Solution de Nanored4498

28 avril 2024

En essayant de résoudre le problème sur les différents jeux de données on se rend assez vite compte que les données ont été générées via une permutation des coureurs d'un certain nombre d'équipes qui initialement avait toutes le même temps. Dès lors, l'objectif n'est plus de réduire la différence de temps entre la pire et la meilleure équipe mais de trouver des équipes ayant toute le même temps. Ca peut paraître anodin mais cela change réellement la façon d'aborder le problème, entre autre car on suppose que le score parfait est atteignable.

Trois approches différentes m'auront permis de résoudre les instances du problème. Une approche pour le premier jeu de données, une autre pour le dernier (le cinquième) et une dernière pour les trois autre instances (2, 3 et 4).

1 Premier jeu de données

Pour cette instance, il n'y a que 8 coureurs. Le résultat demandé est une permutation des coureurs. Or il n'y a que 8! = 40320 permutations possibles. On peut alors tester toutes les permutations possible jusqu'à en trouver une qui produit deux équipes de même temps.

2 Cinquième jeu de données

Pour cette instance, les équipes sont grandes : 128 coureurs. Il y a 128! $\sim 4.10^{215}$ façon d'ordonner les coureurs dans une équipe. Si on couple ça avec le fait que tous les coureurs ont des temps relativement proches (entre 45s et 74.99s) on peut imaginer qu'il est possible d'obtenir à peu près n'importe quel score intéressant avec une équipe. En effet les score sont compris entre 5760 et 121 903.16 ce qui fait extrêmement peu de scores possibles comparés au nombre de permutations d'une équipe. De plus avec une analyse plus poussée sur $5 \cdot 10^6$ équipes générées aléatoirement de manière uniforme montre que la distribution du score d'une équipe aléatoire est une jolie Gaussienne d'espérance $\mu = 1.60 \cdot 10^4$ et d'écart type $\sigma = 1.15 \cdot 10^3$ (Figure 1). A préciser que lorsque que je parle d'équipe tirée aléatoirement il ne s'agit pas seulement d'un sous-ensemble de 128 coureurs mais d'une liste ordonnée de 128 coureurs qui est effectivement associée à un score.

Le graphique nous montre également que la probabilité qu'une équipe aléatoire obtienne un score choisi proche de μ est d'environ $p = 3.5 \cdot 10^{-6}$. Il faut donc en moyenne tirer aléatoirement de manière uniforme $N = 1/p = 2.9 \cdot 10^5$ équipes avant d'en trouver une avec le score que l'on cherche. Pour ma part j'ai choisi de générer des équipes avec le score 16000.82. Il suffit alors de former une équipe aléatoirement avec les coureurs restants jusqu'à obtenir le score souhaité et valider l'équipe. On répète cette recherche parmi les coureurs restants jusqu'à avoir former les 1024 équipes demandées. Il faudra de l'ordre du milliard de tirages uniforme ($1024 * N$) pour trouver une solution parfaite. Les équipes ayant une taille de 128, cela prendra quelques minutes (moins d'une dizaine).

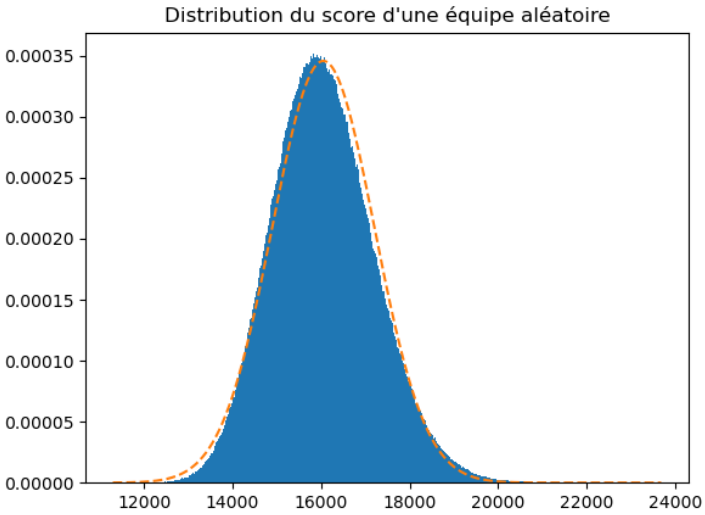

FIGURE 1 – Distribution du score d'une équipe tirée aléatoirement de manière uniforme sur le cinquième jeu de données.

