Turtle, un programme d'Othello

Fabien Letouzey[1]

juin 1997

[1]Maîtrise Informatique, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'ascq

Table des matières

Résumé

La programmation des jeux de réflexion est un domaine de l'intelligence artificielle qui a suscité beaucoup de recherches à l'époque des balbutiements en informatique. L'objectif était de prouver que les machines étaient capables de penser et d'aider à mieux comprendre le raisonnement humain.

De nos jours les programmes développés dans ce domaine qui donnent les meilleurs résultats sont plutôt basés sur la *force brute*, c'est-à-dire l'analyse d'un très grand nombre de positions, cette analyse étant alors nécessairement simplifiée. Ces programmes contribuent maintenant d'avantage au développement d'algorithmes de recherche arborescente (et d'heuristiques associées) et de techniques liées au parallélisme. Ils sont également un terrain d'essai important pour les méthodes d'apprentissage dirigé et automatique.

Le jeu d'Othello est plus « simple » que les échecs (le nombre moyen de coups légaux dans une position est inférieur et une partie dure toujours 60 coups notamment). Ceci permet aux programmeurs d'utiliser des techniques d'apprentissage qui n'ont pas encore donné de résultats probants aux échecs par exemple.

TURTLE se situe dans ce cadre grâce à son apprentissage (dirigé) de la fonction d'évaluation à partir de positions d'exemple — dont le résultat est connu —, et à son apprentissage (automatique) des ouvertures en exploitant son expérience (les parties jouées précédemment).

Chapitre 1

L'algorithme de recherche

1.1 Introduction

Les meilleurs programmes actuels de jeux à deux joueurs, à information complète et à somme nulle (les échecs, les dames, Othello, le go...) utilisent une fonction d'évaluation pour choisir un coup dans une position donnée. Cette fonction, notée ici f fait correspondre à toute position P une valeur numérique estimant les chances de gain du joueur ayant le trait dans P (c'est-à-dire que $f(P)$ est d'autant plus grand que ce joueur a de bonnes chances de l'emporter, 0 représentant une position égale). On ne connaît pas d'algorithme estimant parfaitement le résultat théorique de toute position dans un temps raisonnable. On procède alors de la façon suivante :

- on choisit une fonction f simple estimant rapidement les chances du joueur ayant le trait
- on utilise un algorithme de recherche de type Min-Max qui applique la fonction f à une profondeur p donnée de la position P (à partir de P, p coups sont joués avant de juger les chances de gain à l'aide de f)

On constate dans la pratique que $MinMax(P, p)$ estime d'autant mieux les chances de gain du joueur ayant le trait que p est grand (en particulier, $MinMax(P, 0) = f(P)$ et $MinMax(P, \infty)$ est le résultat théorique de P).

Dans Turtle, la fonction Min-Max est implémentée à l'aide de l'algorithme Nega-Scout. Cet algorithme étant complexe, le Min-Max et l'Alpha-Beta sont tout d'abord présentés.

1.2 Le Min-Max

Le nom de *Min-Max* provient du fait que les jeux concernés étant à deux joueurs jouant alternativement et à somme nulle, le joueur au trait essaie de *minimiser* les chances de gain de son adversaire qui lui, au coup suivant, tentera des les *maximiser*. Pour implémenter le Min-Max, il est plus aisé de prendre le parti du joueur ayant le trait à chaque position au cours de la recherche, qui est la convention *Nega-Max* : chaque joueur maximise sa propre note qui est l'opposée de celle de son adversaire.

La valeur $MinMax(P, p)$ est alors :

- $f(P)$ si P est terminale ou si $p = 0$
- $\max_i(-MinMax(P_i, p-1))$ sinon, P_i étant une position fille de P

Un exemple d'algorithme Min-Max implémenté en Nega-Max est celui de la figure 1.1, présenté en pseudo-code C. `INFINI` $= \infty$ est une valeur numérique finie telle que $\forall P, -\infty \leq f(P) \leq +\infty$.

```
int MinMax(position P, int Prof) {

   int n, meilleur, i, valeur;

   if (Prof == 0) return f(P);

   calculer les positions filles de P : P1, ..., Pn;
   if (n == 0) return f(P);

   meilleur = -INFINI;

   for (i = 1; i <= n; i++) {
      valeur = -MinMax(Pi,Prof-1);
      if (valeur > meilleur) meilleur = valeur;
   }

   return meilleur;
}
```

FIG. 1.1: *L'algorithme Min-Max*

1.3 L'Alpha-Beta

On définit une fonction *AlphaBeta*, la valeur $AlphaBeta(P, p, \alpha, \beta)$ étant :

- $f(P)$ si P est terminale ou si $p = 0$
- $\min(\beta, \max_i(-AlphaBeta(P_i, p-1, -\beta, -\alpha)))$ sinon

Une propriété importante de la fonction *AlphaBeta* est la suivante :

Propriété 1 *si $v = AlphaBeta(P, p, \alpha, \beta)$, $w = MinMax(P, p)$ et $\alpha < \beta$, on a :*

- $v \leq \alpha \Rightarrow w \leq v$
- $\alpha < v < \beta \Rightarrow w = v$
- $v \geq \beta \Rightarrow w \geq v$

Démonstration 1 *(par récurrence sur la profondeur p)*

- $p = 0 \Rightarrow w = f(P) = v$
- *si la propriété est vraie au rang* $p-1$,
  - P *terminale* $\Rightarrow w = f(P) = v$
  - $v \leq \alpha$
    $\Rightarrow \forall P_i, -AlphaBeta(P_i, p-1, -\beta, -\alpha) \leq v$
    $\Rightarrow \forall P_i, AlphaBeta(P_i, p-1, -\beta, -\alpha) \geq -v$
    $\Rightarrow \forall P_i, MinMax(P_i, p) \geq -v$
    $\Rightarrow \forall P_i, -MinMax(P_i, p) \leq v$
    $\Rightarrow w \leq v$
  - $\alpha < v < \beta$
    $\Rightarrow \forall P_i, -AlphaBeta(P_i, p-1, -\beta, -\alpha) \leq v \text{ et } \exists P_i / -AlphaBeta(P_i, p-1, -\beta, -\alpha) = v$
    $\Rightarrow \forall P_i, AlphaBeta(P_i, p-1, -\beta, -\alpha) \geq v \text{ et } \exists P_i / AlphaBeta(P_i, p-1, -\beta, -\alpha) = -v$
    $\Rightarrow \forall P_i, MinMax(P_i, p) \geq v \text{ et } \exists P_i / MinMax(P_i, p) = -v$
    $\Rightarrow \forall P_i, -MinMax(P_i, p) \leq v \text{ et } \exists P_i / -MinMax(P_i, p) = v$
    $\Rightarrow w = v$
  - $v \geq \beta$
    $\Rightarrow \forall P_i, -AlphaBeta(P_i, p-1, -\beta, -\alpha) \geq v$
    $\Rightarrow \forall P_i, AlphaBeta(P_i, p-1, -\beta, -\alpha) \leq -v$
    $\Rightarrow \forall P_i, MinMax(P_i, p) \leq -v$
    $\Rightarrow \forall P_i, -MinMax(P_i, p) \geq v$
    $\Rightarrow w \geq v$

On en déduit la propriété suivante :

Propriété 2 $AlphaBeta(P, p, -\infty, +\infty) = MinMax(P, p)$

Démonstration 2 *On a* $-\infty \leq MinMax(P, p) \leq +\infty$, *d'où :*

- $AlphaBeta(P, p, -\infty, +\infty) = -\infty$
  $\Rightarrow MinMax(P, p) \leq AlphaBeta(P, p, -\infty, +\infty) = -\infty$
  $\Rightarrow MinMax(P, p) = -\infty = AlphaBeta(P, p, -\infty, +\infty)$
- $-\infty < AlphaBeta(P, p, -\infty, +\infty) < +\infty$
  $\Rightarrow MinMax(P, p) = AlphaBeta(P, p, -\infty, +\infty)$
- $AlphaBeta(P, p, -\infty, +\infty) = +\infty$
  $\Rightarrow MinMax(P, p) \geq AlphaBeta(P, p, -\infty, +\infty) = +\infty$
  $\Rightarrow MinMax(P, p) = +\infty = AlphaBeta(P, p, -\infty, +\infty)$

Un exemple d'implémentation de l'Alpha-Beta est donné figure 1.2. Il exploite l'intérêt de cet algorithme, qui est qu'à un noeud donné, on peut arrêter la recherche sur ce noeud dès que l'on a trouvé une position fille P_i telle que $AlphaBeta(P_i, p-1, -\beta, -\alpha) \geq \beta$. On dit alors qu'il y a une *coupure Alpha-Beta*. Il n'est donc pas nécessaire d'étudier toutes les positions issue de P à la profondeur p pour calculer $MinMax(P, p)$!

```
int AlphaBeta(position P, int Prof, int Alpha, int Beta) {

   int n, meilleur, i, valeur;

   if (Prof == 0) return f(P);

   calculer les positions filles de P : P1, ..., Pn;
   if (n == 0) return f(P);

   meilleur = -INFINI;

   for (i = 1; i <= n; i++) {
      valeur = -AlphaBeta(Pi,Prof-1,-Beta,-Alpha);
      if (valeur > meilleur) {
         meilleur = valeur;
         if (meilleur > Alpha) {
            Alpha = meilleur;
            if (Alpha >= Beta) return Alpha;
         }
      }
   }

   return meilleur;
}
```

