À la recherche d'une stratégie gagnante pour Puissance 4

Sujet proposé par François Pottier
mailto:Francois.Pottier@inria.fr

Difficulté : difficile (***)

24 janvier 2014

1 Présentation

Le jeu de Puissance 4 (*Connect 4*, en anglais) est simple. Le jeu se déroule sur une grille rectangulaire de largeur 7 et de hauteur 6. En d'autres termes, la grille est constituée de 7 colonnes, et chaque colonne peut contenir jusqu'à 6 jetons. Les deux joueurs, nommés Blanc et Noir, disposent chacun de 21 jetons correspondant à leur couleur. La grille est initialement vide. Blanc commence. 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. Si l'un des joueurs parvient à aligner *quatre* 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.

Puissance 4 est un jeu *à information parfaite :* chacun des joueurs dispose de la totalité de l'information à propos de l'état courant du jeu. De plus, il s'agit d'un jeu *fini :* le nombre de coups joués au cours d'une partie étant borné par 42, il n'existe qu'un nombre fini (quoique très grand) de parties possibles.

De ce fait, 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 une grille vide de taille 7×6, l'issue est : *Blanc gagne.* Ce résultat a été démontré, à l'aide d'une analyse partiellement manuelle et partiellement 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 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.

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.

2 Conception du système

2.1 Généralités

Plus les dimensions de la grille sont importantes, plus le nombre de positions est élevé, et plus le problème est difficile. Aussi, vous prendrez soin que votre code soit entièrement indépendant des dimensions de la grille. (Vous pourrez toutefois supposer que sa largeur est inférieure ou égale à 7, et sa hauteur inférieure ou égale à 6.)

Le nombre de positions possibles est grossièrement borné par $3^{H \times L}$, où H et L sont respectivement la hauteur et la largeur de la grille. On comprend donc que la difficulté du problème croît *très rapidement* avec H et L. Aussi, on n'accordera pas trop d'importance dans l'absolu au temps d'exécution du programme. On se posera surtout la question : **jusqu'à quelles valeurs de H et L le programme est-il capable de déterminer l'issue du jeu ?** Idéalement, vous atteindrez $H = 6$ et $L = 7$, mais un résultat inférieur peut être honorable.

Vous prendrez soin de construire un algorithme non seulement efficace, mais surtout 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 des jeux de tests (§4) : pour une série de positions de départ construites manuellement ou aléatoirement, je vérifierai que votre programme produit un résultat correct. 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.

Vous essaierez, dans la mesure du possible, de séparer le code de recherche, qui est indépendant du jeu choisi (tic-tac-toe, Puissance 4, échecs, dames, etc.), du code spécifique de Puissance 4 (définition de la grille, des coups valides, des positions gagnantes, des positions nulles, etc.).

2.2 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.

Voici le plan que je vous propose de suivre.

2.2.1 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. Vous utiliserez une profondeur de recherche égale à LH, de façon à ce que les joueurs disposent d'une information parfaite.

2.2.2 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 pourra noter *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.

2.2.3 α-β

Les algorithme 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.

2.2.4 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 hash, parfois appelée « table de transpositions », parce qu'elle permet de détecter que deux séquences de coups distinctes mènent à une même position. Cette table associera aux positions déjà rencontrées des valeurs.

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 hash 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 hash, 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.

2.2.5 Approfondissement itératif

L'écriture la plus simple des algorithmes précédents utilise une formulation récursive, et effectue par conséquent une exploration en profondeur d'abord, jusqu'à la profondeur maximale, à savoir 42, pour une grille de dimensions 7×6. Or, cela n'est pas une bonne chose : si par exemple, à partir d'une position donnée, deux coups sont permis, il serait dommage de commencer par explorer les conséquences du premier à la profondeur 42, pour ensuite découvrir que le second est gagnant à la profondeur 2.

Pour remédier à ce problème, implémentez un « approfondissement itératif ». À partir de la position initiale, explorez d'abord à profondeur 2, puis 4, etc. jusqu'à atteindre enfin la profondeur maximale, par exemple 42.

Vous vérifierez que cette technique d'approfondissement itératif permet à la machine de reconnaître instantanément une position qu'un humain considérerait comme « gagnante en deux coups », ou « perdante en deux coups », par exemple ; ce qui n'était probablement pas le cas auparavant.

