Résolution de jeux à deux joueurs et à information parfaite

Sujet proposé par François Pottier

24 mai 2016

Le **Tic-tac-toe** ou **morpion** $k/L/H$ se déroule sur une grille rectangulaire de largeur L et et de hauteur H. Chaque case est soit libre, soit occupée par un jeton, dont la couleur est blanche ou noire. Les deux joueurs sont eux-mêmes nommés Blanc et Noir. La grille est initialement vide. Blanc commence. Pendant la partie, chaque joueur à son tour place un jeton dans une case libre de son choix. Si l'un des joueurs parvient à aligner k jetons consécutifs, horizontalement, verticalement, ou en diagonale, alors il remporte la partie. Si la grille est entièrement remplie sans qu'aucun joueur n'ait remporté la partie, alors la partie est nulle.

Pour le Tic-tac-toe traditionnel, les paramètres valent $k = L = H = 3$. Pour le morpion traditionnel, k vaut 5, tandis que L et H sont laissés au choix des joueurs.

Le jeu de **Puissance 4**$/L/H$ est très proche du précédent. Le jeu se déroule également sur une grille rectangulaire de largeur L et de hauteur H. Traditionnellement, on a $L = 7$ et $H = 6$. En d'autres termes, la grille est constituée de L colonnes, et chaque colonne peut contenir jusqu'à H jetons. Pendant la partie, chaque joueur à son tour insère un jeton dans la colonne de son choix, pourvu qu'elle ne soit pas déjà pleine. Le jeton, sous l'influence de la gravité, vient occuper la case libre la plus basse dans cette colonne. Pour simplifier les choses, pour ce jeu, on fixe $k = 4$; on considère donc que le but est toujours d'aligner quatre jetons.

Ces jeux sont *à information parfaite :* chacun des joueurs dispose de la totalité de l'information à propos de l'état courant du jeu. De plus, ces jeux sont *finis :* le nombre de coups joués au cours d'une partie étant borné, il n'existe qu'un nombre fini (quoique très grand) de parties possibles.

Donc, si l'on fixe une position initiale (la grille vide, par exemple ; ou bien une grille déjà en partie remplie) et si l'on suppose les deux joueurs *parfaits* – c'est-à-dire que chacun joue toujours le meilleur coup possible – alors l'issue de la partie est en principe connue d'avance. Il suffit, pour la calculer, d'examiner tous les futurs possibles.

Par exemple, pour Tic-tac-toe 3/3/3 et pour une grille initialement vide, l'issue est : *match nul*. Pour Puissance 4/7/6 et pour une grille initialement vide, l'issue est : *Blanc gagne.* Ce dernier résultat a été démontré, à l'aide d'une analyse en partie manuelle et en partie mécanisée, dès 1988 [1]. Il faut noter que l'issue dépend des dimensions de la grille.

L'objectif de ce projet est de **vérifier ce type de résultat à l'aide d'une analyse entièrement mécanisée**. L'algorithme que nous utiliserons est en principe simple : il « suffit » d'examiner toutes les parties possibles, ou toutes les positions possibles. Il est clair, cependant, que ce nombre est beaucoup trop grand pour permettre une analyse exhaustive naïve ; en pratique, des optimisations profondes seront nécessaires pour rendre la chose possible.

Votre programme devra :

1. résoudre **Tic-tac-toe**, pour des valeurs variables des paramètres $k/L/H$;
2. résoudre **Puissance 4**$/L/H$, pour des valeurs variables des paramètres L/H ;
3. résoudre **un troisième jeu dont la nature sera révélée ultérieurement** – disons, en gros, à la moitié de la durée du projet.

Cette part d'inconnu doit en principe vous encourager à séparer dans votre code d'une part l'algorithme de recherche, indépendant du jeu considéré, d'autre part la définition du jeu.

