CHAPITRE 15

La programmation des jeux

Jean-Marc Alliot — Thomas Schiex

15.1 Introduction

Les jeux, activité futile en apparence, ont toujours fasciné les informaticiens[1]. Ainsi, Babbage[2] avait envisagé de programmer sa machine analytique pour jouer aux échecs, Claude Shannon[3] décrivit en 1950 les rudiments d'un programme d'échecs, comme le fit également Alan Mathison Turing. Dès les débuts de l'IA, les premiers chercheurs se lancèrent dans la réalisation de programme de jeux[4]. Ainsi, un des tous premiers programmes de Simon, Newell, Shaw fut un programme de jeux d'échecs et Arthur Samuel écrivit dans les années 50 un programme de jeu de dames américaines.

Pourquoi cette fascination des informaticiens pour le domaine des jeux ? Tout d'abord parce qu'il semble intuitivement (et inconsciemment) que la pratique de jeux intellectuels soit le propre de l'homme. De plus, tout jeu apparaît un peu comme un champ clos de tournoi au Moyen Âge : chacun est astreint à respecter des règles strictes et précises, et ne peut l'emporter que grâce à sa valeur et à son habileté. La victoire et la défaite sont sanctionnées clairement et sans appel possible. Réaliser une machine qui joue aussi bien qu'un homme, voire mieux, « prouverait » l'intelligence (ou la supériorité) des ordinateurs, et serait apparue comme une grande réussite pour l'IA, du moins à ses débuts.

Enfin, comme le dit Donald Michie :

---

1 Voir (Marsland and Schaeffer 1990) : un ouvrage de référence, à la fois sur les techniques de programmation, et sur les relations de la programmation des jeux et particulièrement des échecs avec l'IA.

2 Charles Babbage (1792–1871) peut être considéré comme un des pères des machines programmables. Il définit dès 1833 les principes fondamentaux qui allaient amener à la réalisation des ordinateurs modernes. Protégé et financé par la comtesse Ada Lovelace, fille de Lord Byron, il ne put cependant mener à bien son projet en raison du manque de moyens technologiques de son époque.

3 Claude Shannon est un des pères de la théorie de l'information.

4 Le petit article (Marsland 1990) de Tony Marsland fait le point sur l'histoire de la programmation du jeu d'échecs.

> « La recherche sur le jeu d'échecs est le champ le plus important de la recherche cognitive[5]. Les échecs seront pour nous ce que la drosophile[6] a été pour les généticiens : un moyen simple et pratique de développer de nouvelles techniques. »

Sur le plan technique, le domaine des jeux est très « spécifiable » au sens informatique du terme : les règles sont généralement simples, et surtout les interactions avec le monde extérieur parfaitement nulles : il s'agit d'un *micro-monde* totalement clos, et donc aisément accessible (en apparence) à un programme d'ordinateur. D'autre part, les informaticiens pensaient (plus perfidement) que sa puissance de calcul devrait donner, dans un univers aussi restreint, un avantage considérable à l'ordinateur.

Les choses ne se passèrent pas aussi facilement. Il fallut rapidement distinguer les jeux très simples (morpion) où une analyse exhaustive est possible, des jeux plus complexes comme les dames ou les échecs qui demandent des algorithmes spécifiques et des méthodes heuristiques de recherche, ou encore des jeux à information partielle comme le bridge ou le poker qui demandent en plus des raisonnements de type probabiliste.

Nous allons dans ce chapitre faire un tour d'horizon des techniques de programmation des jeux, et examinerons en détail trois exemples : Othello, les échecs et le bridge. Ces exemples nous permettront d'introduire quelques principes supplémentaires importants de la théorie moderne de la programmation des jeux.

Les joueurs (surtout d'échecs) ayant été des sujets d'études pour les psychologues, nous présenterons quelques-uns des résultats les plus intéressants, résultats qui sont d'autant plus significatifs qu'ils dégagent des facteurs communs à presque toutes les activités humaines.