Le fait d'effectuer une exploration à profondeur limitée introduit un aspect nouveau : *l'information imparfaite.* En effet, lorsque la profondeur limite est atteinte, l'algorithme doit renvoyer immédiatement un résultat, alors que la valeur réelle de la position est inconnue. On utilise pour cela une « fonction d'évaluation », chargée d'attribuer une valeur heuristique à une position, sans en explorer les descendants. Je vous propose ici de choisir, pour commencer, une fonction d'évaluation triviale, qui renvoie toujours la valeur *DRAW*.

Vous prendrez garde à la question suivante : peut-on ré-utiliser, pendant une nouvelle itération, des valeurs ou intervalles de valeurs calculés pendant une itération précédente ? En général, la réponse est non ! Par exemple, considérons une position gagnante en quatre coups. Si on l'évalue avec pour limite de profondeur 2, on obtient la valeur *DRAW*. Il faut attendre l'itération suivante, avec pour limite de profondeur 4, pour découvrir que la valeur correcte est *WIN*.

De ce fait, les intervalles de valeurs stockés dans la table de hash devront être accompagnés du numéro de l'itération pendant laquelle ils ont été calculés. A priori, on ne pourra ré-utiliser une information issue de la table de hash que si elle a été calculée pendant l'itération courante.

Heureusement, on peut être un peu moins pessimiste que ne le suggèrent les paragraphes précédents. Notre fonction d'évaluation triviale, qui renvoie toujours *DRAW*, vérifie trivialement la propriété de *prudence* suivante : elle ne renvoie la valeur *LOSS* (resp. *WIN*) que lorsque la défaite (resp. la victoire) est certaine. Il en découle que, si on a pu obtenir avec certitude la valeur *LOSS* ou la valeur *WIN* lors d'une itération donnée, cette valeur restera correcte lors des itérations suivantes.

2.2.6 Stratégies de remplacement

Parce que la mémoire de la machine est limitée, la table de hash 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 hash, chaque table étant dotée d'une stratégie de remplacement différente.

Par ailleurs, vous aurez probablement employé jusqu'ici une table de hash basée sur la classe `HashMap<K,V>` de la librairie 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 hash. Vous pourrez spécialiser votre table pour le cas où les clefs sont des entiers longs – une position se code en 49 bits, §2.3 – 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 hash, 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 hash 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.

2.2.7 Optimisations spécifiques à Puissance 4

Il est possible d'améliorer quelque peu le comportement de l'algorithme de recherche en utilisant notre connaissance du jeu, ici Puissance 4. Voici quelques suggestions ; d'autres idées peuvent être utilisées.

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

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. Cette technique est nommée « ordonnancement statique des coups ».

2.2.8 Fonction d'évaluation et ordonnancement dynamique des coups

Vous pouvez tenter de définir une fonction d'évaluation plus subtile que celle suggérée plus haut, qui renvoie toujours *DRAW*. Une telle fonction utiliserait des valeurs intermédiaires autres que *LOSS*, *DRAW* et *WIN*, et attribuerait de meilleures notes aux positions qui semblent plus prometteuses.

Vous prendrez garde de préserver la propriété de « prudence » définie plus haut. Par ailleurs, si vous avez exploité une notion d'équivalence entre positions, comme cela est suggéré plus haut (symétrie ; abstraction des jetons inutiles), alors la fonction d'évaluation devra être compatible avec cette relation d'équivalence.

Quel est l'intérêt d'employer une fonction d'évaluation non triviale ? L'objectif est de *guider* l'algorithme d'approfondissement itératif. Lors de l'itération $n + 1$, on peut choisir d'explorer d'abord le ou les coups qui, lors de l'itération précédente, ont semblé les plus prometteurs. Cette technique est nommée « ordonnancement dynamique des coups ». Dans une situation idéale, le premier coup exploré sera le bon, et on se passera totalement d'explorer les autres.

Pour permettre l'ordonnancement dynamique des coups, vous utiliserez la table de hash pour associer, à chaque position, le coup qui dans cette position a semblé jusqu'ici le plus prometteur.

2.3 Représentation des grilles 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 hash, vous devrez représenter une position sous forme d'une séquence de bits aussi courte que possible.

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 !)

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. 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 de la fonction `Long.bitCount`, qui permet, une fois les alignements détectés, de les compter efficacement.

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.