FIG. 1.2: *L'algorithme Alpha-Beta*

1.4 Le Nega-Scout

L'algorithme Alpha-Beta est d'autant plus efficace lorsqu'à chaque noeud de l'arbre de recherche, les coups (correspondants aux positions P_i) sont *triés* dans l'ordre du meilleur potentiel au plus mauvais, c'est-à-dire quand des notes élevées sont trouvées tôt dans ce noeud. L'algorithme Nega-Scout suppose que les coups sont bien triés et exploite directement cette condition au niveau de l'algorithme. A chaque noeud, le calcul de la note du premier coup est effectué de la même manière que pour l'Alpha-Beta mais ensuite une comparaison est effectuée entre cette note et celle des coups suivants (à l'aide d'une fenêtre minimale $[\alpha, \alpha + 1]$). Si la nouvelle valeur est inférieure ou égale à l'ancienne, elle est ignorée. Dans le cas contraire, une nouvelle recherche avec une fenêtre $[valeur, \beta]$ est nécessaire. Il y a dans ce cas une perte de temps due à une double recherche sur la même position P_i mais ce cas est rare étant donné le tri des coups.

```
int NegaScout(position P, int Prof, int Alpha, int Beta) {

   int n, meilleur, i, valeur;

   if (Prof == 0) return f(P);

   calculer les positions filles de P : P1, ..., Pn;
   if (n == 0) return f(P);

   meilleur = -NegaScout(P1,Prof-1,-Beta,-Alpha);

   for (i = 2; i <= n; i++) {
      if (meilleur > Alpha) {
         Alpha = meilleur;
         if (Alpha >= Beta) return Alpha;
      }
      valeur = -NegaScout(Pi,Prof-1,-Alpha-1,-Alpha);
      if (valeur > Alpha && valeur < Beta)
         valeur = -NegaScout(Pi,Prof-1,-Beta,-valeur);
      if (valeur > meilleur) meilleur = valeur;
   }

   return meilleur;
}
```

FIG. 1.3: *L'algorithme Nega-Scout*

Chapitre 2

Les heuristiques de recherche

2.1 Introduction

L'algorithme de recherche Alpha-Beta et ses dérivés sont des algorithmes *directionnels*, ce qui signifie que leur efficacité en terme de temps dépend de l'ordre dans lequel les coups sont évalués à chaque noeud. Plus précisément, le cas optimal est atteint si à chaque position rencontrée au cours de la recherche le meilleur coup est étudié en premier. Pour tous les jeux non triviaux cependant (et c'est pour l'instant le cas d'Othello), on ne connaît pas de moyen de déterminer ce coup autrement que par une recherche de type Min-Max. On se contente alors d'utiliser des heuristiques qui certes ne garantissent pas d'étudier le meilleur coup en premier, mais favorisent la probabilité de se trouver dans ce cas.

2.2 L'approfondissement itératif

L'approfondissement itératif consiste à étudier une position donnée à l'aide de l'Alpha-Beta à la profondeur 1, puis de trier les coups racine (c'est-à-dire les coups légaux dans la position de départ de la recherche) par note décroissantes (les meilleurs, donc, en premier) puis de recommencer à la profondeur 2, puis 3, etc..., jusqu'à ce qu'une limite de temps soit dépassée. L'approfondissement itératif a deux objectifs distincts :

- il permet d'étudier une position pendant une durée déterminée tout en proposant à chaque instant un coup décent : lorsque la limite de temps est atteinte, il suffit de jouer le meilleur coup trouvé à ce moment (ce choix du meilleur coup étant remis en question à chaque approfondissement)
- les coups racine sont étudiés dans un ordre tel que le meilleur coup à une profondeur donnée a de grandes chances de se trouver en premier, ceci favorise donc les élagages Alpha-Beta

Il est à noter que lorsque d'autres heuristiques de recherche sont utilisées conjointement à l'approfondissement itératif, la recherche à la profondeur 1, puis 2, ..., puis n est plus rapide que la recherche directe à la profondeur n ! La raison en est que ces autres heuristiques utilisent des résultats acquis lors de la recherche à la profondeur $p-1$ pour une recherche à la profondeur p.

2.3 La table de transposition

L'idée générale de la table de transposition est double :

1. conserver en mémoire les résultats acquis au cours de la recherche
2. reconnaître les positions identiques obtenues par des suites de coups distinctes (les transpositions)

Le premier point semble pouvoir être facilement implémenté en stockant dans un arbre matérialisant la recherche les résultats obtenus. Malheureusement le nombre de positions étudiées au cours d'une recherche de temps moyen en conditions de tournoi dépasse largement la capacité mémoire des stations de travail actuelles ; il faut donc gérer cette mémoire. De plus on aimerait bien conserver les informations utiles pour les coups suivants de la partie. L'implémentation par arbre ne permet pas de conserver ces informations facilement. On lui préfère l'utilisation d'une « table de hachage » de taille fixe (on ajuste cette taille en fonction de la machine utilisée). On fait alors correspondre à chaque position possible d'Othello une « clé de hachage » (une valeur numérique) de laquelle on peut déduire un index compris entre 0 et $n-1$ inclus pour une table de hachage à n entrées. A toute position peut alors être associée une entrée dans la table.

Ce système résout les deux problèmes rencontrés précédemment :

- deux positions identiques mais atteintes par deux suites de coups différentes (interversion de coups) auront la même clé de hachage, et seront donc associées avec la même entrée de la table
- la limitation en mémoire est gérée par la clé de hachage dont l'ensemble des valeurs possibles est beaucoup plus petit que l'ensemble des positions que l'on peut rencontrer à Othello ; deux positions distinctes peuvent donc être associées à une même entrée dans la table (cette situation est un conflit), on applique alors une règle de choix désignant laquelle de ces deux positions on souhaite conserver

2.3.1 La clé de hachage

Dans Turtle, la taille de la table de transposition (en nombre d'entrées) est 2^n, n pouvant être choisi au lancement du programme. La clé de hachage d'une position est un nombre codé en binaire sur 64 bits, dont les n bits de poids faible serviront d'index.

Au lancement du programme, à chaque case de l'othellier sont associées deux valeurs pseudo-aléatoires de 64 bits, l'une correspondra à un pion noir posé sur cette case, l'autre à un pion blanc. Lorsqu'un calcul de clé de hachage est nécessaire, un parcours de chaque case de la position est effectué et un `xor` (ou exclusif) des valeurs pour chaque case est exécuté. On prend pour chaque case la première valeur aléatoire qui s'y trouve si elle contient un pion noir, la deuxième si elle contient un pion noir, et 0 (l'élément neutre du `xor`) si elle est vide. On considère le trait comme une case virtuelle supplémentaire contenant un pion de la couleur du joueur qui a la main. On obtient donc une valeur 64 bits pour chaque position, les n bits de poids faible servent à indexer la table de transposition et indiquent donc l'entrée correspondant à la position et les 32 bits de poids fort servent de « verrou » qui permet de différencier deux positions différentes associées au même index.

Le choix de valeurs pseudo-aléatoires au départ assure une bonne répartition des clés (la table est remplie équitablement et deux positions se ressemblant auront des clés de hachage très différentes) tandis que le choix du `xor` permet de calculer les clés incrémentalement au cours de la recherche profondeur-d'abord en tenant uniquement compte des pions posés et des pions retournés.

2.3.2 Les entrées de la table

Chaque entrée de la table de transposition contient les informations suivantes sur la position correspondante :

- le verrou (les 32 bits de poids fort de la clé de hachage)
- la « date » de création de cette entrée (utilisée lors d'un conflit)
- la profondeur d'analyse
- une borne inférieure de la note de la position à la profondeur indiquée
- une borne supérieure de la note de la position à la profondeur indiquée
- le meilleur coup trouvé jusqu'à présent dans cette position

Lorsqu'au cours de la recherche, on doit analyser une position à une profondeur p, on commence par calculer sa clé de hachage. On regarde ensuite si le verrou se trouvant dans l'entrée désignée par l'index correspond à celui de la position. Si ce n'est pas le cas, il s'agit de deux positions différentes ; la table de transposition n'est ici d'aucun secours. Si par contre les verrous correspondent, on compare la profondeur de recherche en cours avec celle de l'entrée. Si la profondeur stockée dans l'entrée est supérieure ou égale à celle demandée, on réajuste la fenêtre $[\alpha, \beta]$ en fonction des bornes trouvée dans l'entrée (α devient $max(\alpha, borne_{inf})$ et *beta* devient $min(\beta, borne_{sup})$). Si on obtient une fenêtre $[\alpha, \beta]$ telle que $\alpha \geq \beta$ alors on provoque un élagage correspondant au cas où les informations contenues dans l'entrée suffisent à éviter toute nouvelle recherche dans cette position. Dans tous les autres cas, on effectue une recherche Alpha-Beta classique en essayant comme premier coup celui se trouvant dans l'entrée.

