Algorithmique Avancée

Projet 2 : Tournoi de combats

Axel PIGEON
axel.pigeon@univ-tlse3.fr

Contexte — *Ce projet porte sur la résolution d'un problème d'optimisation combinatoire appliqué à l'organisation d'un tournoi de combats. L'objectif est de maximiser le gain net d'une équipe de combattants opposée à une équipe d'hôtes, sous des contraintes strictes de ressources et de capacités. Le modèle intègre une gestion de budget énergétique global, des pénalités pour les combats évités, ainsi que des mécanismes stratégiques tels que la désignation d'un capitaine (bénéficiant d'un bonus de compétence) ou l'usage d'une carte "Joker" (doublant les enjeux).*
La problématique réside dans l'appariement optimal des duels, où chaque combattant dispose d'un nombre limité d'actions et chaque hôte possède des valeurs de profit et de perte distinctes. Ce rapport détaille l'approche algorithmique choisie, les structures de données mises en œuvre pour traiter les instances du problème, ainsi que l'analyse de la complexité de notre solution face à l'explosion des combinaisons possibles.

Introduction

Nous détaillons ici deux méthodes utilisées pour la résolution de différentes instances de ce problème contenant respectivement 6, 20 et 100 combattants. Nous nous intéressons d'abord à la résolution grâce au solveur SCIP en modélisant le problème sous forme de Programme Linéaire en Nombres Entiers. Cette approche nous permettra d'obtenir des solutions optimales. Dans un second temps, nous nous intéresserons à une approche heuristique. L'idée sera ici de déterminer une solution proche de l'optimal grâce à une marche aléatoire type recuit simulé sur l'ensemble des solutions.

1 Programmation en nombres entiers

Pour résoudre le problème de manière exacte, nous avons modélisé le tournoi sous la forme d'un *Programme Linéaire en Nombres Entiers (PLNE)*. Cette approche garantit l'obtention de l'optimum global pour les instances de taille raisonnable en explorant l'ensemble des solutions possibles via l'algorithme de *Branch-and-Bound*. La version du problème résolut ici est celle comportant les rôles de Capitaine et de Joker.

Variables de décision. Le modèle repose sur une matrice de variables binaires $x_{i,j}$ telle que :

- $x_{i,j} = 1$ si le combattant i affronte l'hôte j, 0 sinon.
- $c_i \in \{0, 1\}$: variable binaire indiquant si le combattant i est désigné Capitaine.
- $j_i \in \{0, 1\}$: variable binaire indiquant si le combattant i possède le Joker.

La matrice $(g_{i,j})$ représente le gain brut de chaque combat calculé en amont de la résolution.

Objectif et Linéarisation. L'objectif est de maximiser le gain net, incluant les bonus du Capitaine (+5 en compétence) et du Joker (×2 sur les gains et pertes). Pour conserver un modèle linéaire, nous avons utilisé des variables de couplage $actCap_{i,j}$ et $actJok_{i,j}$ pour représenter l'activation des bonus lors d'un combat spécifique. L'objectif s'écrit alors :

$$\max \left(\sum_{i,j} (g_{i,j} \cdot x_{i,j} + \Delta Cap_{i,j} \cdot actCap_{i,j} + \Delta Jok_{i,j} \cdot actJok_{i,j}) - P \cdot \left(n - \sum_{i,j} x_{i,j} \right) \right) \tag{1}$$

Où ΔCap et ΔJok représentent les gains incrémentaux apportés par les bonus respectifs et P est la pénalité déduite pour chacun des n hôtes non combattu.

Contraintes. Les contraintes du modèle assurent la validité de la solution :
- *Capacité des combattants :* $\sum_{j=1}^{n} x_{i,j} \leqslant 2 - c_i$. Un combattant normal effectue au plus 2 combats, le capitaine au plus 1.
- *Unicité de l'hôte :* $\sum_{i=1}^{p} x_{i,j} \leqslant 1$. Chaque hôte ne peut être combattu qu'une seule fois.
- *Budget énergétique :* $\sum_{j=1}^{n} E_j \cdot (\sum_{i=1}^{p} x_{i,j}) \leqslant B$.
- *Rôles uniques :* $\sum_i c_i \leqslant 1$ et $\sum_i j_i \leqslant 1$.

