aléatoire, moyenné sur 20 itérations :

TABLE 2.3

Nombre d'équipes	Temps (s)
6	4.32
8	0.17
10	1.96
12	25.03
14	81
16	375

Nous remarquons donc que les hyperparamètres jouent un rôle essentiel dans la résolution du problème. En effet, il y a un facteur de 10 pour 6 équipes et un facteur de 5 pour 12.

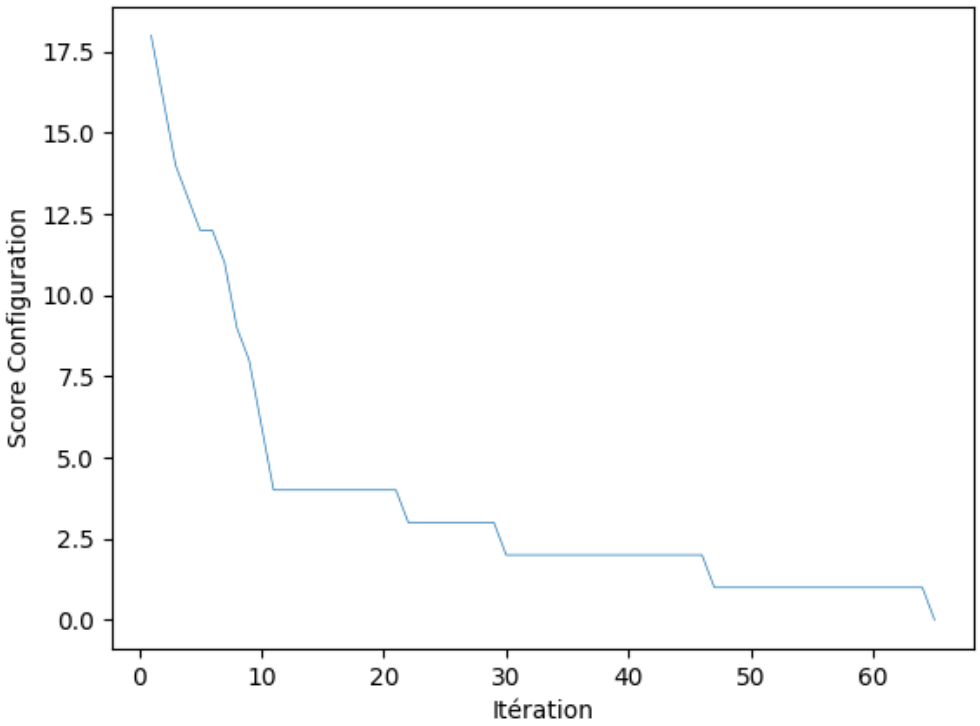


FIGURE 2.1 – Evolution du score en fonction du nombre d'itérations

On peut remarquer les phases d'exploitations (score constant), et qu'il ne remonte pas : lorsqu'on vide la liste tabu, la configuration actuelle reprend la meilleure.

2.4 Recuit simulé

2.4.1 Présentation

Tout comme la recherche tabou et contrairement à l'algorithme *Hill Climbing*, le recuit simulé permet de faire varier l'exploitation et la diversification (exploration) en fonctions de certains paramètres, dont deux principaux :

1. Température : Plus l'algorithme avance, plus on tend à exploiter une configuration se rapprochant de la solution.
2. Énergie : En fonction du score de la configuration obtenue, et même si celle ci est dégradante, nous pouvons l'accepter avec une certaine probabilité. (exploration)

Voici son fonctionnement : Tant qu'un nombre maximum d'itérations n'a pas été effectué et qu'une solution n'a pas été trouvée :

1. Prendre un voisin aléatoirement dans le voisinage
2. Calculer sa probabilité d'être accepté en fonction de son énergie et de sa température
3. S'il est améliorant ou que sa probabilité est > a un nombre aléatoire compris entre 0 et 1, alors la meilleure configuration prend l'actuelle.