Vous devrez concevoir un algorithme non seulement efficace, mais d'abord correct. Il serait évidemment **facile de calculer un résultat faux** très rapidement ; et une erreur est plus vite arrivée qu'on ne croit. Vous prendrez donc un soin particulier à vous convaincre, et à me convaincre, que votre algorithme est correct. Pour cela, le code devra rester aussi simple que possible. Enfin, vous devrez être conscients du fait que, par manque de temps, une simple lecture de votre code ne suffira probablement pas à me convaincre de sa correction. J'utiliserai donc un jeu de tests dont le format est défini plus loin (§2) et qui sera publié en ligne [3]. N'hésitez pas à effectuer vos propres tests : par exemple, construisez une position pour laquelle il est clair que Blanc perd après quelques coups, et vérifiez que la machine le confirme. Ou bien, construisez aléatoirement des positions relativement faciles à résoudre (des grilles déjà partiellement remplies) et vérifiez que votre algorithme optimisé produit le même résultat qu'un algorithme simple.

Pour les deux jeux mentionnés ci-dessus, le nombre de positions possibles est grossièrement borné par $3^{L \times H}$. On comprend donc que la difficulté du problème croît très rapidement avec L et H. Aussi, pour évaluer l'efficacité de votre programme, je n'accorderai pas trop d'importance à son temps d'exécution, dans l'absolu : qu'il résolve un problème $4/4/4$ en 5 secondes ou bien en 10 secondes ne change pas grand-chose. Je poserai plutôt la question : **jusqu'à quelles valeurs de $k/L/H$ votre programme est-il capable de déterminer l'issue du jeu en un temps raisonnable et à l'aide d'une quantité de mémoire raisonnable ?** Je fixerai comme limite, disons, une durée de moins d'une minute et une mémoire de 1 gigaoctets – tout en étant conscient que le temps d'exécution dépend de la machine utilisée.

Je jugerai votre code d'après les deux critères ci-dessus, c'est-à-dire **correction** et **efficacité**, le premier étant le plus important.

1 Conception du système

1.1 Algorithme de recherche

En ce qui concerne le code de recherche, je vous propose de commencer par implémenter un algorithme aussi simple que possible, puis, par raffinements successifs, d'en améliorer l'efficacité. À chaque étape, vérifiez que le programme produit des résultats corrects ; testez-le pour différentes dimensions de la grille, afin d'en évaluer l'efficacité ; faites afficher des statistiques (par exemple, le nombre de positions étudiées) afin de mieux en comprendre le comportement.

Je vous recommande fortement la lecture *détaillée* des transparents du cours de Jonathan Schaeffer [5], un expert mondial du sujet, en particulier les leçons 1 (introduction) et 3 à 5 (l'algorithme α-β ; arbres et DAGs ; approfondissement itératif et ordonnancement des coups). Un livre de cours comme celui de Russell et Norvig [4] pourra également être utile.

Voici le plan que je vous propose de suivre.

1.1.1 Minimax

Implémentez l'algorithme « Minimax ». Le Joueur, celui qui a la main, cherche le meilleur coup parmi tous ceux possibles, c'est-à-dire celui qui maximise son gain ; l'Opposant cherche la meilleure réponse, donc le coup qui minimise le gain de Joueur.

1.1.2 Negamax

L'écriture la plus simple de l'algorithme Minimax est redondante, car l'un des joueurs calcule un maximum, tandis que l'autre calcule un minimum. On obtient donc typiquement

deux fonctions mutuellement récursives identiques, modulo la dualité entre maximum et minimum.

Pour éliminer cette redondance, implémentez la formulation « Negamax », dans laquelle les deux joueurs sont identiques – tous deux calculent un maximum – mais travaillent avec des visions duales de l'échelle des valeurs – les valeurs vues par l'un sont les opposés des valeurs vues par l'autre.

On notera *LOSS* (« Joueur perd »), *DRAW* (« match nul ») et *WIN* (« Joueur gagne ») les trois valeurs possibles pour une position. On fera en sorte que les équations $DRAW = -DRAW$ et $LOSS = -WIN$ soient satisfaites.

1.1.3 α-β

Les algorithmes Minimax et Negamax sont très naïfs, car ils explorent l'intégralité de l'arbre des possibilités. Or, si on peut déterminer, par exemple, qu'un certain coup est gagnant, il est évidemment inutile d'explorer les autres.