15.2 Principe minimax (négamax)

Les programmes de jeux à deux joueurs et à information totale utilisent des techniques de représentation semblables à celles que nous avons vues dans le paragraphe précédent : espace d'états et arbres ou graphes. On utilise également des systèmes de production pour générer les états. En revanche, du fait de la présence de deux joueurs ayant des objectifs antagonistes (et non plus identiques), la recherche dans ces arbres[7] ne peut se faire en recourant à des algorithmes du type A^*.

La méthode générale utilisée pour ces jeux est la suivante : à partir d'une position (ou état) donnée, on génère l'ensemble des positions (ou états) que l'ordinateur peut atteindre en jouant un coup (niveau 1 de profondeur). À partir de chacune de ces positions du niveau 1, on génère l'ensemble des positions que l'adversaire peut à

---

5 Il est particulièrement intéressant de remarquer qu'aujourd'hui les meilleurs programmes s'appuient sur des méthodes de force brute et non sur des modèles cognitifs. Sur le problème de la force brute dans la programmation des jeux, voir (Michie 1990).

6 La phrase « Les échecs sont la drosophile de l'Intelligence Artificielle » doit être attribuée, d'après John McCarthy, au physicien soviétique Alexandre Kronrod, qui l'utilisa sans doute pour la première fois en 1966. McCarthy a repris l'expression a son compte dans (McCarthy 1990).

7 Voir (Kaindl 1990) : un excellent article faisant le point sur les méthodes modernes de recherche dans les arbres de jeux.

son tour atteindre (niveau 2). On peut alors recommencer l'opération aussi longtemps que le permet la puissance de calcul de l'ordinateur et générer les niveaux $3, 4, \ldots, n$. On construit ainsi un arbre de l'espace d'états du problème. Il est clair qu'il est impossible, dans la plupart des jeux, de générer l'ensemble de l'espace d'états du problème. Ainsi, aux échecs, le facteur de branchement[8] est environ de 35 à chaque niveau de profondeur. Une partie d'échecs convenablement jouée devant contenir au moins une trentaine de demi-coups, le nombre d'états de l'espace d'états est alors de l'ordre de $35^{30} = 2099139642966190174954314623028039932250976562$: même l'ordinateur le plus puissant du monde ne pourrait le générer en un temps raisonnable (inférieur à l'âge de l'univers par exemple). On doit donc limiter l'exploration à une *profondeur maximale de résolution*. Lorsqu'on a atteint cette profondeur, l'ordinateur attribue à chacune des feuilles une valeur par l'intermédiaire d'une *fonction d'évaluation*. Cette valeur correspond à l'estimation de la position. Cette estimation peut se faire du point de vue d'un joueur fixe (convention minimax) ou du joueur qui a le trait à cette profondeur (convention négamax). Il doit alors, à partir du niveau 0 (la racine), jouer le coup de niveau 1 qui lui garantit le gain maximal *contre toute défense de son adversaire*, en supposant que celui-ci utilise également une stratégie optimale, c'est-à-dire qu'il joue lui-même à chaque coup le gain qui lui garantit le gain maximal contre toute défense. Ce mécanisme est appelé *principe minimax*[9].

Nous allons détailler les deux phases principales de la résolution : la fonction d'évaluation des états terminaux et les algorithmes de recherche.

15.2.1 Fonctions d'évaluation

La fonction d'évaluation tente d'associer à une position une valeur estimant la qualité de la position pour un des deux joueurs. Les fonctions d'évaluation furent étudiées très tôt. Les premières proposées furent évidemment les plus simples. Aux dames par exemple, on peut effectuer la différence entre le nombre de ses pions et le nombre de pions de l'adversaire. Il est clair qu'une fonction heuristique élaborée permet de mieux jouer : si j'inclus dans ma fonction d'évaluation des notions d'attaque et de défense de pions, j'améliore mon niveau de jeu. En revanche, plus la fonction est élaborée et plus il faut de temps pour la calculer, ce qui limitera la profondeur d'exploration de l'arbre. Nous examinerons plus en détail les fonctions d'évaluation en étudiant la programmation des échecs et d'Othello.