L'évaluation est celle décrite dans l'introduction, cependant elle a été adaptée. En effet, afin de garder des valeurs entre 0 et 1, nécessaire pour le calcul de l'énergie, j'ai divisé le score obtenu par le score maximal atteignable. Ceci est nécessaire, car le calcul de l'énergie est très sensible. Par exemple, pour 6 équipes nous avons des score en général inférieurs à 18, tandis que pour 20 le score peut monter à Plus de 50. La normalisation est donc nécessaire afin de ne pas fausser les calculs par le nombre d'équipes.

2.4.2 Calcul de l'évaluation

$$f(s) = f_p(s) = \sum_{p=1}^{|P|} \sum_{n=1}^{|T|} \chi_P(s,n,p),$$

$$\chi_P(s,n,p) = \begin{cases} 0 & \text{if } \mathrm{OccP_S}(p,t_n) \leq 2, \\ \mathrm{OccP_S}(p,t_n) - 2 & \text{otherwise.} \end{cases}$$

$$évaluation = \frac{f(s)}{score\ maximal}$$

2.4.3 Calcul de l'énergie

$$énergie = (eval\_courante + \frac{best\_eval - eval\_courante}{4}) * 2$$

L'énergie est calculée en fonction de deux paramètres : l'évaluation courante ainsi que la distance avec la meilleure évaluation. La soustraction entre l'évaluation courante et la meilleure évaluation locale permet de favoriser les configurations peu dégradantes, relativement à la meilleure évaluation locale. Par exemple, une configuration ayant une meilleure évaluation locale de 0.5 et une évaluation courante de 0.6 aura plus de chance d'être acceptée qu'une ayant une évaluation courante de 0.9.

De plus, l'addition de l'évaluation courante (premier terme) permet de plus favoriser les configurations proches de la meilleure et creuser l'écart avec les plus éloignées. Les facteurs 4 et 2 sont des poids (empiriques) associés à ces évaluations. De ce fait, peu importe la distance entre la meilleure évaluation et la courante, si celle est trop haute, alors elle aura peu de chance d'être acceptée.

Plus la distance entre les configurations sont grandes, moins elles sont acceptées grâce au calcul de la probabilité qui utilise une négation de l'énergie : plus l'énergie est grande, plus $e^{-energie}$ diminue pour tendre vers 0.

2.4.4 Calcul de la Température

$$température = (1 - (\frac{i}{nombre\_max\_iterations})) * 10$$

La température permet de baisser l'exploration au fil de la recherche et ainsi la faire converger vers un optimum. Au départ, une grande part de configurations dégradantes sont acceptées afin de favoriser l'exploration car la température est élevée. En effet, celle ci fait baisser le calcul de l'exponentielle vers 0 et pousse donc la probabilité à tendre vers 1.

2.4.5 Calcul de la probabilité

$$probabilité = e^{-\frac{énergie}{température}}$$

Voici le calcul final de la probabilité qu'a une configuration d'être acceptée.
Pour résumer : plus on a d'énergie (configuration non améliorante), moins celle ci est acceptée, et plus la température baisse, moins une configuration a de chance d'être acceptée également.

2.4.6 Résultats et améliorations

N'ayant pas obtenu de résultats satisfaisant avec cette méthode, j'ai tenté d'ajouter un critère d'exploitation, car l'exploration est grande au début. En effet, tout comme pour la recherche tabou, j'ai ajouté un compteur de non amélioration au bout du coup nous pouvons accepter une configuration non améliorante.

J'ai fixé le nombre maximal de configurations non améliorantes (exploration) avant de passer à l'exploration à :

$$enable\_exploration\_after = \min\left(taille\_voisinage * \frac{nombre\_semaines}{6}, 2000\right)$$

De ce fait, pendant n itération l'algorithme ne va pas explorer, même si sa probabilité est supérieure au nombre aléatoire généré. Il n'est autorisé à explorer qu'après un certain

nombre d'itérations. Cela est différents de baisser la température, car elle ne ferait que diminuer l'exploration globalement. On veut garder l'exploration grande, mais après une phase d'exploitation. Cependant, les résultats restent peu convaincants.