Pour réduire le nombre de branches explorées, implémentez l'algorithme « α-β ». On peut décrire le principe de cet algorithme de la façon suivante. Lorsqu'on demande à l'algorithme de déterminer la valeur d'une position, on lui fournit également un intervalle ouvert de la forme $]\alpha, \beta[$, où $\alpha < \beta$. La spécification (ou « mission ») de l'algorithme est alors la suivante : si la valeur de la position se trouve dans l'intervalle $]\alpha, \beta[$, alors il doit renvoyer cette valeur exactement ; si la valeur se trouve au-dessous de cet intervalle, alors il peut en renvoyer une *approximation supérieure ;* enfin, si la valeur se trouve au-dessus de cet intervalle, alors il peut en produire une *approximation inférieure.* De ce fait, dès que l'algorithme a trouvé un coup qui garantit un résultat supérieur ou égal à β, il peut se passer d'évaluer les coups restants ; on parle alors de *coupure* (*cut-off*, en anglais).

L'algorithme est décrit dans un article de Knuth et Moore [2]. Prenez soin de bien comprendre son fonctionnement, et vérifiez la correction de votre implémentation.

1.1.4 Partage et mémoisation

L'algorithme α-β le plus simple explore un *arbre,* dont les positions forment les sommets et les coups forment les arêtes. C'est bien sûr une approximation, certes correcte, mais naïve : en réalité, c'est un *graphe* orienté acyclique (DAG) que l'on devrait explorer. En d'autres termes, plusieurs séquences de coups distinctes peuvent mener à la même position. Il est alors souhaitable de n'évaluer qu'une fois cette position, et de *mémoriser* le résultat de cette évaluation.

Ajoutez donc un mécanisme de *mémoisation* à votre implémentation de l'algorithme α-β. Vous utiliserez pour cela une table de hachage, parfois appelée « table de transpositions », parce qu'elle permet de détecter que deux séquences de coups distinctes mènent à une même position. En première approximation, cette table associe à chaque position déjà rencontrée une valeur.

Il est important de noter que l'algorithme peut être appelé à évaluer deux fois une même position pour des valeurs *différentes* de α et β. Par conséquent, lors du second appel, on ne peut pas toujours renvoyer aveuglément la valeur mémorisée lors de l'appel précédent. Cependant, cette valeur mémorisée permet en général de *réduire* l'intervalle $]\alpha, \beta[$ lors du second appel, ce qui peut alors dans certains cas provoquer une coupure.

Il faut également noter que, puisque l'algorithme produit parfois une approximation de la valeur réelle d'une position, la table de hachage ne pourra pas associer aux positions des valeurs absolues, mais des *intervalles* de valeurs.

Enfin, il est vital de contrôler la taille de la table de hachage, de façon à ce qu'elle ne dépasse pas la quantité de mémoire vive disponible. On pourra pour cela employer une stratégie simple, par exemple : n'enregistrer dans la table que les positions dont la profondeur est inférieure à une certaine constante.

1.1.5 Stratégies de remplacement

Parce que la mémoire de la machine est limitée, la table de hachage ne peut pas enregistrer toutes les positions rencontrées. Il faut donc décider quelles informations conserver et quelles informations jeter. J'ai suggéré plus haut de ne stocker dans la table que les positions proches de la racine, car ce sont les plus coûteuses à évaluer. On peut cependant imaginer de meilleures stratégies.

On peut imaginer, par exemple, de mesurer (à l'aide d'un simple compteur) le travail qu'il a fallu pour évaluer une certaine position ; d'enregistrer cette information dans la table ; et de conserver dans les tables les entrées qui ont demandé le plus de travail. Pour cela, on pourra autoriser une entrée peu coûteuse à être *remplacée* par une entrée plus coûteuse.

Une autre idée consiste à conserver dans la table les positions les plus récentes, car elles ont des chances d'être bientôt à nouveau rencontrées. Dans ce cas, on pourra autoriser une entrée plus ancienne à être remplacée par une entrée plus récente.