2.3.3 La règle de résolution de conflits

Une règle de conflit est un algorithme qui détermine lorsque le résultat de l'analyse d'une position doit être stocké dans la table de transposition si l'entrée correspondant à la clé de hachage calculée est conservée (et donc le résultat venant d'être trouvé est « oublié ») ou si l'entrée est mise à jour avec la nouvelle information. L'algorithme utilisé dans TURTLE est le suivant :

1. si les informations de l'entrée proviennent d'une recherche antérieure [1], alors on remplace ces informations par les nouvelles

[1] les *dates* de l'entrée et de la recherche actuelle sont différentes, cette dernière étant incrémentée à chaque nouvelle recherche

2. si la profondeur de recherche stockée dans l'entrée est inférieure ou égale à la profondeur de recherche en cours, alors on écrit
3. sinon l'entrée de la table reste inchangée

Ceci revient à conserver dans la table les informations obtenues aux profondeurs les plus importantes, c'est-à-dire les positions les plus proches de la racine de l'arbre de recherche. Le principe sous-jacent est que lorsque l'une de ces informations provoque un élagage, celui-ci implique un gain de temps important car le sous-arbre élagué est profond et contient à ce titre de nombreux noeuds.

2.4 Les recherches courtes

La table de transposition est une heuristique très efficace pour les positions qui ont déjà été étudiées au cours d'une recherche. Il est néanmoins nécessaire d'utiliser d'autres techniques pour les positions qui n'ont jamais été analysées. La notion de recherche courte consiste lorsqu'une position doit être étudiée à la profondeur n à :

- générer tous les coups légaux dans la position
- effectuer une recherche Alpha-Beta à une profondeur $p < n - 1$ pour chacun de ces coups et conserver le résultat obtenu
- trier les coups légaux par notes décroissantes
- lancer une recherche Alpha-Beta à profondeur n sur la position en essayant les coups dans l'ordre préconisé par ce tri

Cette heuristique — basée en fait sur la fonction d'évaluation — trie très bien les coups mais est malheureusement très coûteuse en temps. Dans TURTLE, son utilisation est limitée aux positions à étudier à des profondeurs $n \geq 8$ et l'analyse de chaque coup est effectué à la profondeur $p = 1$.

2.5 Les listes de coups meurtriers

Lorsque ni la table de transposition ni les recherches courtes n'apportent d'information il faut appliquer une heuristique simple, et donc rapide, car les positions concernées sont encore nombreuses. Les listes de coups meurtriers (« killer lists ») permettent d'avoir pour toute position à étudier une liste ordonnées de coups (par forcément légaux) à essayer lors de la recherche. L'ordre de ces listes est dicté non pas par des appels à la fonction d'évaluation (éventuellement indirects dans le cas des recherches courtes) coûteuse en temps d'exécution mais par des statistiques sur les élagages produits par un même coup précédemment au cours de la recherche.

Les listes de coups meurtriers consistent en un tableau à deux dimensions indexé par la couleur du joueur au trait dans la position actuelle et le dernier coup joué pour aboutir à cette position. Chacun des éléments de ce tableau est une liste ordonnée de toutes les cases vides, celles donnant lieu à des coups potentiellement meilleurs en premier. Au lancement du programme, toutes les listes sont ordonnées suivant le type des cases d'un othellier : les coins, puis les cases A, F, B, G, E, D, C, puis X[2]. A chaque position, une liste est associée suivant la couleur du joueur au trait et le dernier coup joué. A l'issue de la recherche sur cette position, le coup déterminé comme étant le meilleur est permuté dans la liste avec son prédécesseur s'il en a un.

Une liste de cases vides étant disponible pour toute position à étudier, le générateur de coups légaux teste en respectant l'ordre de la liste si un coup joué provoque au moins un retournement de pion adverse. Cette heuristique trie moins bien les coups que les recherches courtes mais est nettement moins coûteuse en temps de calcul.

[2] voir l'annexe A

Chapitre 3

La fonction d'évaluation

3.1 Introduction

Dans le cadre d'une utilisation par un algorithme de recherche de type Min-Max, une fonction d'évaluation fait correspondre à toute position d'un jeu donné une valeur numérique qui représente les chances de gain d'un joueur (par exemple celui ayant le trait dans cette position).

Le calcul proprement dit de cette fonction d'évaluation se fait en deux temps :

1. on évalue les différents critères d'évaluation de la position
2. on combine les valeurs de ces critères pour obtenir une valeur unique

3.2 Les critères d'évaluation

Les critères d'évaluation sont des caractéristiques d'une position que l'on doit pouvoir exprimer numériquement et qui doivent aider à prédire le résultat de la partie parfaite (au sens Min-Max) issue de cette position.

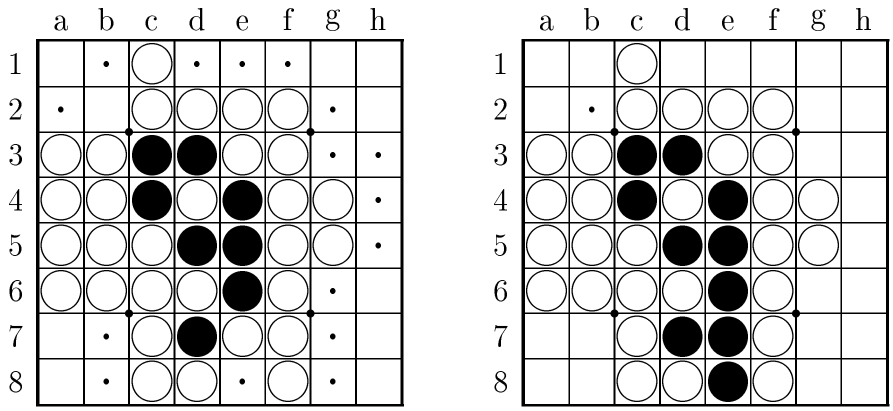
Du fait de la nature Min-Max de l'algorithme de recherche, et en particulier que ce type d'algorithme ne peut être appliqué qu'à des jeux à somme nulle (ce qui est gagné par un joueur est perdu par l'autre et inversement), la valeur d'un critère pour une position donnée est en fait la différence des valeurs de ce critère pour les deux joueurs.

Le compromis vitesse/connaissance intervient dans le choix des critères d'évaluation :

- ajouter des critères apporte de la connaissance mais ralentit l'évaluation globale
- supprimer des critères accélère l'évaluation au prix d'une diminution de la connaissance
- on peut choisir de programmer partiellement un critère (c'est-à-dire d'en calculer une valeur approchée) si l'évaluation s'en trouve accélérée
- certains critères peuvent être fortement *corrélés* et il peut dans ce cas être judicieux de choisir un sous-ensemble de critères plus *indépendants*

3.2.1 La mobilité (ou mobilité réelle)

La mobilité d'un joueur est le nombre de coups légaux dont dispose ce joueur dans une position donnée. Il a été constaté qu'avoir une grande mobilité est un facteur primordial à Othello. Dans la position où Noir à la trait de la figure 3.1, après un coup en e8, la mobilité de Blanc est de 1 coup ! Ce dernier est obligé de jouer en b2 — offrant le coin a1 sans compensation à son adversaire [1] — et Noir gagne donc facilement. C'est la réduction de la mobilité de Blanc qui a permis à Noir de remporter la partie car Blanc n'aurait pas donné de coin s'il avait eu le choix.



à Noir de jouer — à Blanc de jouer

FIG. 3.1: *La mobilité réelle*

L'inconvénient de la mobilité est que l'on ne peut pas la calculer rapidement. TURTLE calcule en fait une approximation de la mobilité. La mobilité globale est approchée par une somme de mobilités locales qui elles sont calculées exactement. Les mobilités locales sont les mobilités calculées sur les sous-ensembles *unidimensionnels* des cases d'une position, c'est-à-dire les lignes, les colonnes et les diagonales. Cette approche est justifiée a posteriori en comparant numériquement les pouvoirs prédictifs de la mobilité exacte et de la mobilité approchée, en tenant compte du fait que la mobilité approchée nécessite un temps de calcul trois fois inférieur à peu près.

La figure 3.2 est un exemple de position où la mobilité approchée diffère de la mobilité exacte. Le coup f4, légal pour Noir, est compté 4 fois car il est « légal » horizontalement, verticalement et le long des deux diagonales ; de même pour c7 et b6 qui sont comptés respectivement 3 et 2 fois pour blanc.