TABLE 2.4 – Résultat du recuit simulé sur 20 itérations

Nombre d'équipes	Temps
6	0.01
8	11.295

On peut remarquer ici, au contraire de la recherche tabou que l'on peut dégrader le score (exploration).

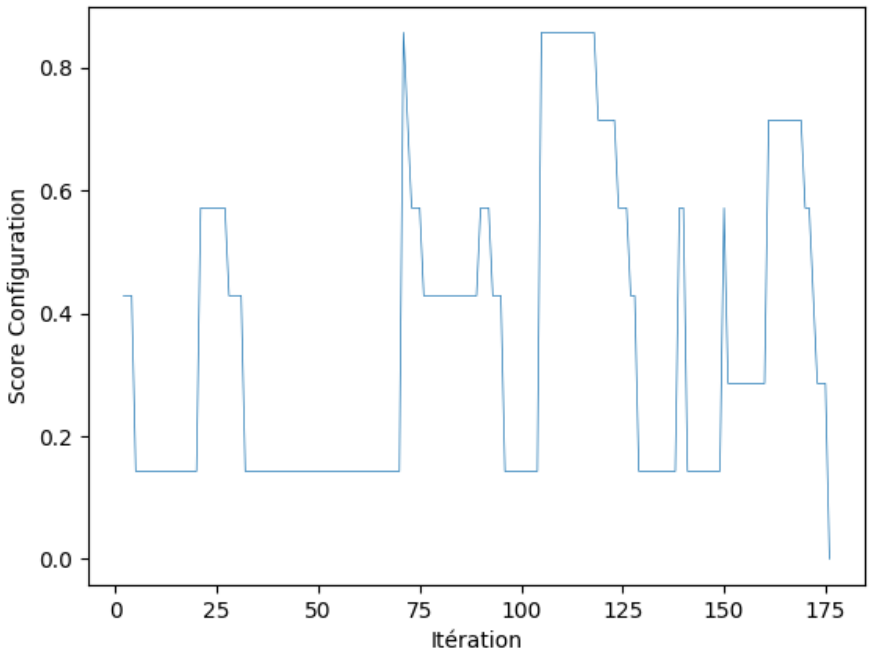


FIGURE 2.2 – Evolution du score en fonction du nombre d'itérations

3 ALGORITHME GÉNÉTIQUE

3.1 PRÉSENTATION

L'algorithme génétique présenté ci-dessous est celui développé dans le cadre du cours *algorithmes intelligents pour l'aide à la décision.*
Le projet est le rapport sont disponibles à l'adresse :

`https://github.com/KyrillosL/AlgorithmeGenetique`

Dans notre cas il y a 3 mutateurs :

1. Inverser 1 période dans 1 semaine
2. Inverser 3 période dans 1 semaine
3. Inverser 5 période dans 1 semaine

Le crossover entre deux individus pouvant supprimer la contrainte précédemment satisfaite sur les semaines, je ne l'ai pas implémenté.

Ensuite, une heuristique est choisie pour sélectionner quel opérateur est le plus efficace au cours du temps.J'ai implémenté :

1. opérateur fixe
2. oracle myope
3. adaptive wheel
4. adaptive pursuit
5. ucb
6. exp3

La sélection des individus/parents à muter est possible de 3 manières :

1. N meilleurs individus
2. N meilleurs individus issus d'un tournois en i itérations

La sélection des individus/parents à retirer de la population :

1. N pires individus
2. N individus plus anciens

Pour ce problème et après quelques expérimentation j'ai choisis d'utiliser l'adaptive pursuit, avec une sélections par N meilleurs individus (le tournois donnant également de bons résultats) et un élagage de la population par les pires individus.