Ces deux idées étant complémentaires, on pourra se doter de *plusieurs* tables de hachage, chaque table étant dotée d'une stratégie de remplacement différente.

Par ailleurs, vous aurez probablement employé jusqu'ici une table de hachage basée sur la classe `HashMap<K,V>` de la bibliothèque standard de Java. Cette implémentation est correcte, mais quelque peu coûteuse, d'une part parce qu'elle exige que clefs et données soient allouées dans le tas, d'autre part parce qu'elle gère les collisions en maintenant des listes chaînées d'entrées.

Il peut maintenant être intéressant pour vous d'implémenter votre propre table de hachage. Vous pourrez spécialiser votre table pour le cas où les clefs sont des entiers longs – une position de Puissance 4 se code en 49 bits, §1.2 – et où les données sont des entiers ou entiers longs – 32 ou 64 bits suffisent probablement à coder les informations associées à chaque position. De plus, vous pourrez adopter une stratégie très simple de gestion des collisions : en cas de collision entre deux entrées, décidez laquelle des deux vous souhaitez conserver, et ne conservez que celle-ci ; n'utilisez pas de listes chaînées. Vous obtiendrez ainsi une implémentation particulièrement économique des tables de hachage, qui vous permettra de stocker un plus grand nombre d'entrées, tout en réduisant à zéro l'activité du GC.

Au passage, n'oubliez pas de vérifier que votre fonction de hachage est raisonnable, c'est-à-dire qu'elle ne provoque pas trop de collisions. Lorsque vous utilisez la classe `HashMap<K,V>`, vous pouvez compter les appels aux méthodes `hashCode` et `equals`, et vérifier que la première est appelée plus souvent que la seconde. Si ce n'est pas le cas, c'est que les collisions sont nombreuses.

1.1.6 Optimisations spécifiques à certains jeux

Il est parfois possible d'améliorer quelque peu le comportement de l'algorithme de recherche en utilisant des propriétés spécifiques du jeu considéré. Naturellement, lorsqu'on cherche à écrire l'algorithme de recherche sous une forme indépendante du jeu, ces propriétés sont parfois difficiles à exploiter. À vous de déterminer comment organiser le code pour que certaines de ces propriétés puissent être exploitées lorsqu'elles existent.

Les idées qui suivent concernent Puissance 4. Elles sont données à titre d'exemple. À vous de les adapter, si possible, à d'autres jeux. Vous pouvez n'en implémenter qu'une partie ; vous pouvez aussi en imaginer d'autres.

Symétrie La grille est symétrique vis-à-vis d'un axe vertical situé au centre de la grille. Vous pourrez modifier la façon dont les positions sont stockées dans la table de hachage, de façon à tenir compte de cette symétrie. Le nombre de positions évaluées sera ainsi potentiellement réduit de moitié.

Abstraction des jetons inutiles Lorsque la grille est en grande partie pleine, les jetons proches du bas sont souvent devenus *inutiles,* au sens où ils n'ont plus aucune chance de participer à un alignement de quatre jetons. Dans ce cas, leur couleur n'a plus d'importance : on pourrait les colorer en *gris* sans influencer la suite de la partie, donc la valeur de la position. Il semble donc intéressant de considérer comme *équivalentes* deux positions qui ne diffèrent que par la couleur de jetons inutiles. Vous pourrez modifier la façon dont les positions sont stockées dans la table de hachage, de façon à tenir compte de cette équivalence. Le nombre de positions évaluées sera ainsi réduit.

Calcul immédiat d'une borne supérieure Il est parfois facile de déterminer que Joueur ne peut plus gagner la partie. Si on remplit toutes les cases encore libres de jetons de la couleur de Joueur, et si on constate que, même dans ces conditions, Joueur n'a pas pu aligner 4 jetons, alors on peut en déduire que la valeur de la position courante est au plus *DRAW*. Cela peut éventuellement permettre une coupure immédiate.