15.2.2 Algorithme minimax

L'algorithme minimax[10] est un algorithme de type *exploration en profondeur d'abord* comme nous les avons décrits au chapitre précédent. Il implante le principe minimax que nous avons évoqué. Le jeu comporte deux joueurs : O (l'ordinateur) et H (son adversaire). Une fonction d'évaluation h est chargée d'évaluer la qualité d'une

8 Le facteur de branchement est le nombre de nœuds moyens que l'on peut générer à partir d'une position.

9 On peut trouver une présentation complète de cet algorithme avec bien d'autres dans (Adelson-Velsky *et al.* 1952).

10 Les premières descriptions de cet algorithme remontent à la fin des années 40. Ce sont Turing et Shannon qui, les premiers, ont envisagé ce mécanisme de recherche pour les ordinateurs.

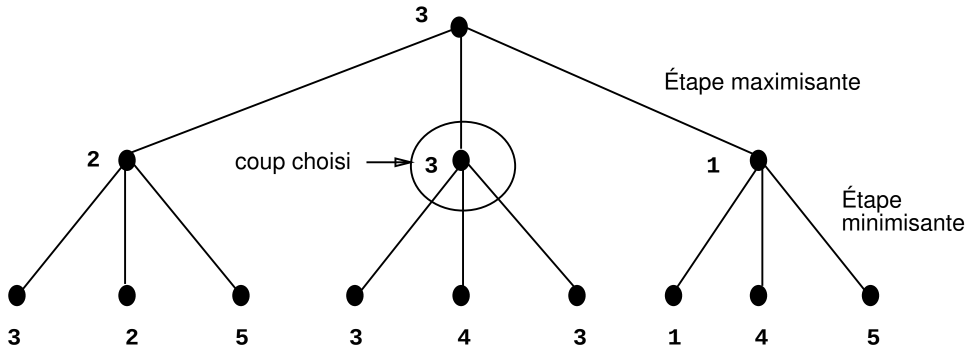

Figure 15.1 – Parcours minimax d'un arbre de jeu

position de jeu terminale (position de profondeur maximale ou sans descendance) du point de vue de O (par exemple). À chaque niveau où O a le trait, il peut choisir son coup et choisira donc celui de valeur maximale pour lui, on parle de nœud de type Max et du joueur Max. À chaque niveau où H a le trait, O va supposer que l'adversaire essaie de le mettre en mauvaise posture (minimiser ses gains) et choisira donc le coup de valeur minimale pour O (nœud de type Min, joueur Min). Ce n'est que sur les feuilles que l'on peut directement appliquer la fonction d'évaluation h. Sinon, on s'appuie sur une écriture récursive. L'algorithme, qui travaille en profondeur d'abord et jusqu'à la profondeur n, ne stocke jamais plus de n positions à la fois. En effet, il commence par générer une branche complète. Il évalue alors la feuille terminale et marque le sommet $n-1$ avec cette valeur. Il repart alors de la position de niveau $n-1$ et génère la feuille suivante (niveau n). Il utilise l'évaluation de cette feuille pour modifier le marquage du niveau $n-1$, et répète l'opération jusqu'à ce qu'il ait généré toutes les feuilles de niveau n issues de cette position de niveau $n-1$. Il utilise alors la valeur de ce niveau $n-1$ pour marquer le niveau $n-2$, remonte à la position du niveau $n-2$ et génère la position de niveau $n-1$ suivante, et continue le processus jusqu'à ce que toutes les feuilles et tous les niveaux aient été générés.

Une convention simplificatrice (du moins pour le programmeur) consiste à considérer qu'on évalue la qualité d'une position non pas du point de vue d'un joueur fixe donné (joueur racine) mais du point de vue du joueur qui a le trait sur cette position. La fonction heuristique doit évidemment être modifiée en conséquence. Dans tous les cas, que ce soit O ou H qui ait le trait, le raisonnement sera le même : pour évaluer le mérite d'un nœud du point de vue du joueur qui a le trait, en connaissant le mérite des positions filles (du point de vue de l'adversaire, qui a le trait sur ces positions), on considère le coup qui maximise (car on veut gagner) l'opposé des mérites des positions filles (car elles ont été évaluées par l'adversaire). Dans tous les cas, la valeur d'un nœud est simplement le maximum des opposés de la valeur des fils. L'écriture de la fonction récursive s'en voit simplifiée. On parle de convention négamax.