3.2 RÉSULTATS

TABLE 3.1

Nombre d'équipes	Temps (s)
6	0.57
8	120

D'après mes expérimentations, les mutateurs 3N et 5N sont dans l'ensemble plus efficaces que le 1N car ils ne tombent pas dans des minimums locaux et favorisent la diversité. Cependant, les résultats généraux restent décevants. Il faudrait plus de travail de réglages de paramètres pour améliorer ce modèle.

Voici un exemple de courbes représentants l'évolution des populations. Le score et les probabilités d'opérateurs sont fournies.

exploration moyenne (b=0.5) | 1000 individus de taille 6

Avec une exploration moyenne l'algorithme favorise tout de même le mutateur 5N.

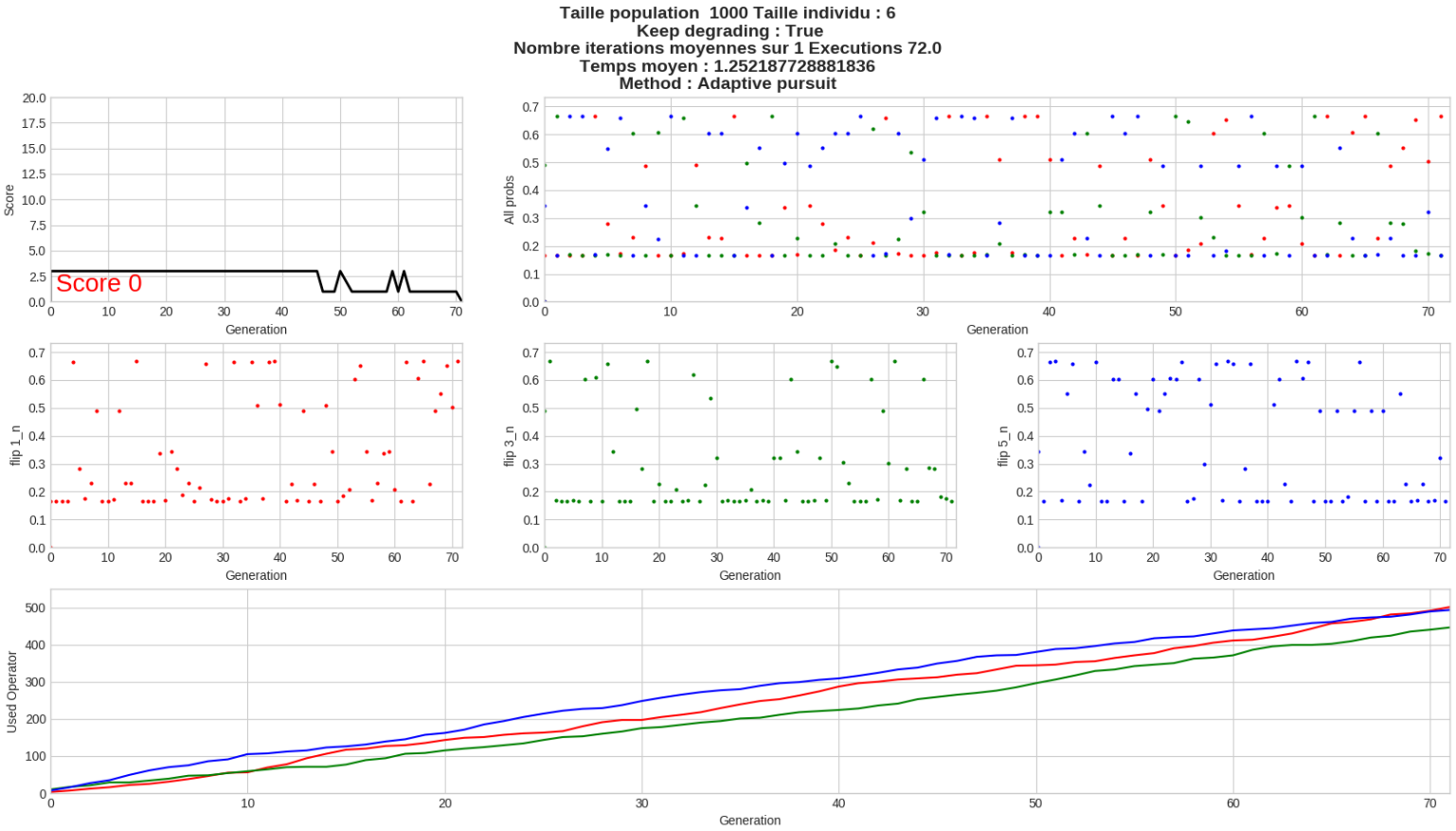


FIGURE 3.1 – Adaptive Pursuit avec une exploration moyenne

4 MODÈLE SAT

4.1 MODÈLE

Dans le cadre du cours de programmation par contraintes j'ai modélisé le problème en une instance SAT. Il est donc intéressant de comparer les résultats avec les algorithmes décrits précédemment.

Une rencontre (match) entre deux équipes est modélisée de la sorte :

$$M_{wpsn}$$

1. w - semaine
2. p - période
3. s - Slot (Équipe 1 / Équipe 2)
4. n - Numéro de l'équipe

Pour des raisons de lisibilité, dans certains cas je n'écrirai pas la forme normale conjonctive. Elle est disponible dans les fichiers source du projet.

Au moins une équipe par slot :

$$\bigwedge_{w}\bigwedge_{p}\bigwedge_{s=1}^{2}\bigvee_{n} M_{wpsn}$$

Au plus une équipe par slot :

$$\bigwedge_{w}\bigwedge_{p}\bigwedge_{s=1}^{2}\bigwedge_{n1<n2} \neg M_{wpsn1} \vee \neg M_{wpsn2}$$

Une équipe joue contre une autre équipe :