3 Évaluation

Je jugerai votre code d'abord d'après deux critères, le premier étant le plus important :

1. *Correction :* le programme attribue-t-il à toute position initiale une valeur correcte dans l'ensemble $\{LOSS, DRAW, WIN\}$?
2. *Efficacité :* quelles sont les plus grandes dimensions L et H pour lesquelles le programme est capable, en un temps « raisonnable » et à l'aide d'une quantité de mémoire « raisonnable », de déterminer la valeur de la position initiale ?

Je vérifierai la *correction* de votre programme à l'aide d'une batterie de positions test dont le format textuel est défini plus loin (§4). Votre programme devra lire sur la ligne de commande le nom d'un fichier contenant une position, effectuer l'analyse de cette position, puis afficher sur la sortie standard (`System.out`) 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 (`System.err`). Le jeu de positions test concernera des grilles de différentes tailles, et contiendra des positions plus ou moins faciles à analyser. (En règle générale, plus la grille est grande et plus la position considérée contient de cases libres, plus cette position est coûteuse à analyser.) Ce jeu de tests est en partie publié sur la page de suivi du projet [3].

Prenez bien soin de vérifier la correction de votre code, car s'il est faux, son efficacité n'aura aucun sens !

Je vérifierai *l'efficacité* de votre programme de deux façons.

D'une part, j'utiliserai le jeu de positions test : pour des positions de difficulté variable, je mesurerai le temps de réponse de votre programme. Je regarderai jusqu'à quelle difficulté votre programme parvient à répondre très rapidement ; disons, en quelques secondes au maximum.

D'autre part, comme indiqué plus haut, je déterminerai quelles sont les plus grandes dimensions L et H pour lesquelles le programme est capable, en un certain temps et à l'aide d'une quantité de mémoire vive, de déterminer la valeur de la position initiale. On peut fixer comme

limite, disons, une durée de quelques minutes et une mémoire de 1 gigaoctets.

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.

4 Format textuel des grilles

Le format textuel des grilles du jeu de test est simple. On trouve en tête du fichier les dimensions de la grille, de la forme $L \times H$, par exemple `7x6`. Vient ensuite une série de LH symboles représentant le contenu de la grille. Le symbole « `.` » indique une case vide ; le symbole « `@` » indique un jeton Noir ; le symbole « `O` » indique un jeton Blanc. Blanc a la main. Les caractères d'espacement et de retour à la ligne sont ignorés. Les commentaires, ouverts par le symbole « `#` » et qui s'étendent jusqu'à la fin de la ligne, sont ignorés.

Voici un exemple de fichier de description de grille :

```
7x6
# White wins.
.......
.......
.......
....@..
...OOO.
...@@@O
```

5 Questions subsidiaires

La stratégie gagnante que vous avez trouvée est un graphe. Sauriez-vous compter le nombre de ses sommets, ou en donner une approximation supérieure ?

Sauriez-vous utiliser cette stratégie gagnante pour écrire un joueur parfait *interactif,* c'est-à-dire capable de répondre instantanément (disons en une seconde au maximum) à chaque coup de l'adversaire ? Ce joueur exigera de commencer toujours, et gagnera toujours. Une personnalité assez désagréable, somme toute !

Je rappelle qu'on ne demande pas d'interface graphique. Le joueur interactif pourra être doté d'une interface textuelle.

6 Conseils généraux

L'une des difficultés de ce sujet est qu'il fait apparaître une tension entre deux objectifs contradictoires : d'une part la clarté et l'élégance du code, qui renforcent les chances que le code soit correct et facilitent son évolution ; d'autre part, l'efficacité du code, en particulier l'économie de mémoire, qui est nécessaire pour réussir.

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^{20}.

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.

Lorsque 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.

N'oubliez pas d'utiliser un système de contrôle de versions pour gérer l'évolution de votre travail et l'interaction entre vous si vous travaillez en binôme. `svn` et `git` sont gratuits et très répandus.

Insérez des assertions dans votre code, et faites-les vérifier pendant l'exécution (`java -ea`). Le coût en sera minime, et vous détecterez plus facilement vos erreurs.

Si besoin, contrôlez la taille du tas (`java -Xmx1024M`, par exemple) et le comportement du GC (`java -verbose:gc`).

Ce sujet est difficile. **Commencez tôt**, et 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 trop 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*. 2009.

[5] Jonathan Schaeffer. Heuristic search.