Optimisations pour les grandes instances. Pour l'instance `tournament_100.txt`, le nombre de variables binaires dépasse les 20 000. Pour réduire le temps de calcul, nous avons implémenté :

1. *Priorités de branchement :* Nous avons forcé le solveur à brancher en priorité sur les variables c_i et j_i (`chgVarBranchPriority`), car le choix des rôles structure fortement le reste de la solution.
2. *Relâchement de la précision :* L'acceptation d'une précision d'optimalité de 5% permet d'obtenir une solution de haute qualité en quelques secondes là où l'optimum exact requerrait plusieurs heures.

2 Méta-heuristique

Dans cette section, nous essayons d'approcher une solution optimale au problème sous la forme d'une méta-heuristique. Contrairement à la recherche exhaustive précédente, nous n'essayons pas de considérer toutes les solutions possibles, nous simulons plutôt une marche aléatoire sur l'ensemble des solutions en essayant de trouver un maximum. Pour cela, nous devons définir plusieurs notions telles que le *voisinage* d'une solution, sa *valeur* ainsi que sa *validité*. Comme précédemment, le choix des structures de données sera déterminant pour réduire le temps de calcul et pouvoir traiter de grosses instances.

Structures de données. Commençons par introduire les structures de données utilisées pour modéliser le problème. L'objectif est de trouver une combinaison de combats qui maximise les gains de l'équipe des combattants. On doit donc affecter à chacun des p combattants un ou plusieurs des n hôtes. Pour chaque solution possible, nous devons être capables d'évaluer sa valeur très rapidement. Il nous semble donc judicieux de vectoriser toutes les données du problème pour n'avoir qu'à effectuer des produits matriciels pour l'évaluation et la vérification. Nous avons donc :
- C : La matrice représentant une solution au problème telle que $(i, j) = 1$ ssi le combattant i affronte l'hôte j et 0 sinon.
- E : Un vecteur qui représente l'énergie nécessaire pour combattre chacun des n hôtes.
- W : Une matrice représentant le gain possible de chaque combat, $(i, j) = W_j$ si le combattant i bat l'hôte j lors d'un combat.
- L : Une matrice représentant la perte de possible de chaque combat, $(i, j) = L_j$ si le combattant i est battu par l'hôte j.
- M : La matrice des gains bruts pour chaque combat possible.

M se calcule très simplement par $M = W - L$. On remarque donc que pour évaluer le gain brut d'un combat (sans pénalités), il suffit d'évaluer :

$$\sum_{j=1}^{n} \sum_{i=1}^{p} C \odot M$$

où $\odot$ est la multiplication termes à termes des deux matrices. On a donc :

$$Tr({}^tCM)_{i,j} = \sum_{j=1}^{n}\sum_{i=1}^{p} C_{i,j}M_{i,j} = \sum_{j=1}^{n}\sum_{i=1}^{p} C \odot M \tag{2}$$

D'autre part, pour calculer le coût total en énergie d'une configuration C, il suffit aussi d'effectuer :

$$\sum_{j=1}^{n}(C \times E). \tag{3}$$

Nous effectuons ce calcul grâce à la fonction `get_energy_value`. Enfin, pour déterminer la valeur de la pénalité à soustraire au gain de chaque configuration, nous devons compter le nombre d'attaquants combattus par hôtes. Soit P la pénalité définie par l'instance, on a donc :

$$P_{totale} = P \times \left(n - \sum_{j=1}^{n} \max_i C_{i,j} \right) \tag{4}$$

Finalement, le gain net V d'une configuration peut être calculé par la formule suivante :

$$V = Tr({}^tCM) - P \times \left(n - \sum_{j=1}^{n} \max_i C_{i,j} \right) \tag{5}$$

Ce calcul est effectué dans `get_sol_value`. La bibliothèque `numpy` est utilisée pour tous les calculs matriciels et les opérations de recherche sur les lignes/colonnes de la matrice C. L'intérêt de cette approche est que les matrices encodant les données du problème sont calculées à l'initialisation de celui-ci et seuls quelques produits matriciels sont utilisés pour l'évaluation. Cependant, pour de très grosses instances du problème, le stockaque de M et E peuvent s'avérer plus contraignant.