Techniquement, le calcul de la mobilité approchée est effectué en deux temps :

Au lancement, TURTLE alloue et remplit 9 tables qui correspondent aux regroupements de cases différents nécessaires à ce calcul, à savoir les quatre différents types de lignes et les

[1]ce qui est comparable à une perte de pièce aux échecs

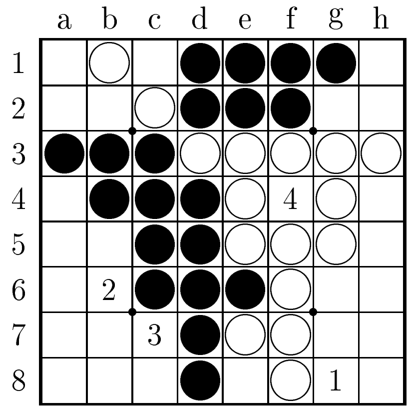


Fig. 3.2: *L'approximation de la mobilité réelle*

diagonales de longueur quatre au moins (ce qui correspond aux patterns Edge, Hor2, Hor3, Hor4, Diag4, Diag5, Diag6, Diag7 et Diag8[2]). Les diagonales de longueur un et deux n'interviennent pas dans la mobilité et les diagonales de longueur trois sont ignorées étant donné qu'un coup légal dans ces diagonales retourne forcement une case X[3] ce qui donne le coin à l'adversaire dans la majorité des positions. Dans chacune de ces tables, on stocke les mobilités locales de toutes les instances possibles (une *instance* d'une pattern est obtenue en fixant la « valeur » de chaque case de la pattern à case vide, pion noir ou pion blanc ; chaque case pouvant avoir une valeur différente). Pour une pattern de n cases, il y a 3^n instances possibles. Chacune de ces instances peut être représentée par un nombre entier compris entre 0 et $3^n - 1$ inclus (il est facile de le coder en base 3). Chaque table correspondant à une pattern de n cases contiendra 3^n entiers qui sont chacun la mobilité locale d'une instance de cette pattern.

A l'exécution, pour chacune des 9 patterns et pour chaque symétrie on va chercher dans la table correspondant à la pattern la mobilité locale de l'instance en indexant la table par le numéro de l'instance. La rapidité du calcul de la mobilité dépend ainsi de la vitesse du calcul des indices correspondant aux instances. Dans Turtle, un calcul incrémental (et donc rapide) de ces indices est effectué, partant de la constatation que 40 des 64 cases de l'othellier (plateau de jeu à Othello) sont chacune comprises dans quatre patterns distinctes (en comptant les symétries). Avec quatre indices modifiés par un placement/retournement de pion, on peut donc calculer incrémentalement les indices de toutes les symétries des patterns Edge, Hor2, Hor3, Hor4, Triangle (au lieu de Diag4), Diag5, Diag6, Diag7 et Diag8. Les informations sur Diag4 peuvent être placées dans la table, plus grande, de Triangle et seront accessibles aussi rapidement.

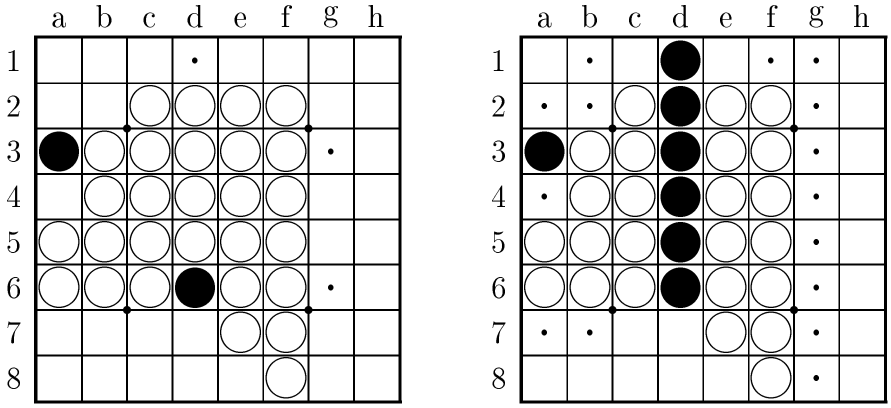
3.2.2 La frontière (ou mobilité potentielle)

La frontière d'un joueur est l'ensemble des pions qu'il possède qui sont contigus à au moins une case vide. On appelle ces pions les *pions frontière*. Le terme de mobilité poten-

[2]voir l'annexe D
[3]voir l'annexe A

tielle vient du rapport étroit entre mobilité et frontière : tout coup légal pour un joueur à Othello retourne un pion frontière de son adversaire. Le nombre de coups légaux d'un joueur est donc lié aux nombre et à la répartition des pions frontière de l'adversaire.

La figure 3.3 donne un exemple de position où Noir n'a que 3 coups légaux. Il est pourtant loin d'avoir perdu car après un coup en d1, il possède 15 coups légaux ! Ici cette mobilité de Noir après d1 pouvait être « devinée » en comptant le nombre de pions frontière de Blanc avant ce coup. La frontière de Blanc représentait la *mobilité potentielle* de Noir.



à Noir de jouer — après un coup de Noir en d1

FIG. 3.3: *La mobilité potentielle*

La frontière (on considère ici le nombre de pions frontière) possède des avantages séduisants comparée à la mobilité :

1. elle est plus rapide à calculer aussi bien pour les joueurs humains que pour les programmes
2. elle est plus stable : un avantage de frontière est plus durable qu'un avantage de mobilité
3. elle aide à prédire la mobilité future

Trois critères numériques basés sur la frontière sont couramment utilisés dans les programmes d'Othello :

1. le nombre de pions frontière
2. le nombre de cases vides contiguës à la frontière
3. la somme des nombres de cases vides contiguës à chaque pion frontière (une même case vide pouvant être comptée plusieurs fois)

TURTLE calcule le troisième pour deux raisons :

– c'est le moins corrélé avec la mobilité
– il est calculable exactement avec les même tables que celles utilisées pour la mobilité approchée

Ce troisième critère est en effet égal à la somme sur toutes les lignes, colonnes et diagonales du calcul local de ce même critère. Les valeurs sont stockées de la même manière que pour la mobilité.

3.2.3 Les *patterns*

L'idée d'utiliser les patterns comme critères d'évaluation provient du fait que puisque l'on a besoin d'accéder à des tables pour calculer la mobilité et la frontière, on peut ajouter autant d'information que l'on veut sur les instances d'une pattern déjà utilisée du moment que les effets se cumulent linéairement. Cela signifie qu'en un accès mémoire on peut récupérer une somme pondérée d'un nombre quelconque d'informations numériques sur une instance.

Ceci signifie déjà que l'on peut facilement ajouter de nouveaux critères si ils sont évaluables (exactement ou de manière approchée) par une somme de critères locaux aux patterns utilisées dans le programme.

Estimation des patterns

Dans TURTLE, un nouveau critère a été intégré dans chaque pattern utilisée pour le calcul de la mobilité. En partant d'une base de données de parties jouées, on peut, en la stockant sous la forme d'un arbre de positions, étiqueter chaque position avec la valeur -1 si elle a été perdue par Blanc, 0 si elle donné lieu à une partie nulle et $+1$ si elle a été gagnée par Blanc. La valeur du nouveau critère pour une instance donnée est la moyenne des valeurs des positions de la base qui contiennent l'instance. Le concept sous-jacent est donc la notion empirique de proportion de gain dans une base d'exemples de positions contenant un groupement de pions donné. La qualité des valeurs obtenues dépend à la fois de la manière de sélectionner les parties de la base et de la façon d'étiqueter les positions.

Si l'on prend l'exemple de la pattern Edge[4], le critère pattern permet d'évaluer numériquement la possession d'un coin, d'une case C[5], ou de toute configuration de bord. On voit bien que ce critère permet de coder de manière automatique des notions qui font intervenir des relations non linéaires entre les cases, sans que l'on aie complètement spécifié la nature de ces relations.

Sélection de nouvelles patterns

Si l'évaluation de ces nouveaux critères (un par pattern) est gratuite en terme de temps consommé lors de l'exécution (ce qui n'est pas le cas du temps d'apprentissage), l'information supplémentaire apportée n'est pas énorme comparée à la mobilité et la frontière (mis à part Edge qui donne de bons résultats). On a donc essayé d'autres patterns que celles utilisées pour le calcul de la mobilité. C'est ainsi qu'a été découverte la pattern Corner4×2

[4]voir l'annexe D
[5]voir l'annexe A

qui discrimine encore mieux les gains des pertes que la pattern de bord Edge.