Un exemple de parcours par un algorithme minimax est fourni sur la figure 15.1.

On constate que l'algorithme minimax doit complètement décrire l'arbre pour fournir une solution, ce qui, on le verra, est souvent excessif. L'algorithme α-β[11] est une amélioration de l'algorithme minimax qui réalise un élagage de certaines branches qu'il est inutile de visiter.

15.2.3 Algorithme α-β

Examinons l'arbre de la figure 15.2, alors que l'algorithme minimax a déjà parcouru les deux premières branches de la racine, et va s'engager dans la troisième et dernière branche. On sait alors que la racine a une valeur au moins égale à 3 puisqu'un des nœuds fils a déjà été évalué à 3 et que le nœud racine est de type Max. Lorsque, durant l'exploration de la troisième branche, on évalue la première feuille, celle-ci retourne la valeur 1. Son père, qui est un nœud Min, ne va pas pouvoir dépasser la valeur 1, ceci *quelles que puissent être les valeurs des deux fils encore inexplorés*. Or, pour modifier la valeur de la racine, et donc le coup à jouer, il faudrait que ce nœud dépasse la valeur 3, puisque la racine est de type Max. Les deux feuilles restantes sont donc inintéressantes et ne seront pas explorées par α-β.

De façon plus synthétique, lorsque dans le parcours de l'arbre de jeu par minimax il y a remise en cause de la valeur d'un nœud, si cette valeur atteint un certain seuil, il devient inutile d'explorer la descendance encore inexplorée de ce nœud.

Il y a en fait deux seuils, appelés pour des raisons historiques, α (pour les nœuds Min) et β (pour les Max) :

- le seuil α, pour un nœud Min n, est égal à la plus grande valeur (connue) de tous les nœuds Max ancêtres de n. Si n atteint une valeur inférieure à α, l'exploration de sa descendance devient inutile ;
- le seuil β, pour un nœud Max n', est égal à la plus petite valeur (connue) de tous les nœuds Min ancêtres de n'. Si n' atteint une valeur supérieure à β, l'exploration de sa descendance devient inutile.

Ceci nous amène à l'énoncé de l'algorithme α-β qui maintient ces deux valeurs durant le parcours de l'arbre (cf. Algorithme 9). Lors de son utilisation, on l'appellera par Alpha-Béta(*Racine*, $-\infty$, $+\infty$).

Une définition beaucoup plus concise, mais peut-être moins intuitive, peut être obtenue en utilisant la convention négamax ; il devient alors inutile de distinguer les nœuds Min des nœuds Max (cf. Algorithme 10). Cet algorithme calcule alors la valeur négamax de la racine qui est, au signe près, égale à sa valeur minimax.

L'algorithme est d'autant plus efficace que l'on parcourt l'arbre « de la bonne façon ». Quand le niveau est maximisant, il faut examiner d'abord les coups qui ont le plus de chance de générer des positions à forte valeur d'évaluation et réciproquement. Nous verrons dans la suite de ce chapitre comment réaliser de tels mécanismes. Si l'arbre est parcouru exactement dans le bon ordre, le nombre de feuilles examinées est environ $2\sqrt{N}$ où N est le nombre de feuilles total de l'arbre, alors que l'algorithme minimax examine, lui, les N feuilles.

