L'extension à plusieurs joueurs : optimisations et compromis

HOSSAIN Akash
L3 MI

1 Optimisation pour le parcours

On suppose que la table a cinq cartes. Dans mon programme, on remplit la table avant de remplir les donnes inconnues. Soit n le nombre de joueurs dont on ne connait pas la donne. Soit $I = [\![1,\ n]\!]$, et π l'ensemble des cartes pouvant appartenir aux joueurs de donne inconnue (on exclue les cartes de la table et celles des donnes connues). Posons $<$ une relation d'ordre (strict) total sur π (qui est fini). Dans mon programme, $<$ est déterminée par la position des cartes dans la variable *pioche* (plus à gauche ou à droite). Je note aussi $\bigsqcup$ le symbole de réunion disjointe, et $P_k(E)$ l'ensemble des parties à k éléments d'un ensemble E.

On veut calculer la probabilité de victoire (stricte) de chaque joueur de donne connue en testant toutes les manières de remplir les n donnes inconnues. Mon univers sera $\Omega = \left\{(p_i)_i \in (P_2(\pi))^I | \bigsqcup_{i\in I} p_i \in P_{2n}(\pi)\right\}$. J'interprète un $(p_i)_{i\in I} \in \Omega$ comme l'issue "pour tout $i \in I$, la i-ème donne inconnue contient les cartes de p_i". Je munis Ω de la mesure de probabilité uniforme $\mathbb{P}$, ce qui fait bien de $(\Omega,\ \mathbb{P})$ un espace probabilisé fini. Soit J un joueur de donne connue. Dans le programme, J est le vainqueur du tournoi des donnes connues (pour une table remplie). Soit $E \subset \Omega$ l'évènement "J a une main strictement meilleure que toutes les mains de donnes inconnues". Quel est $\mathbb{P}(E)$?

Je vais partitionner E et Ω de manière intelligente pour gagner du temps de calcul. Pour $\sigma \in S_I$ une permutation, je note :

$$A_\sigma = \left\{(p_i)_{i\in I} \in \Omega | \forall i,\ j \in I\ i < j \Rightarrow max(p_{\sigma(i)}) < max(p_{\sigma(j)})\right\},$$

où le maximum d'un ensemble est calculé selon $<$ définie ci-dessus. On a alors deux résultats :

$$\begin{cases} \Omega = \bigsqcup_{\sigma\in S_I} A_\sigma \\ f_\sigma\ :\ (p_i)_{i\in I} \mapsto (p_{\sigma^{-1}(i)})_{i\in I} \text{ est une bijection } A_{id_I} \to A_\sigma \end{cases}$$

On a partitionné Ω en ensembles équipotents deux à deux. En particulier, $|A_{id_I}| = \dfrac{|\Omega|}{n!}$.

Soit $\sigma \in S_I$ et $(p_i)_i \in A_{Id_I} \cap E$. Alors pour tout $i \in I$, la main de J bat la donne p_i. Donc pour tout $i \in I$, elle bat aussi $p_{\sigma^{-1}(i)}$. Donc $f_\sigma(p_i)_i \in A_\sigma \cap E$.

Ainsi, $(A_\sigma \cap E)_{\sigma \in S_I}$ est une partition de E dont les éléments sont équipotents deux à deux. Donc, en particulier, $|A_{Id_I} \cap E| = \dfrac{|E|}{n!}$.

Au final, $\mathbb{P}(E) = \dfrac{|E|}{|\Omega|} = \dfrac{n!|A_{Id_I} \cap E|}{n!|A_{Id_I}|} = \dfrac{|A_{Id_I} \cap E|}{|A_{Id_I}|}$. Ainsi, on peut se contenter de parcourir A_{Id_I} pour calculer notre probabilité. Cela divise le temps total de calcul par $n!$, ce qui peut aller jusqu'à $9! = 362\ 880$. En pratique, cette optimisation se révèle insuffisante.

Le comportement du programme est exactement comme suit :

1. On a une pioche π, qui ne contient pas les cartes connues.
2. On remplit la table avec des éléments de π. On en a besoin d'au plus deux : $tc1$ et $tc2$. L'ordre de ces cartes n'a pas d'importance, donc après avoir choisi $tc1$, on choisit $tc2$ parmi les cartes de π qui sont $<$ à $tc1$.
3. On calcule J le vainqueur du tournoi des donnes connues pour cette table.
4. Pour i allant de 1 à n on construit p_i dont la carte la plus grande est plus grande que la plus grande carte de p_{i-1}, et dont la seconde carte est quelconque.
5. On teste si J est plus fort que tous les p_i. On incrémente nos compteurs en fonction du résultat.
6. On poursuit le parcours.

Pour être encore plus précis, je parcours A_σ ou $\sigma(i) = n - i + 1$, pour $c > c' \iff c$ est plus à gauche de c' dans π. Mais ces informations ont peu d'importance, tant qu'on a compris l'idée du programme.

2 Eviter les calculs superflus

Supposons que pour une table donnée, le tournoi des donnes connues n'admet aucun vainqueur strict. Alors il est inutile de remplir les donnes inconnues : on sait que les scores ne seront pas incrémentés. Cependant, on doit tout de même incrémenter notre compteur de cas parcourus (omega dans le programme), sinon les probabilités calculées à la fin seront erronées. La question est : de combien l'incrémenter ? Il faut dénombrer l'ensemble des cas parcourus, c'est à dire A_{Id_I}.