$$\bigwedge_{w}\bigwedge_{p}\bigwedge_{s1<s2}\bigwedge_{n} \neg M_{wps1n} \vee \neg M_{wps2n}$$

Pas deux fois le même match dans la même période :

$$\bigwedge_{w1<w2}\bigwedge_{p}\bigwedge_{s1<s2}\bigwedge_{n1<n2} (M_{w1ps1n1} \wedge M_{w1ps2n2}) \rightarrow \left[(\neg M_{w2ps1n1} \vee \neg M_{w2ps2n2}) \wedge (\neg M_{w2ps2n1} \vee \neg M_{w2ps1n2})\right]$$

Pas deux fois le même match dans la même semaine :

$$\bigwedge_{w}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1<n2}(M_{wp1s1n1}\wedge M_{wp1s2n2})\rightarrow\left[(\neg M_{wp2s1n1}\vee\neg M_{wp2s2n2})\wedge(\neg M_{wp2s2n1}\vee\neg M_{wp2s1n2})\right]$$

Pas deux fois le même match dans une semaine et une période différente.

$$\bigwedge_{w1<w2}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1<n2}(M_{w1p1s1n1}\wedge M_{w1p1s2n2})\rightarrow\left[(\neg M_{w2p2s1n1}\vee\neg M_{w2p2s2n2})\wedge(\neg M_{w2p2s2n1}\vee\neg M_{w2p2s1n2})\right]$$

$$\bigwedge_{w1<w2}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1<n2}(M_{w2p1s1n1}\wedge M_{w2p1s2n2})\rightarrow\left[(\neg M_{w1p2s1n1}\vee\neg M_{w1p2s2n2})\wedge(\neg M_{w1p2s2n1}\vee\neg M_{w1p2s1n2})\right]$$

Une équipe ne joue qu'une seule fois par semaine :

$$\bigwedge_{w}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1}M_{wp1s1n1}\rightarrow\left(\neg M_{wp1s2n1}\wedge\neg M_{wp2s1n1}\wedge\neg M_{wp2s2n1}\right)$$

$$\bigwedge_{w}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1}M_{wp1s2n1}\rightarrow\neg M_{wp2s1n1}$$

$$\bigwedge_{w}\bigwedge_{p1<p2}\bigwedge_{s1<s2}\bigwedge_{n1}M_{wp1s2n1}\rightarrow\neg M_{wp2s2n1}$$