Le choix de nouvelles patterns est motivé par des raisons différentes et souvent contradictoires :

- le pouvoir discriminant
- le temps de calcul nécessaire à l'exécution
- le nombre de cases de la pattern (trop petit, on obtient pas assez d'information ; trop grand, on a pas assez d'exemples pour calculer les valeurs)
- le nombre de symétries (dans un plateau d'Othello, il y a huit patterns Corner4×2 mais seulement deux patterns Diag8) qui influe à la fois sur le nombre d'exemples nécessaires à son évaluation et la quantité d'accès mémoire à l'exécution
- le nombre de cases communes avec les autres patterns (corrélation)
- l'inclusion dans une autre pattern («gratuité» des accès mémoire)
- la vitesse de calcul de l'index de la pattern (valeur numérique caractérisant une instance et permettant de trouver l'information lui correspondant)

Choix effectué dans Turtle

Le choix final effectué dans Turtle est présenté an annexe D.

On remarque l'inclusion de certaines patterns dans d'autres :

- Diag4 et CornerCar dans Triangle
- Edge dans Edge+2X
- Corner4×2 dans Corner5×2

Les patterns dynamiques

Il a été constaté qu'à Othello, la manière de bien jouer dépend du degré d'avancement de la partie, ce qui devrait apparaître dans les patterns sous la forme d'une évolution de la note d'une instance donnée lorsque le numéro du coup (que l'on définit comme le nombre total de pions duquel on retranche les 4 pions de la position de départ) augmente.

Pour tenir compte de ce facteur, on ajoute comme index supplémentaire pour les tables correspondant aux patterns de huit cases et moins le nombre total de pions présents dans la position à analyser (les bases de données ne contiennent pas assez d'exemples pour calculer les valeurs dynamiques de patterns de 10 cases). Bien qu'il y aie dans ce cas 60 phases lors d'une partie, on ne s'intéresse qu'aux numéros de coups situés entre 12 et 52 inclus. En effet, une bibliothèque d'ouvertures ou une recherche à 12 demi-coups de profondeur d'une part, et une recherche exacte de fin de partie d'autre part permettent de s'affranchir des numéros de coups non compris dans cet intervalle. Le nombre d'étapes est alors de 41.

Il est nécessaire que le nombre d'étapes soit important afin de garantir une évolution lente des valeurs et ainsi d'éviter les effets désastreux de changements trop brutaux de celles-ci lors d'un changement d'étapes. Les tables pouvant être regroupées par étape de partie, les accès mémoire sont pratiquement aussi rapides que s'il n'y avait aucune gestion de ces étapes. Il ne faut pas non plus négliger l'inconvénient qui est de taille : la taille mémoire nécessaire est 41 fois supérieure ! Néanmoins la taille mémoire nécessaire dans ce cas n'excède pas 8 Mo, ce qui est raisonnable sur les stations de travail actuelles.

Le lissage des patterns dynamiques

Un inconvénient plus grave des patterns dynamiques est que pour le calcul de la valeur d'une instance de pattern au coup numéro n, seules les positions ayant le même numéro de coup n sont exploitées. Les valeurs ainsi calculées sont donc moins «sûres». Par ailleurs, certaines instances sont totalement absentes de la base de donnée pour certains numéros de coups.

La solution utilisée dans TURTLE est de lisser numériquement toutes les courbes obtenues en traçant les valeurs d'une instance (notées a^0 le long de l'axe y) en fonction du numéro de coup (noté i le long de l'axe x). Puis les valeurs successives de la suite de terme général suivant sont calculées jusqu'à ce que la différence entre deux itérations successives passe en dessous d'un seuil fixé :

$$a_i^n = c_i a_i^0 + (1 - c_i)\frac{a_{i-1}^{n-1} + a_{i+1}^{n-1}}{2}$$

où a_i^n est la nouvelle valeur pour le numéro de coup i à l'itération n (a_i^0 étant la valeur de départ, $c_i = \sqrt{\frac{n_i}{n_t}}$ si $n_t \neq 0$ et $c_i = 0$ sinon, est la *confiance* attribuée à la valeur de départ a_i^0 (n_i étant le nombre d'exemples pour le coup i et n_t le nombre total d'exemples).

Cette méthode, bien que simple numériquement, corrige à la fois les valeurs bruitées, interpole ou extrapole les valeurs là où il n'y a pas d'exemple. Elle donne de meilleurs résultats qu'en faisant intervenir un chevauchement des étapes lors de la collecte des exemples.

3.3 La combinaison des critères d'évaluation

3.3.1 Généralités

Tout comme pour le choix des critères, le compromis vitesse/connaissance intervient dans le choix de la nature de la combinaison des critères :

- une combinaison linéaire des critères est très rapide à calculer, mais ne permet pas de faire intervenir des relations complexes entre eux
- un réseau de neurones multi-couches aurait, dans le cadre d'une fonction d'évaluation, un bon pouvoir de prédiction mais au prix d'un temps de calcul très important

Entre ces deux extrêmes, beaucoup d'essais ont été effectués. Cependant, à Othello, les fonctions d'évaluation linéaires l'ont toujours emporté. Une partie de la raison est qu'Othello est un jeu à facteur de branchement (nombre apparent de coups légaux avec élagages) faible (entre 3 et 4) et qu'augmenter la profondeur de recherche améliore également le niveau de jeu du programme.

3.3.2 Le choix fait dans TURTLE

Présentation

La fonction d'évaluation de TURTLE renvoie la valeur suivante :

$$\pi = \frac{1}{1 + \exp(-\sum_{i=1}^{n} a_i c_i)}$$

où n est le nombre de critères, c_i la valeur du critère i et a_i le coefficient associé au critère i. Les coefficients a_i ont été calculés par régression logistique (sur la même base de positions d'exemple que celle décrite page 19), ce qui signifie que ce sont les coefficients correspondant au maximum de vraisemblance (comparable à la méthode des moindres carrés mais davantage utilisé pour comparer des probabilités). La valeur de l'évaluation prédit donc la probabilité de gain du joueur ayant le trait dans la position étudiée.

La régression logistique

On représente les exemples de positions par une matrice $\mathbf{X}$ de taille $(N \times n)$ de terme général x_{ij} qui est la valeur du critère d'évaluation j pour la position d'exemple i, et par un vecteur colonne $\mathbf{y}$ de taille N de terme général y_i qui représente le résultat de la partie issue de la position i : il vaut 0 pour une partie perdue par Blanc, $\frac{1}{2}$ pour une partie nulle et 1 pour une partie gagnée. On cherche le vecteur colonne de poids $\mathbf{a}$ maximisant la vraisemblance $L(\mathbf{a})$ qui est le produit des vraisemblances pour chaque position i entre $\pi_i = \frac{1}{1+\exp(-\mathbf{a}\mathbf{x}_i)}$ (la valeur calculée) et y_i (la valeur connue). La vraisemblance de a pour la position i étant $L(a, y_i) = \pi_i^{y_i}(1 - \pi_i)^{1-yi}$.

Pour trouver la valeur optimale du vecteur $\mathbf{a}$, il est nécessaire de résoudre un système d'équations non-linéaires, ce qui peut être fait par la méthode itérative de NEWTON-RAPHSON. Pour garantir la convergence, une adaptation du modèle statistique doit être effectuée : il faut choisir une constante entière n_i et remplacer dans le vecteur $\mathbf{y}$ les 0 par des 1, les $\frac{1}{2}$ par des $\frac{n_i}{2}$ et les 1 par des $n_i - 1$. $\frac{\mathbf{y}}{n_i}$ est une approximation du vecteur $\mathbf{y}$ de l'ancien modèle d'autant plus proche de celui-ci que n_i est grand. La valeur $n_i = 100$ a été choisie pour les calculs.

$$\hat{\mathbf{a}}^{(t+1)} = (\mathbf{X}'\mathbf{\Delta}^{(t)}\mathbf{X})^{-1}\mathbf{X}'\mathbf{\Delta}^{(t)}\mathbf{z}^{(t)}$$

où :

- $\mathbf{\Delta}^{(t)} = \mathbf{diag}[n_i\hat{\pi}_i^{(t)}(1-\hat{\pi}_i^{(t)})], \hat{\pi}_i^{(t)} = [1+\exp(-\sum_{j=1}^{n}\hat{a}_j^{(t)}x_{ij})]^{-1}$
- $z_i^{(t)} = log(\frac{\hat{\pi}_i^{(t)}}{1-\hat{\pi}_i^{(t)}}) + \frac{y_i - n_i\hat{\pi}_i^{(t)}}{n_i\hat{\pi}_i^{(t)}(1-n_i\hat{\pi}_i^{(t)})}$

L'implémentation

Malgré les apparences, la relation entre les critères est linéaire. Il suffit en effet de calculer tout d'abord $x = \sum_{i=1}^{n} a_i c_i$ puis $\pi(x) = \frac{1}{1+\exp(-x)}$. La première opération est le calcul classique d'une fonction d'évaluation linéaire, et la seconde peut être pré-calculée dans un tableau car elle ne fait intervenir qu'une seule variable x.