---

11 L'algorithme α-β a été explicitement décrit pour la première fois par Hart et Edwards en 1961.

Algorithme 9: l'α-β (version minimax)

```
;;; n est le sommet dont on veut calculer la valeur;
;;; alpha et β valent initialement −∞ et +∞ respectivement;
Alpha-Béta(n, α, β);
si n est terminal alors retourner h(n);
si n est de type Max alors
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        α ← max(α,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner α;
sinon
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        β ← min(β,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner β;
```

Algorithme 10: l'α-β (version négamax)

```
;;; n est le sommet dont on veut calculer la valeur;
;;; alpha et β valent initialement −∞ et +∞ respectivement;
Alpha-Béta(n, α, β);
si n est terminal alors retourner h(n);
sinon
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        α ← max(α, −Alpha-Béta(f_j, −β, −α));
        j ← (j + 1);
    retourner α;
```

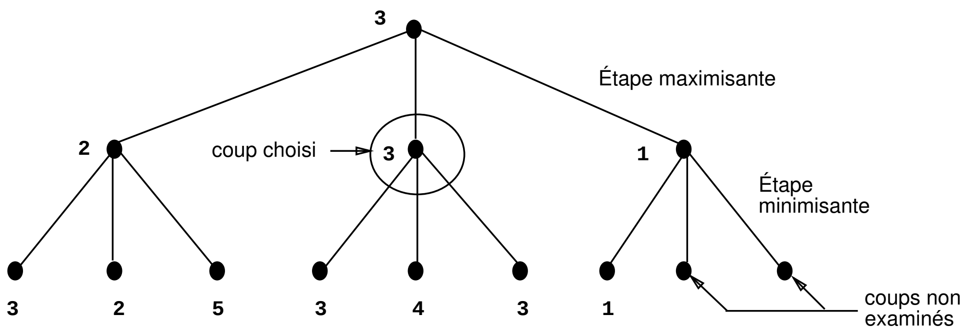

Figure 15.2 – Parcours α-β d'un arbre de jeu

15.3 L'algorithme SSS*

Il s'agit d'un algorithme relativement peu connu, en tous cas nettement moins connu que l'algorithme α-β. Il a pourtant été démontré qu'il lui est théoriquement supérieur[12], dans le sens où il n'évaluera pas un nœud si α-β ne l'examine pas, tout en élaguant éventuellement quelques branches explorées par α-β. Cette qualité supplémentaire se paie, SSS* étant un gros consommateur de mémoire.

Définition 15.1 – Stratégie – *Étant donné un arbre de jeu $\mathcal{J}$, on appelle* stratégie *ou* sous-arbre solution *pour le joueur* Max*, un sous-arbre de $\mathcal{J}$ qui :*
- *contient la racine de $\mathcal{J}$;*
- *dont chaque nœud* Max *a exactement un fils ;*
- *dont chaque nœud* Min *a tous ses fils.*

Une *stratégie* indique au joueur Max ce qu'il doit jouer dans tous les cas. S'il s'agit d'un nœud Max, il a toute liberté de choix et jouera donc le coup indiqué par la stratégie. Si c'est un nœud Min, la stratégie contient tous ses fils et envisage donc toutes les réponses éventuelles de l'adversaire. Si Max respecte une stratégie, il est assuré d'aboutir à une des feuilles de la stratégie. Si l'on se place dans le cas habituel où l'arbre de jeu est développé jusqu'à une profondeur n, le mérite des positions terminales étant estimé par une fonction d'évaluation, la valeur d'une stratégie pour Max est donc le minimum des valeurs des feuilles de cette stratégie, gain assuré contre toute défense de Min. Le but de SSS* est d'exhiber la stratégie de valeur maximum pour Max. Sa valeur sera, par définition, la valeur minimax de l'arbre $\mathcal{J}$.

Définition 15.2 – Stratégie partielle – *Étant donné un arbre de jeu $\mathcal{J}$, on appelle* stratégie partielle *pour le joueur* Max*, un sous-arbre de $\mathcal{J}$ qui :*
- *contient la racine de $\mathcal{J}$;*
- *dont chaque nœud* Max *a au plus un fils.*

Une *stratégie partielle* $\mathcal{P}$ représente implicitement l'ensemble des stratégies complètes auxquelles on peut aboutir en développant $\mathcal{P}$ via :
- l'ajout d'un fils à un nœud Max qui n'en a pas. Pour chaque fils, on aboutit à une nouvelle stratégie partielle ;