3 Jeux de données 2, 3 et 4

Pour ces instances le nombre de coureurs par équipe est de 4. Or 4! = 24 est bien trop petit pour espérer pouvoir faire n'importe quel score avec 4 coureurs. D'autant plus qu'il faut diviser par 2 le nombre de permutations car inverser l'ordre ne change pas le score. 4 coureurs ne peuvent former qu'au plus 12 scores différents. On va donc former dans un premier des équipes avec un algorithme glouton puis via de la recherche locale on va mixer les équipes pour réduire la différence de score entre la pire et la meilleure équipe.

3.1 Algorithme glouton pour former des équipes

L'idée est de constamment avoir un pool d'équipes de 4 coureurs ordonnées et d'ajouter à notre solution l'équipe qui fait le moins augmenter la différence entre la pire et la meilleur équipe. Pour cela, je crée un premier pool de 10^8 équipes aléatoires. Je calcule m le score médian de ces équipes et j'ajoute à ma solution une équipe avec ce score médian. Ensuite, je divise le pool en deux files de priorité : une file des équipes avec un score plus grand que m et une file avec les équipes ayant un score plus petit que m. Les têtes de ces deux files sont les équipes avec les scores les plus proches de m. J'ajoute alors itérativement de manière gloutonne la tête de file qui a le score le plus proche de m. Si l'équipe que je veux ajouter contient un coureur déjà utilisé auparavant dans la solution je supprime simplement l'équipe de la queue et je génère une nouvelle équipe aléatoire avec les coureurs restants puis je l'ajoute à l'une des deux queues. Cette algorithme permet de générer une solution avec beaucoup d'équipes qui ont un score très proche de m (avec un écart d'au plus 0.1) et quelques équipes qui ont un score très loin de m (les dernières ajoutées).

3.2 Première recherche locale

Cette première recherche locale va permettre de réduire la différence entre la pire et la meilleure équipe à une valeur proche de 0.2. Désormais m ne référera plus à la médiane calculer initialement lors de la section précédente mais à la médiane de la solution courante. Étant données deux équipes A et B, notre recherche locale va tenter de mixer les deux équipes de sorte à minimiser $|\text{score}(A) - m| + |\text{score}(B) - m|$. L'idée est simplement de parcourir toutes les

permutations des coureurs des deux équipes et de retenir la permutation qui forme deux nouvelles équipes A' et B' minimisant $|\text{score}(A') - m| + |\text{score}(B') - m|$. Comme pour le premier jeux de données il n'y a que $8! = 40320$ permutations à tester. Toutefois il est possible de réduire encore plus le nombre de possibilité à seulement $\binom{8}{4} \cdot \frac{4!}{2} = 840$. Je tente alors d'appliquer cette recherche locale sur tous les couples (A, B) où A est soit l'équipe avec le plus ou celle avec le plus grand score.

3.3 Seconde recherche locale

Maintenant que la différence entre la pire et la meilleur équipe est fortement réduite, on va aller chercher une solution parfaire par force brute. Supposons que les score de nos équipes sont dans l'intervalle $I = [a, b]$. Je permutent aléatoirement l'ordre des équipes de ma solution, puis je les regroupe par paquets de $E = 32$ ou $E = 64$ équipes. Chaque paquet contient $C = E \cdot 4 = 128$ (ou $C = 256$ pour $E = 64$) coureurs. Puis pour chaque paquet, j'itère sur les $C!/(C-4)!$ 4-uplets de coureurs. Il y a de l'ordre du milliard de tels 4-uplets et je ne conserve que ceux qui sont associés à un score dans $I' = [a + 0.01, b]$ (ou $I' = [a, b - 0.01]$). On obtient alors un ensemble S d'équipes ordonnées de 4 coureurs ayant un score dans I'. Généralement S a une taille de l'ordre de 10^4. Il est possible de construire un graphe G dont les sommets sont les éléments de S et $(u, v) \in S^2$ est une arrête si et seulement si les coureurs des équipes u et v sont deux ensembles disjoints. On cherche alors à trouver une clique de taille E dans G. Une telle clique correspond à E équipes disjointes ayant des scores dans I'. Il se trouve qu'en codant correctement, la recherche de clique est très rapide (de l'ordre de la seconde). Si on est capable de trouver une telle clique pour tous les paquets de E équipes alors on obtient une nouvelle solution dont les équipes ont toute un score dans I' qui est strictement plus petit que I. Pour $b - a > 0.05$ cette procédure n'échoue quasiment jamais. Pour $b - a \leqslant 0.05$ il faudra relancer quelques fois la procédure jusqu'à trouver un regroupement par paquets de E équipes qui permettent de trouver les cliques recherchées. Pour $b - a \leqslant 0.02$ sur le troisième jeu de données j'ai du utiliser des paquets de $E = 96$ équipes.