Soit m le nombre de joueurs de donnes connues. Comme la table est déjà remplie, et qu'un jeu de poker fait 52 cartes, on a d'abord $|\pi| = 47 - 2m$. On enlève les 5 cartes de la table, ainsi que les 2 cartes de chacune des m donnes. Ensuite, pour dénombrer Ω, on doit d'abord compter le nombre de manières de choisir les $2n$ cartes qui seront celles des donnes inconnues. Il y en a $\begin{pmatrix} |\pi| \\ 2n \end{pmatrix}$. Une fois qu'on a choisi nos $2n$ cartes, on doit les partitionner en n donnes de 2 cartes. Le coefficient multinomial pour calculer le nombre de tels choix est $\dfrac{(2n)!}{2^n}$. Pour finir, si on ajoute l'optimisation de la première section, on peut diviser notre nombre par $n!$. En multipliant tous ces nombres, on obtient au final $|A_{Id_I}| = \dfrac{|\pi|!}{(|\pi| - 2n)!n!2^n}$.

En pratique, j'incrémente vraiment omega à partir de cette formule, et j'obtiens des probabilités qui semblent correctes. Cela donne un argument "expérimental" supplémentaire pour justifier ma formule.

3 De la théorie à la pratique

J'ai codé la fonction nbCas qui donne le nombre de cas à parcourir en fonction du nombre de joueurs de donnes connues et inconnues. Dans la pire configuration (1 joueur connu, 9 inconnus), et avec l'optimisation de la section 1, ce nombre est de l'ordre de $6x10^{19}$ (pour une table pleine). En pratique, mon programme n'est donc pas utilisable quand il y a trop de cartes inconnues. Si on veut implémenter cette extension en temps d'exécution raisonnable, il aurait fallu passer par des méthodes plus astucieuses qu'un simple parcours.

3.1 Les différents cas de figure

On sait que dans le cas général, le programme est lent. On va avoir deux cas particuliers :

1. Le "meilleur cas" : dans toutes les manières de remplir la table, le tournoi des donnes connues n'admet aucun vainqueur strict (il y a ex aequo). A ce moment, on n'a jamais a remplir les donnes, mais juste a remplir la table, ce qui se fait en temps rapide. Ce cas est cependant inintéressant, car la probabilité pour un joueur connu de gagner est toujours 0...
2. Le "pire cas" : le contraire du meilleur cas, le tournoi des donnes connues admet un vainqueur strict dans quasiment toutes les manières de remplir la table. Cela arrive par exemple quand on a qu'un seul joueur connu, ou bien quand l'un des joueurs a une quinte flush royale dès le départ. A ce moment, on remplira les donnes à chaque fois, et c'est cette partie qui prends du temps dans le programme.

Quand on appellera la fonction, son temps d'exécution peut donc être vraiment très variable selon l'"écart type" des puissances des mains des joueurs connus. Pour 5 cartes inconnues, j'ai rencontré des cas ou l'exécution prenait 8 minutes (pire cas), 6 minutes, 45 secondes ou même moins d'une seconde (meilleur cas).

3.2 Flottant ou entier ?

En regardant les résultats de nbCas pour diverses situations, j'ai remarqué que ce nombre dépassait parfois la valeur maximum d'un entier. Sur un ordinateur de Sophie Germain, l'expression "let x = nbCas 9 1 in int_of_float x ; ;" s'évalue en 0. Donc, au moins pour le compteur de cas parcourus, on utilisera directement des flottants plutôt que des entiers. Comme le compteur est toujours incrémenté avec des valeurs très grandes, cela ne nuit pas trop à la précision. Devrait-on faire de même avec les scores ?

Si on utilise des entiers pour les scores, alors pour les "pires cas" décrits ci-dessus, on dépassera la taille d'un entier pour trop de cartes inconnues, ce

qui faussera complètement nos résultats. Si on utilise des flottants, alors comme les scores sont incrémentés par un "+ 1" à chaque fois, pour un nombre de cas parcourus trop grand, on peut atteindre une valeur x de score pour laquelle $x +. 1 = x$. A ce moment, le score ne sera plus jamais incrémenté, et les résultats seront aussi faussés. Sur le même ordinateur, l'expression "(nbCas 9 1) = (nbCas 9 1) +. 1.;;" s'évalue en true. Pour résoudre ce problème, j'ai une solution hybride. J'ai un tableau de scores entiers et un autre de flottants. J'ai une constante *upperBound* calculée par le programme qui approxime la valeur maximum atteignable par un entier. Quand un score entier atteint cette valeur, on incrémente le score flottant et on remet le score entier à 0. On arrive ainsi à contourner les défauts de chacune des deux représentations. Sur le même ordinateur, l'expression "(nbCas 9 1) +. (float_of_int upperBound) = nbCas 9 1;;" s'évalue en false.

3.3 Résultats concrets

Je donne le temps d'exécution en pire cas de la fonction en fonction du nombre de cartes inconnues. J'ai chronométré ces temps sur les ordinateurs des salles de TP de Sophie Germain.

1. 3 cartes inconnues ou moins : temps rapide.
2. 4 cartes : une vingtaine de secondes.
3. 5 cartes : un peu moins de 8 minutes au pire, mais avec l'optimisation de la section 2, il est raisonnable de s'attendre à moins.
4. Plus de 6 cartes : je n'ai pas essayé, et je le déconseille. Juste avec 6 cartes, on peut s'attendre à un temps 40 fois plus long qu'avec 5.