Ordonnancement statique des coups Il semblerait qu'il soit souvent préférable de jouer au centre, ou près du centre, plutôt que sur les bords. En l'absence de toute information, on pourra donc préférer explorer d'abord les coups au centre, avant les coups sur les bords. On obtiendra ainsi plus rapidement une bonne approximation, ce qui permettra plus de coupures, donc un gain de temps. Cette technique est nommée « ordonnancement statique des coups ».

1.2 Représentation des positions en mémoire

Vous aurez besoin de deux représentations des positions en mémoire.

D'abord, pour les besoins de l'algorithme α-β, il vous faut une notion de *grille courante*, ou *position courante*. Cette structure de données doit être capable de répondre rapidement aux questions : Opposant a-t-il gagné ? Le match est-il nul ? Quels sont les coups valides pour Joueur ? Elle doit également être capable d'effectuer rapidement les opérations : jouer un coup ; défaire un coup. Pour des raisons d'efficacité, cette structure de données n'existera qu'en un seul exemplaire, et son contenu évoluera au cours du temps.

Ensuite, pour les besoins de la table de hachage, vous devrez représenter une position sous forme d'une séquence de bits aussi courte que possible.

Dans le cas de Puissance 4, voici une suggestion quant à la façon de représenter un ensemble de jetons de même couleur par une séquence de bits, tout en permettant de déterminer rapidement s'il existe un alignement de quatre jetons. Ajoutons une ligne vide au sommet de la grille, dont l'intérêt apparaîtra plus loin. Représentons chaque case de la grille par un bit, symbolisant l'absence ou la présence d'un jeton. Pour $L = 7$ et $H = 6$, par exemple, on propose d'organiser ces bits comme suit :

.	.	.	.	.	.	.
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Ainsi, en 49 bits, donc un entier long, on représente la position des jetons du Joueur ; de même, en un second entier long, on représente la position des jetons de l'Opposant. On obtient ainsi un codage particulièrement compact des positions. (On peut éventuellement faire mieux – je laisse cours à votre imagination !)

Le type des entiers de 64 bits est nommé `long` en Java et `int64` en OCaml. En OCaml, le type `int` correspond à des entiers de 31 ou 63 bits, suivant les machines ; vous pouvez faire

l'hypothèse que l'on est dans le second cas, i.e., que `Sys.word_size` vaut `64`.

Comment déterminer rapidement, à partir de la position des jetons du Joueur, si Joueur a réussi à aligner quatre jetons ? Notons que, si i est le numéro d'une certaine case, alors $i + 1$ est le numéro de la case située immédiatement au-dessus. Donc, si l'entier long `bitmap` représente l'ensemble des indices i tels que la case i contient un jeton Joueur, alors l'entier long `bitmap & (bitmap >> 1)` représente l'ensemble des indices i tels que les cases i et $i + 1$ contiennent *toutes deux* un jeton Joueur. (Il s'agit de la notation de Java. En OCaml, on écrit `bitmap land (bitmap lsr 1)`.) C'est ici que la ligne vide située au sommet de la grille est utile : elle rend le débordement hors d'une colonne, par le haut, inoffensif. À vous de vous en convaincre !

En généralisant cette idée, on détecte facilement les alignements verticaux de quatre jetons, puis les alignements horizontaux ou diagonaux de quatre jetons.

Signalons l'existence en Java de la fonction `Long.bitCount`, qui permet, une fois les alignements détectés, de les compter efficacement. Pour OCaml, il faut l'implémenter soi-même [6].

On peut utiliser une technique similaire pour déterminer relativement rapidement quels sont les jetons *inutiles* – qui n'ont plus aucune chance de participer à un alignement de quatre – ou encore quels sont les alignements viables de trois jetons, etc.

2 Cahier des charges

Votre programme devra respecter un cahier des charges précis. J'insiste sur ce point, car trop souvent, le cahier des charges que je propose n'est pas respecté. Le fait de le respecter non seulement facilite ma tâche lorsque j'évalue votre travail, mais démontre que vous êtes capable(s) de comprendre et de respecter une spécification imposée par un client. Cela fait donc partie de l'exercice.