Voisinnages. Passons maintenant à la définition du voisinnage d'une solution C. Trois types de pas ont été implémentés :

- *L'échange d'hôte* (`moove_swap_host`) : qui consiste à changer l'hôte combattu par un hôte non combattu pour un attaquant i. Ce pas permet de tester un combattant avec différents hôtes pour maximiser le gain.
- *Le transfert d'hôte* (`moove_shift_contestant`) : qui consiste à transférer un hôte combattu à un autre combattant encore libre. On libère ainsi combattants pour optimiser l'utilisation du budget.
- *L'ajout/la suppression de combat* (`moove_add_drop`) : cela permet de compléter le nombre de combats pour utiliser tout le budget d'énergie possible.

À chaque itération de l'algorithme, un tirage aléatoire entre les différents types de pas est effectué dans la fonction `get_neigbor`. Lors que le budget d'énergie est complètement utilisé, le tirage d'un add/drop supprimera systématiquement un combat.

Capitaine et Joker. Grâce aux structures de données utilisées, l'implémentation du Capitaine et du Joker ne requièrent pas beaucoup de modifications. Le Joker assigné à un combattant double les gains/pertes de tous ses combats. Il suffit donc de multiplier par 2 la ligne de M correspondant au combattant choisit. De même pour le capitaine, lors de la construction des matrices W et L, il suffit d'additionner 5 au niveau de compétence du capitaine choisi. Pour cela, nous choisirons un indice `captain_idx` à l'initialisation de la recherche et le Joker sera représenté par un masque ajouté à la matrice M lors de l'évaluation d'une solution.
Au lieu de fixer le Joker à l'initialisation comme effectué avec le Capitaine, un autre type de pas a été ajouté et permet de simplement changer l'indice du Joker dans la configuration (fonction `move_joker`).

Types de recherche et résultats. Pour approcher des solutions optimales, il a été choisi d'implémenter la recherche locale de type Hill-Climbing ainsi que le recuit simulé. Les résultats présentés en conclusion sont obtenus grâce à l'algorithme de recuit simulé. Celui-ci permet à l'algorithme de ne pas stagner dans les optimums locaux de l'espace de recherche. Pour cela, nous permettons à l'algorithme d'accepter de moins bonnes solutions selon une proportion suivant une loi exponentielle au cours du temps. La marche aléatoire tend donc à se stabiliser sur un optimum global.

3 Conclusion

Pour conclure, les deux approches de résolution du problème nous ont permis de l'aborder de deux manières complètement différentes. L'approche PLNE nous a forcés à optimiser sa formulation. Cet effort est récompensé par l'obtention de solutions exactes malgré un long temps de calcul. L'approche heuristique, quant à elle, nous a permis de formuler le problème sous forme matricielle plus intuitive permettant d'approcher très finement les solutions exactes des différentes instances plus rapidement.
Le tableau ci-dessous présente les résultats obtenus pour les différentes instances en fonction de la méthode utilisée :

Instance	**Méthode**	**Statut**	**Valeur**	**Temps (Δt)**
`tournament_6.txt`	PLNE	Optimal	2555	0.27s
	MH	Approché	2550	1.077
`tournament_20.txt`	PLNE	Optimal	8422	3.75s
	MH	Approché	8319	5.42s
`tournament_100.txt`	PLNE	Optimal	38192	438.79
	MH	Approché	25912	176.93

Table 1 – Comparaison des performances PLNE vs Méta-heuristique