12 Lors de l'exploration du même arbre, les feuilles étant ordonnées de façon similaire dans les deux cas.

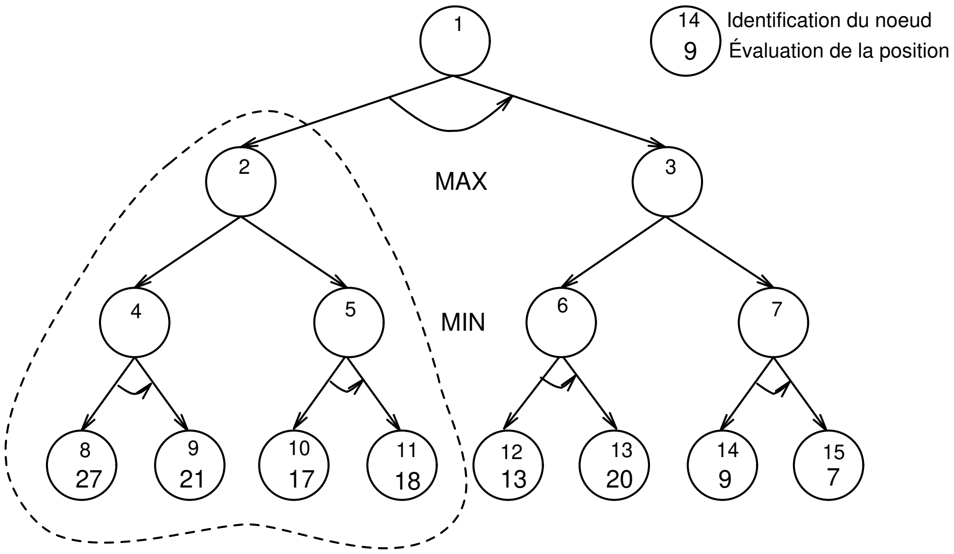

Figure 15.3 – SSS* et arbre de jeu

– l'ajout d'un ou plusieurs fils à un nœud Min, raffinant la stratégie partielle $\mathcal{P}$.

Toutes les stratégies complètes représentées par une stratégie partielle partagent les nœuds de la stratégie partielle, et en particulier, elles partagent les nœuds terminaux sur lesquels on a appliqué la fonction d'évaluation. La valeur d'une stratégie étant égale au minimum de la valeur de ses feuilles, la valeur d'une stratégie partielle constitue une borne supérieure (*i.e.*, une estimation optimiste) de la valeur des stratégies complètes qu'elle représente.

L'algorithme SSS* explore un espace d'état dont chaque nœud est une stratégie partielle, en utilisant une approche *meilleur d'abord* avec une heuristique minorante qui sera la valeur des stratégies partielles et qui garantit l'optimalité. Il ajoute un raffinement permettant d'éviter le développement de certaines stratégies partielles dont on peut démontrer l'absence d'intérêt.

Considérons par exemple l'arbre de jeux de la figure 15.3 et appliquons un *meilleur d'abord* avec une heuristique égale à la valeur des stratégies partielles (valeur de $+\infty$ si la stratégie ne contient pas de nœud terminal). Dans la suite, nous décrirons une stratégie partielle par un couple formé de l'ensemble des nœuds qu'elle contient et de sa valeur estimée. Nous désignerons par G la liste des stratégies partielles.

1. $G = ((\{1\}, +\infty))$. Il y a deux façons de développer la stratégie $(\{1\}, +\infty)$: en ajoutant le nœud 2 ou le nœud 3. Cependant, toutes les stratégies complètes issues de 1 contiennent les nœuds 2 et 3 car le nœud 1 est de type Min. Afin d'éviter d'explorer deux voies menant à la même solution, nous commencerons systématiquement par le premier nœud dans l'ordre lexicographique.
2. $G = ((\{1, 2\}, +\infty))$. Le nœud 2 étant de type Max, le rajout des nœuds 4 ou 5 définit deux stratégies partielles distinctes qu'il nous faut envisager.
3. $G = ((\{1, 2, 4\}, +\infty)(\{1, 2, 5\}, +\infty))$. On développe la première stratégie partielle.