L'évaluation dynamique

Tout comme les valeurs de patterns lors de la phase d'estimation, les coefficients de la fonction d'évaluation dépendent de la phase de la partie, représentée par le numéro du coup.

Pour assurer la variation lente, le coefficient correspondant au coup i est calculé à partir des exemples dont le numéro de coup est compris dans l'intervalle $[i-2, i+2]$.

Chapitre 4

La gestion du temps

4.1 Introduction

La gestion du temps est un aspect des jeux de réflexion (dans des conditions de tournoi) dont on parle assez peu. Il est pourtant important pour un programme comme pour un joueur humain d'utiliser au mieux le temps dont il dispose pour jouer sa partie. A Othello, étant donné que toutes les parties durent 60 coups si l'on ne compte pas les cas où un joueur passe son tour, une durée est attribuée à chacun des deux joueurs pour toute la partie et ne sera jamais modifiée (15 à 30 minutes en général dans les tournois). La gestion du temps implémentée dans TURTLE provient de l'application de quelques principes élémentaires :

- utiliser la plus grande portion possible du temps imparti
- ne jamais dépasser la durée totale (sinon le gain est accordé à l'adversaire)
- ne pas lancer de nouvelle recherche si celle-ci a toute les chances de n'apporter aucune information supplémentaire

4.2 L'allocation du temps

Avant le lancement de la recherche Alpha-Beta sur une position à n cases vides — il reste donc n coups à jouer — la durée de cette recherche est calculée comme suit, t étant le temps total restant au programme pour terminer sa partie :

1. la durée d'une résolution complète du jeu à partir de la position est estimée par la formule $d_{fin} = a \exp(bn)$ [1] où a et b sont des constantes évaluées à l'aide de positions test dont la résolution a été chronométrée (a doit en outre être adapté à la puissance de calcul de la machine utilisée)
2. si $t \geq d_{fin}$, une recherche de fin de partie est lancée avec une durée $d = 0,6 \times t$ (pour conserver du temps au cas où la recherche de finale est infructueuse)

[1]il est logique d'utiliser cette formule car le temps pris par une recherche est exponentiel en fonction de la profondeur

3. le nombre de coups restant à jouer pour le programme est évalué : $nb_{coups} = \lfloor \frac{n-11}{2} \rfloor$ (on supprime les tous derniers coups de la partie dont on sait qu'ils seront joués instantanément)
4. on calcule $d = 1,2 \times \frac{t}{nb_{coups}}$, la constante $1,2$ ajoutant toujours 20 % de temps, forçant le programme à réfléchir plus longtemps sur les premiers coups de la partie

La recherche se termine dans l'un des cas suivants :

1. la position a été entièrement résolue (un coup « parfait » au sens du Min-Max a donc été trouvé)
2. la durée impartie pour cette recherche d s'est écoulée
3. une étape de l'approfondissement itératif s'est terminée (la recherche à une profondeur p vient de s'achever) et plus de 40 % du temps imparti pour ce coup s'est écoulé ; étant donné qu'il faut en moyenne 3,2 fois plus de temps pour effectuer une recherche à la profondeur $p + 1$, il est préférable de conserver le temps restant pour les coups à venir

4.3 La réflexion sur le temps adverse

Lorsque c'est à son adversaire de jouer, Turtle effectue une recherche du meilleur coup pour l'adversaire, mais en n'allouant qu'un dixième du temps qui serait pris si le programme avait le trait. Une fois cette recherche terminée, Turtle lance une autre recherche sans limite de temps (l'allocation de la durée de recherche est effectuée mais on autorise le dépassement de cette durée tant que l'adversaire ne joue pas son coup) en supposant que le coup ainsi trouvé est joué par l'adversaire (ce coup est joué dans un othellier interne qui contient alors la position de départ de la recherche). Deux évènements peuvent alors se produire :

– l'adversaire joue le coup attendu ; la recherche en cours se poursuit mais le système de détection de dépassement du temps est réenclenché, ce qui produit un arrêt instantané si la durée allouée était déjà dépassée
– l'adversaire choisit un autre coup ; la recherche est stoppée et une nouvelle est relancée sur la position correcte

On ne gagne de temps que si l'adversaire joue le coup attendu, ce qui représente tout de même plus de la moitié des cas dans une partie de tournoi. Cette technique est donc très importante même si l'implémentation en est malaisée (gestion des interruptions, ...).

Chapitre 5

La bibliothèque d'ouvertures

5.1 Introduction

Les programmes d'Othello jouent à un très bon niveau à l'aide de leur algorithme de recherche, de leurs heuristiques, et de leur fonction d'évaluation. Néanmoins des problèmes sont rencontrés dans la phase d'ouverture :

- le coup joué dans une position donnée est toujours le même (déterminisme) ; les mêmes calculs sont toujours réeffectués, alors qu'un coup, une fois calculé, pourrait être stocké en vue d'une récupération instantanée
- les conséquences d'un coup peuvent intervenir très tardivement dans une partie ce qui ne peut être prévu par une recherche de type milieu de partie
- le déterminisme est lui-même un problème car si un adversaire gagne une partie, il lui suffit de rejouer les mêmes coup pour l'emporter une nouvelle fois

L'idée de la bibliothèque d'ouvertures est de stocker les coups déjà joués dans le but de les jouer instantanément lorsque la position se reproduit.

5.2 La structure interne

La bibliothèque d'ouvertures de TURTLE est constituée d'un ensemble de parties (éventuellement incomplètes) stocké sous la forme d'un arbre, chaque noeud représentant une position. La racine de l'arbre correspond à la position de départ à Othello.

Chaque noeud contient deux valeurs : un score et une note.

Le score d'un noeud représente un résultat de partie. Pour les feuilles de l'arbre correspondant à une position terminale, le score est la différence de nombre de pions entre ceux du joueur ayant le trait et son adversaire (celui qui a joué le dernier coup). Pour les feuilles correspondant à des positions non terminales, un score fictif de -66 est associé (dans le pire des cas à Othello, un joueur perd par -64).

La note d'un noeud représente une évaluation de milieu de partie. Pour les feuilles de l'arbre correspondant à une position terminale, la note est : $-\infty$ pour une position perdue, 0 pour une position nulle et $+\infty$ pour une position gagnée. Pour les feuilles correspondant à des positions non terminales, la note est l'évaluation de la position par l'Alpha-Beta.

5.3 Le choix d'un coup

Le choix d'un coup dans tout noeud de l'arbre est dicté par un algorithme de recherche Min-Max tenant à la fois compte des scores et des notes stockées dans les feuilles de l'arbre. Comme la bibliothèque d'ouvertures change peu après l'ajout d'une nouvelle partie, une quantité substantielle de calculs peut être économisée en établissant la valeur théorique de chaque noeud une fois pour toutes :

Le score est calculé pour chaque noeud interne de l'arbre en faisant remonter les scores des feuilles jusqu'à la racine en suivant le principe du Min-Max.

Pour toute position *perdue* de l'arbre (noeud interne dont le score est négatif ou nul) ne possédant aucun fils ayant pour score -66, une recherche de la *meilleure alternative* est effectuée à l'aide de l'algorithme de milieu de partie. Il s'agit du choix fait par la recherche Alpha-Beta entre tous les coups légaux qui ne sont pas déjà présents en tant que fils du noeud en question. A chaque position perdue de la bibliothèque on associe donc un coup *alternatif* qui est ajouté dans l'arbre des parties avec sa note de milieu de partie et un score fictif de -66 indiquant qu'il s'agit d'un nouveau coup.

Une seconde remontée est alors effectuée sur les notes cette fois-ci. Une restriction est ajoutée comparativement à un Min-Max classique : dans les noeuds internes dont le score est strictement négatif, seuls les fils dont le score est strictement positif sont évalués.

Le choix d'un coup dans une position correspondant à un noeud de la bibliothèque est celui fait par l'algorithme suivant :

1. s'il existe au moins un fils dont le score est strictement positif, tous les fils ayant un score négatif ou nul sont éliminés (il ne sont pas envisagés par l'algorithme mais restent stockés dans l'arbre).
2. seuls les fils restants ayant la meilleure note (la note maximum sur tous les fils restants) sont conservés
3. seuls les fils restants ayant le meilleur score (le score maximum sur tous les fils restants) sont conservés
4. le premier des fils restants est choisi

5.4 L'auto-apprentissage

L'apprentissage d'une partie complète consiste à ajouter cette partie dans l'arbre puis à recalculer l'évaluation des noeuds. Ceci implique automatiquement la recherche puis l'incorporation de nouveaux coups alternatifs, ce qui prend du temps (dans TURTLE, une minute est allouée à chaque recherche de coup alternatif).