Une équipe joue au maximum deux fois par période :

$$\bigwedge_{w1<w2<w3}\bigwedge_{p}\bigwedge_{s1<s2}\bigwedge_{n}\left[\left(M_{w1p1s1n}\vee M_{w1p1s2n}\right)\wedge\left(M_{w2p1s1n}\vee M_{w2p1s2n}\right)\right]\rightarrow\neg\left(M_{w3p1s1n}\vee M_{w3p1s2n}\right)$$

Une symétrie de relation d'ordre a été cassée : Les équipes dans le slot 1 on un numéro inférieur à celle dans le slot 2.

Dans un premier temps, et pour tester de casser une symétrie, nous avons interdit la dernière équipe de jouer dans le slot 1 :

nde = dernière équipe

s2 = slot 1

$$\bigwedge_{w}\bigwedge_{p}\neg M_{wps1nde}$$

Si une équipe joue dans le slot 1, alors les équipes ayant un numéro inférieures ne peuvent pas jouer dans le slot 2

s1 = slot 1

s2 = slot 2

t = (nombre d'équipes -1)

$$\bigwedge_{w}\bigwedge_{p}\bigwedge_{i=1}^{t}\bigwedge_{j=1}^{i+1} M_{wps1i} \rightarrow \neg M_{wps2j}$$

4.2 Résultats

Voici les résultats obtenus jusqu'à 12 équipes. Nous n'avons pas fait plus, car la conversion dnf->cnf était trop longue.

Table 4.1

	symétries cassées			avec symétries		
	v	c	time	v	c	time
6	180	8100	0.01	180	10020	0.02
8	448	51590	0.06	448	60872	0.32
10	900	210255	0.3	900	240690	2.67
12	1580	651588	2.6	1580	732336	17.424

5 Conclusion et comparaison avec l'état de l'art

Nous sommes très loin des résultats obtenus dans l'état de l'art. En effet, l'article [1], qui met en avant une liste tabou réussi à trouver une solution pour 40 équipes en moins d'une minute. L'énumérative search, présenté dans [2] met 2 secondes, pour 40 équipes également. Cette méthode arrive même à résoudre le problème pour 70 équipes en 3h.

TABLE 5.1 – Comparatif des résultats avec l'état de l'art

nb teams	RandomWalk	Tabou	Recuit simulé	Génétique	SAT	Tabou (état de l'art)	Enum Search
6	0.01	0.31	0.01	0.57	0.01		
8	5.87	0.1	11.295	120	0.06		
10		1.75			0.3		
12		25			2.6		
14		81					
16		375				0.5	
40						53	<1
70							8000

Afin d'améliorer notre modèle présenté dans ce rapport, nous pourrions changer la modélisation en elle-même : ne plus utiliser de classes, mais de simples tableaux contenants des entiers. Ceci minimiserait le coûts des calculs et permettrait de faire plus d'itérations. Ce serait moins lisible, mais plus efficace. De même, le choix du langage à C, plus robuste et plus rapide. Également, de plus amples recherches autour de l'algorithme devraient être réalisé, afin d'obtenir des résultats proches de ceux obtenus dans [1] pour la recherche tabou.

Concernant les expérimentations il aurait été intéressant d'essayer des méthodes hybrides, tel qu'introduire notre recherche tabou au sein d'une population dans un algorithme génétique.

RÉFÉRENCES

[1] Jean-Philippe Hamiez and Jin-Kao Hao. Solving the sportd league scheduling problem with tabu search. *Lecture Notes in Computer Science*, 2148, 10 2001.

[2] Jean-Philippe Hamiez and Jin-Kao Hao. Using solution properties within an enumerative search to solve a sports league scheduling problem. *Discrete Applied Mathematics*, 2018.