Ce cahier des charges indique, en particulier, quels arguments vous devez accepter sur la *ligne de commande*, et quelles informations vous devez (ou pouvez) afficher sur la *sortie standard* et sur la *sortie d'erreur*.

Rappelons que, en Java, les arguments de la ligne de commande sont accessibles via le paramètre `args` de la méthode `main`. La sortie standard correspond au canal `System.out`, et la sortie d'erreur correspond au canal `System.err`. En OCaml, les arguments de la ligne de commande sont analysés à l'aide de la fonction `Arg.parse` de la bibliothèque standard. La sortie standard correspond au canal `stdout`, et la sortie d'erreur correspond au canal `stderr`.

Les instructions sont les suivantes.

Écrivez du code portable. Ne supposez pas que la machine utilise Windows, ou Linux, ou MacOS. Ne supposez pas l'existence de certains répertoires ou de certains fichiers.

N'employez pas de bibliothèques externes. Seules les bibliothèques standard de Java et OCaml peuvent être utilisées.

Soumettez un seul programme. Même si vous avez expérimenté avec différents algorithmes, différents réglages, différentes idées, choisissez-en un seul (correct ! et le plus efficace possible) et soumettez-moi celui-là. Les autres pourront être décrits dans le rapport.

Structurez votre archive comme suit. Soumettez une archive au format `.zip` ou `.tar.gz` telle que, une fois l'archive décompressée, on obtient un fichier `rapport.pdf` et un sous-répertoire `src` contenant **tous** les fichiers source Java ou OCaml. (Donc, ne créez pas de sous-répertoires dans `src`.) Nommez le fichier principal `Main.java` ou `Main.ml`. Je compilerai votre code comme suit :

```
cd src
javac *.java           # pour Java
ocamlbuild Main.native # pour OCaml
```

J'exécuterai ensuite votre programme comme suit :

```
cd src
java -ea -Xmx1024M Main <arguments> # pour Java
./Main.native <arguments>           # pour OCaml
```

Ici, les options `-ea` et `-Xmx1024M` sont destinées à la machine virtuelle Java : la première active l'instruction `assert` ; la seconde spécifie la taille du tas. Les arguments `<arguments>` vous sont destinés.

Parmi les arguments qui vous sont destinés, on trouvera uniquement **le nom du jeu**. Ce nom pourra être :

1. `tictactoe`
2. `connect4`
3. le nom du jeu-mystère, qui sera révélé ultérieurement.

Votre programme devra **lire sur l'entrée standard** une position (dont le format textuel est décrit ci-dessous), puis effectuer l'analyse de cette position, puis **afficher sur la sortie standard** l'un des trois mots suivants : `LOSS`, `DRAW`, ou `WIN`. Il n'affichera rien d'autre ! Si vous souhaitez afficher d'autres informations (nombre de positions visitées, etc.), vous pouvez le faire sur la sortie d'erreur.

On ne demande pas d'interface graphique. Vous pouvez bien sûr en réaliser une si vous le souhaitez, mais je n'en tiendrai pas compte dans l'évaluation de votre travail.

Le **format textuel** des positions, pour les jeux de Tic-tac-toe et Puissance 4, est le suivant.

On trouve en tête du fichier les paramètres du jeu. Ceux-ci sont de la forme $k/L/H$ pour Tic-tac-toe (par exemple, « `3/3/3` ») et de la forme L/H pour Puissance 4 (par exemple, « `7/6` »). **Vous pouvez supposer** $L \leq 7$ **et** $H \leq 6$**.**

Vient ensuite une série de $L \times H$ symboles représentant le contenu de la grille, de gauche à droite et de haut en bas. Le symbole « `.` » indique une case vide ; le symbole « `@` » indique un jeton Noir ; le symbole « `0` » indique un jeton Blanc. Blanc a la main. **Les caractères d'espacement et de retour à la ligne sont ignorés.**

Voici un exemple de description de grille, pour Puissance 4 :

```
7/6
.......
.......
.......
....@..
...000.
...@@@0
```

Dans cette configuration, Blanc doit gagner ; le résultat attendu sur la sortie standard est `WIN`.