L'auto-apprentissage consiste à itérer l'algorithme suivant :

1. choisir dans tout l'arbre le coup alternatif possédant la meilleure note
2. faire jouer le programme contre lui-même dans des conditions de tournoi à partir de la position ainsi désignée
3. intégrer la partie une fois terminée dans l'arbre
4. mettre à jour les scores et notes des noeuds internes, comme dans le cas d'ajout d'une partie normale

Annexe A

Le jeu d'Othello

Othello est un jeu de stratégie à deux joueurs : Noir et Blanc. Il se joue sur un plateau unicolore de 64 cases, 8 sur 8, appelé othellier. On repère les cases par un système de coordonnées : les lignes sont numérotées de haut en bas, de 1 à 8 ; les colonnes sont étiquetées de gauche à droite, de 'a' à 'h'.

	a	b	c	d	e	f	g	h
1	a1	b1	c1	d1	e1	f1	g1	h1
2	a2	b2	c2	d2	e2	f2	g2	h2
3	a3	b3	c3	d3	e3	f3	g3	h3
4	a4	b4	c4	d4	e4	f4	g4	h4
5	a5	b5	c5	d5	e5	f5	g5	h5
6	a6	b6	c6	d6	e6	f6	g6	h6
7	a7	b7	c7	d7	e7	f7	g7	h7
8	a8	b8	c8	d8	e8	f8	g8	h8

FIG. A.1: *Les coordonnées de l'othellier*

Chaque joueur dispose de 32 pions bicolores, noirs d'un coté et blancs de l'autre. La couleur d'un pion est celle de sa face visible, au dessus. La position de départ est celle de la figure A.2. Noir commence toujours et les deux adversaires jouent ensuite à tour de rôle.

A son tour de jeu, le joueur doit poser un pion de sa couleur sur une case vide de l'othellier, adjacente à un pion adverse. Il doit également, en posant son pion, encadrer un ou plusieurs pion(s) adverse(s) entre le pion qu'il pose et un pion à sa couleur, déjà placé sur l'othellier. Il retourne alors de sa couleur le ou les pion(s) qu'il vient d'encadrer. Les pions ne sont ni retirés de l'othellier, ni déplacés d'une case à l'autre.

Il n'y a pas de réaction en chaîne : les pions retournés ne peuvent pas servir à en retourner d'autres lors du même tour de jeu. Ainsi dans la position où Blanc a le trait de la figure A.3, Blanc joue en b5, retournant notamment e5. Bien que e4 et e6 soient alors encadrés, ils ne

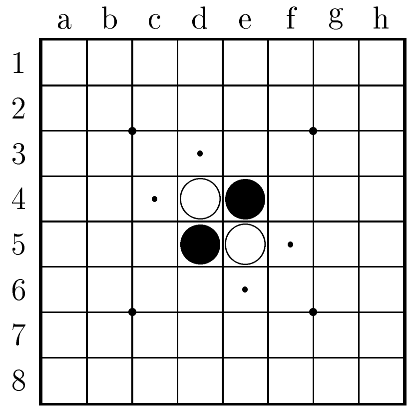
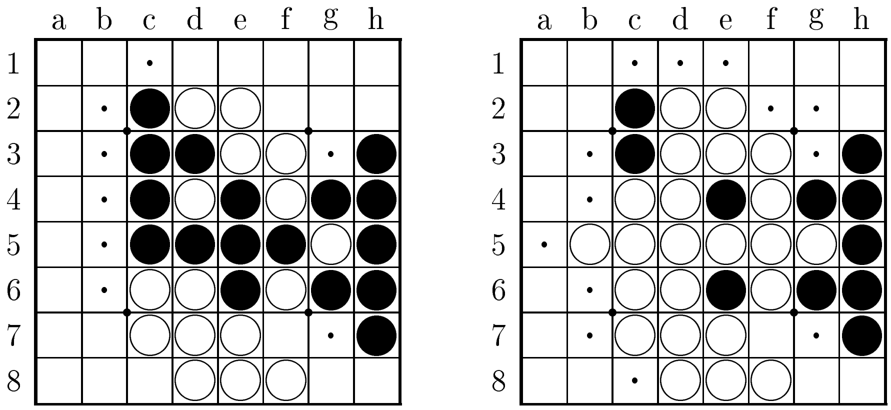


FIG. A.2: *La position de départ, trait à Noir*

sont pas retournés car non encadrés par b5 et un autre pion blanc. Les points représentent les cases où le joueur ayant le trait peut placer un pion en retournant au moins un pion adverse.



à Blanc de jouer — après un coup de Blanc en b5

FIG. A.3: *Blanc joue b5*

Si à un moment de la partie, le joueur ayant le trait ne peut pas poser de pion en retournant un pion adverse, il passe son tour et c'est à son adversaire de jouer. La partie se termine lorsqu'aucun des deux joueurs ne peut jouer. On compte alors les pions pour déterminer le score. Le gagnant est celui qui possède le plus de pions (les éventuelles cases vides restantes lui sont attribuées).

Les cases de l'othellier se répartissent en dix types différents — pour des raisons de symétries — représentés par des lettres de l'alphabet, mis à part les coins, et les cases centrales où l'on ne peut jouer car elles sont occupées dès la position de départ.

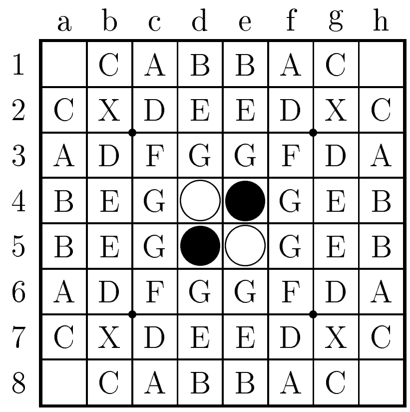


FIG. A.4: *La nomenclature des cases de l'othellier*

Annexe B

Le serveur internet `<ios>`

Le plus important serveur pour joueurs et programmes d'Othello sur internet est `<ios>` (**I**nternet **O**thello **S**erver). Il est accessible par :

`telnet external.nj.nec.com 5000` (dans un interpréteur de commandes)
ou
`telnet://external.nj.nec.com:5000` (dans un navigateur WEB)

Outre les tournois internationaux de programmes qui y sont organisés tous les deux mois et qui permettent de faire jouer entre eux les plus gros calibres de la planète, ce site est très important car de nombreux programmes y sont connectés durant les nuits et les week-ends de leur auteur et jouent constamment les uns contre les autres, permettant la mise en place d'algorithmes d'apprentissage automatique, notamment en ce qui concerne les bibliothèques d'ouvertures. Il est également le point de rencontre et de discussion pour les programmeurs d'Othello de tous niveaux.

Annexe C

Les résultats de Turtle en tournoi

C.1 Le tournoi `<ios>` d'avril 1997

La version de Turtle qui a été présentée à ce tournoi possédait deux *bugs* importants :

1. le programme plantait lorsqu'il restait exactement 3 cases vides sur l'othellier
2. il prenait trop de temps pour jouer ses coups (du moins en apparence) car un autre processus tournant sur la même machine ce jour là perturbait l'algorithme de gestion du temps

Turtle a donc été logiquement éliminé du tournoi, après que le gain dans la première ronde lui ait été accordée (les coups restants de la partie étant forcés). Cet événement serait passé inaperçu si son adversaire n'avait été Logistello, futur vainqueur du tournoi et tout récemment doté d'une nouvelle fonction d'évaluation. Ce résultat, bien que non significatif, est encourageant. Par ailleurs, les bugs repérés grâce à ce tournoi furent rapidement corrigés.

	a	b	c	d	e	f	g	h
1	60	45	38	40	43	41	47	49
2	29	59	14	37	39	44	48	52
3	15	18	7	1	6	8	24	53
4	50	10	9			19	25	56
5	51	11	2			4	33	28
6	17	16	13	12	5	3	30	34
7	54	32	21	20	22	35	46	58
8	55	36	27	26	23	31	42	57

Turtle 38 - 26 Logistello

Fig. C.1: *La seule partie de* Turtle *lors du tournoi* `<ios>` *d'avril 1997*

C.2 Le tournoi <ios> de juin 1997

Le classement final de ce tournoi est donné figure C.2. TURTLE est 3e ex æquo mais termine derrière YINGYANG au départage. Ce résultat est du à la perte contre YINGYANG justement à la dernière ronde du tournoi (les deux premiers au classement étant hors d'atteinte pour le moment). YINGYANG a gagné cette partie grâce à sa bibliothèque d'ouvertures (TURTLE, trop récent, n'en possédait pas) et à la machine sur laquelle il tournait (un PentiumPro à 200 MHz), trois fois plus rapide que la Sparc4 de TURTLE.

