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.

Table des matières

0.1 Introduction

1 Programmation en nombres entiers

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 possibles en essayant de trouver un maximum. Pour cela, nous devons définir plusieurs notions telles que le *voisinage* d'une solution, la 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.

2.1 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 que 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 :

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

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

$$\sum_{j=1}^{n}(C \times E). \tag{2}$$

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 :

$$P_{totale} = P \times \left(n - \sum_{j=1}^{n} \max_i C_{i,j} \right) \tag{3}$$

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

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

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.

2.2 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 le combat avec différents hôtes pour un même combattant.
- **Le transfert d'hôte** (`moove_shift_contestant`) : qui consiste à transférer un hôte combattu à un autre combattant encore libre qui permet de libérer des combattants.
- **L'ajout/la suppression de combat** (`moove_add_drop`) : cela permet de complémter 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 un combat.

2.3 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'évalusation d'une solution.

2.4 La recherche locale