3 Conseils généraux

Utilisez un système de contrôle de versions pour gérer l'évolution de votre travail et l'interaction entre vous si vous travaillez en binôme. `git`, par exemple, est gratuit et très bien conçu. Si vous n'avez jamais utilisé un tel outil, ce projet est une bonne occasion d'apprendre ; ce n'est pas très difficile !

Privilégiez la correction vis-à-vis de la performance. L'une des difficultés de ce sujet est qu'il fait apparaître une tension entre deux objectifs contradictoires. D'une part, la clarté et la modularité du code renforcent les chances que le code soit correct et facilitent son évolution et son application à plusieurs jeux différents. D'autre part, l'efficacité du code (en particulier, l'économie de mémoire) est importante. Cependant, l'efficacité ne peut être que secondaire vis-à-vis de la correction : en effet, l'efficacité d'un programme incorrect n'a aucun sens !

Insérez des assertions `assert (...)` dans votre code, et faites-les vérifier pendant l'exécution. (En Java, pour activer l'évaluation des assertions, il faut passer l'option `-ea` à la machine virtuelle `java`. En OCaml, elle est activée par défaut.). Le coût en sera minime, et vous détecterez plus facilement vos erreurs.

N'optimisez pas prématurément. Le succès ultime proviendra non pas de la vitesse brute de votre programme, mais de sa capacité à effectuer des coupures importantes. En d'autres termes, le succès proviendra non pas du grand nombre de positions évaluées par seconde, mais du faible nombre total de positions évaluées.

Ne cédez donc pas à la tentation de l'optimisation prématurée. Il est inutile d'obscurcir le code pour gagner un facteur 2, alors qu'il manque encore des idées pour gagner un facteur 2^{10}. Accordez la priorité à la clarté de votre code. Faites tout pour qu'il soit facile à modifier, de façon à pouvoir expérimenter rapidement de nouvelles idées.

Procédez scientifiquement. Lorsque vous apportez une soi-disant « amélioration » au programme, sachez dans quel but vous faites cela (par exemple pour gagner du temps, ou bien pour économiser de la mémoire vive, etc.) et faites des mesures pour déterminer si ce but a effectivement été atteint.

Si vous imaginez une nouvelle optimisation, qui promet plus de coupures, mais demande plus de calculs auxiliaires, procédez en deux temps. Commencez par introduire ces calculs auxiliaires, de façon à en mesurer le surcoût ; puis effectuez les coupures permises par ces informations nouvelles, et mesurez le gain alors obtenu. Vous évaluerez ainsi séparément les inconvénients et les avantages de votre idée.

Certaines optimisations semblent prometteuses sur le papier, et se révèlent peu efficaces ; parfois, c'est l'inverse. Prenez bonne note du gain apporté par chaque optimisation, et ne conservez que les optimisations les plus rentables.

Si besoin, contrôlez la taille du tas (`java -Xmx1024M`, par exemple) et le comportement du GC (`java -verbose:gc`). En OCaml, le module `Gc` de la bibliothèque standard permet de contrôler le comportement du GC.

Testez. Un jeu de tests sera publié sur la page de suivi du projet [3]. N'hésitez pas à effectuer vos propres tests.

Commencez tôt, car ce sujet est difficile. N'hésitez pas à me faire part de vos questions, de vos doutes, et de vos progrès par courrier électronique (`francois.pottier@inria.fr`).

Lors de l'exposé, essayez dans la mesure du possible de ne pas répéter ce qui est déjà contenu dans ce sujet. Expliquez plutôt vos contributions : quelles structures de données vous avez choisies, quelles heuristiques vous avez utilisées, quels ont été les gains de performance réalisés grâce à telle ou telle idée, de quelle manière élégante vous avez pu écrire le code, etc.

Références

[1] Victor Allis. A knowledge-based approach of connect-four. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, October 1988.

[2] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4) :293–326, 1975.

[3] François Pottier. Page de suivi.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[5] Jonathan Schaeffer. Heuristic search.

[6] Thibault Suzanne. Implementing hash array mapped tries for OCaml.