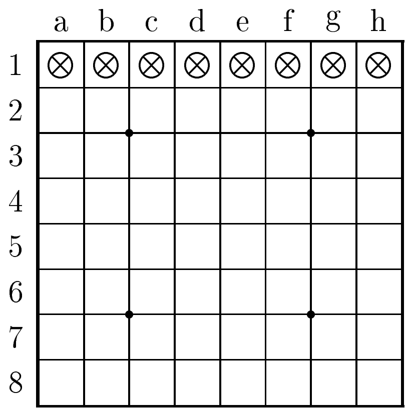
Classement	Nom	Machine	Score	BQ	Pions
1	HANNIBAL	Pentium 166 MHz	11	988	476
2	LOGISTELLO	Alpha 500 MHz	10.5	994	450
3	YINGYANG	PPro 200 MHz	7	900	420
-	TURTLE	Sparc4 110 MHz	7	860	415
5	BRUTUS	Pentium 166 MHz	6.5	838	436
-	SNAIL	Sparc4 110 MHz	6.5	763	430
7	TADPOLE	SparcULTRA1 140 MHz	6	759	373
8	SCOOTER		5	719	359
-	OO7		5	550	321
10	BOOBY reversi		4	830	417
11	Michael HANDEL		3	570	272
12	Marc TASTET		2.5	629	275
13	NEMFLUM		0	143	39

FIG. C.2: *Le classement final*

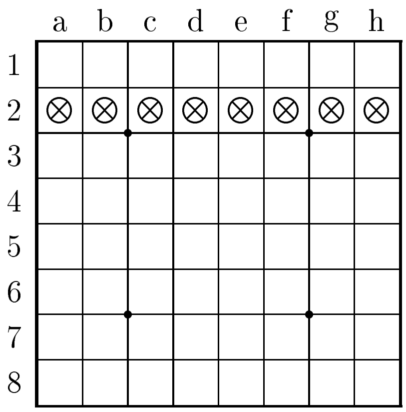
Annexe D

Les patterns

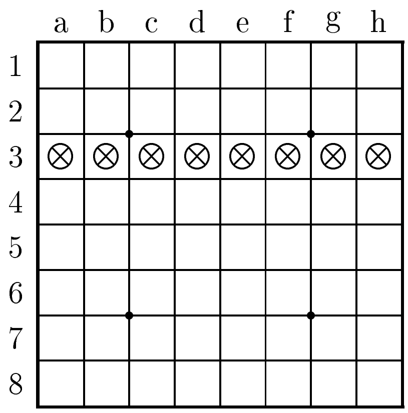
Un exemple de chaque « pattern » (regroupement de cases) utilisé par TURTLE est présenté ici. Les autres exemples se déduisent par symétrie (les colonnes notamment sont analogues aux lignes par symétrie).



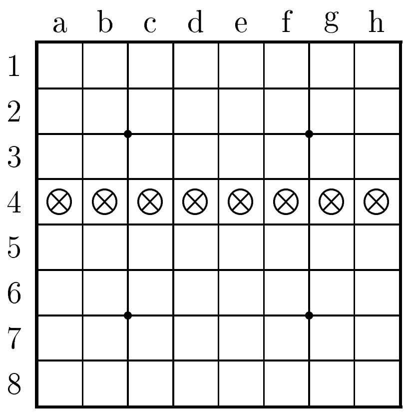
pattern Edge



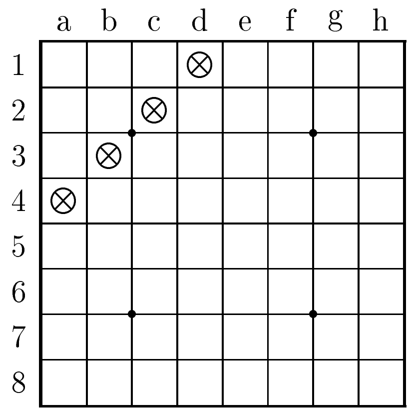
pattern Hor2



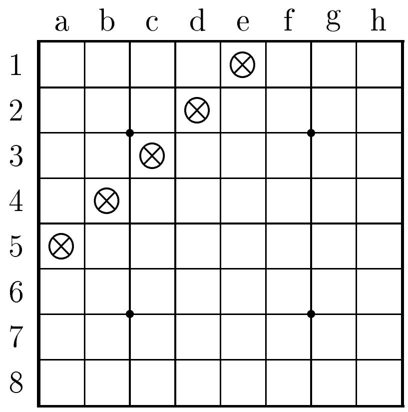
pattern Hor3



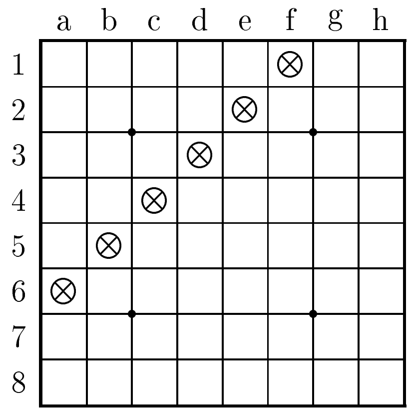
pattern Hor4



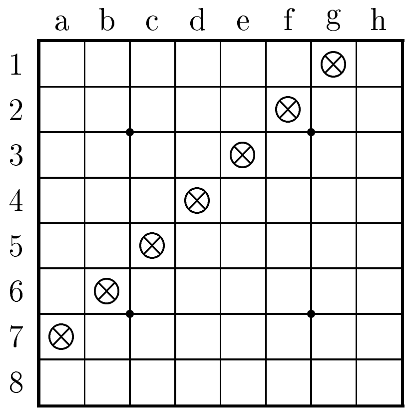
pattern Diag4



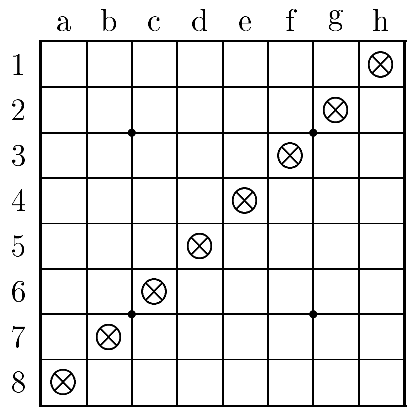
pattern Diag5



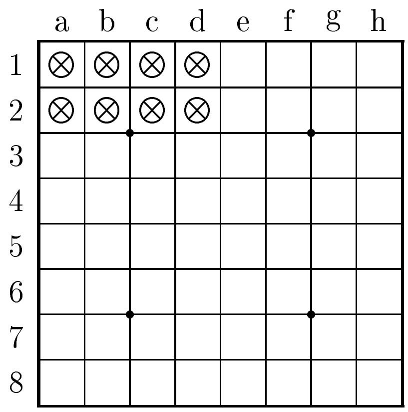
pattern Diag6



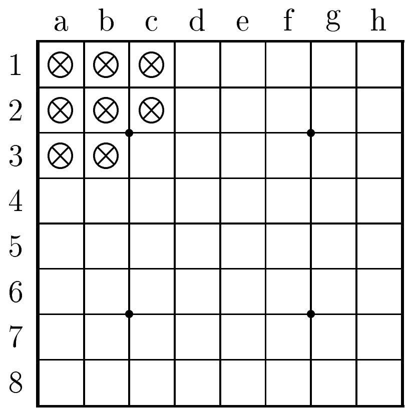
pattern Diag7



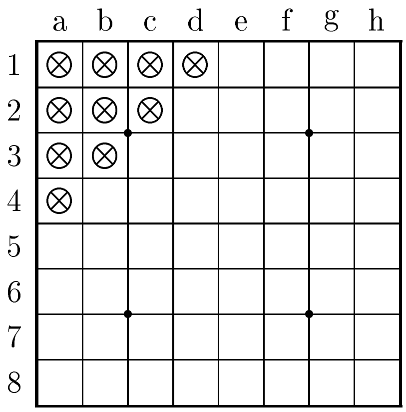
pattern Diag8



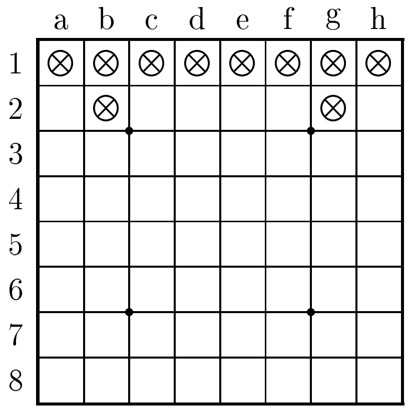
pattern Corner4×2



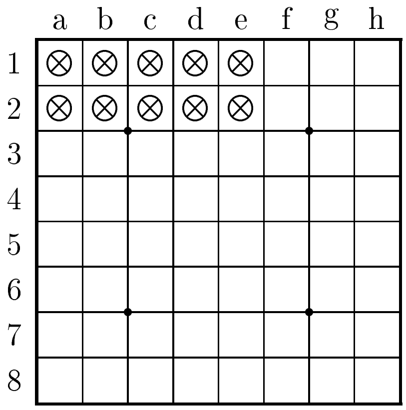
pattern CornerCar



pattern Triangle



pattern Edge+2X



pattern Corner5×2