4. $G = ((\{1, 2, 5\}, +\infty)(\{1, 2, 4, 8\}, 27))$. La stratégie $(\{1, 2, 5\}, +\infty)$ est passée en tête.
5. $G = ((\{1, 2, 4, 8\}, 27)(\{1, 2, 5, 10\}, 17))$. Il faut maintenant affiner l'estimation de la stratégie partielle $(\{1, 2, 4, 8\}, 27)$. Les différents nœuds envisageables sont 9 et 3. On choisit 9.
6. $G = ((\{1, 2, 4, 8, 9\}, 21)(\{1, 2, 5, 10\}, 17))$. À ce point du développement, il faut noter que la stratégie complète optimale issue de 2 est *connue*. Il s'agit de $(\{1, 2, 4, 8, 9\}, 21)$. On étend la première stratégie avec 3.
7. $G = ((\{1, 2, 4, 8, 9, 3\}, 21)(\{1, 2, 5, 10\}, 17))$. Le nœud 3 est un nœud Max, il nous faut considérer les deux nœuds 6 et 7.
8. $G = ((\{1, 2, 4, 8, 9, 3, 6\}, 21)(\{1, 2, 4, 8, 9, 3, 7\}, 21)(\{1, 2, 5, 10\}, 17))$. Puis, on développe 12.
9. $G = ((\{1, 2, 4, 8, 9, 3, 7\}, 21)(\{1, 2, 5, 10\}, 17)(\{1, 2, 4, 8, 9, 3, 6, 12\}, 13))$. On passe à 14.
10. $G = ((\{1, 2, 5, 10\}, 17)(\{1, 2, 4, 8, 9, 3, 6, 12\}, 13)(\{1, 2, 4, 8, 9, 3, 7, 14\}, 9))$. La stratégie $(\{1, 2, 5, 10\}, 17)$ est alors en tête. Or, elle contient la sous-stratégie $(2, 5, 10\}, 17)$ dont on sait, d'après la remarque faite au (6) qu'elle est sous-optimale. Il est donc *inutile* de l'explorer plus avant.

L'algorithme du SSS* s'inspire de cette remarque. À partir du moment où une sous-stratégie optimale est établie (comme au point (6)), la stratégie optimale est marquée, et toutes les sous-stratégies sous-optimales supprimées de G. En voici la description précise. Nous noterons :

1. $f_1(n)$: le premier fils du nœud n ;
2. $f_i(n)$: un fils de n ;
3. $\mathrm{fr}(n)$: le frère suivant de n (quand il existe) ;
4. $p(n)$: le père de n.

On dira qu'un nœud est *résolu*, si la stratégie complète issue de ce nœud a été déterminée. Un nœud qui n'est pas résolu est *vivant*. Il ne faut pas confondre par la suite nœud (lié à l'arbre de jeu) et état (lié à l'arbre d'exploration des stratégies).

– Un état sera donc un triplet de la forme *(nœud, type, valeur)* où *nœud* dénote un nœud de l'arbre de jeu, *type* vaut *résolu* ou *vivant* et *valeur* est la valeur estimée de la stratégie partielle que l'on vient d'étendre en ajoutant le *nœud*.
– La liste G des états générés non développés sera, tout comme dans l'algorithme *meilleur d'abord*, triée par ordre décroissant des valeurs. Cette liste sera manipulée par les opérateurs Premier, ExtraitPremier et Inserer qui permettent respectivement d'accéder au meilleur état, d'extraire le meilleur état et d'insérer un nouvel état en maintenant la liste ordonnée[13].
– $h(n)$, retournera la valeur estimée d'un nœud de l'arbre de jeu du point de vue de l'un des deux joueurs.

13 Ces fonctionnalités peuvent être obtenues par une structure de données du type « file de priorité » ou *Heap* (Cormen *et al.* 1990).

Algorithme 11: Le SSS*.

; ; ; n_0 est le nœud à évaluer.

$G \leftarrow ((n_0, \textit{vivant}, +\infty))$;
tant que *Premier(G) n'est pas de la forme* $(n_0, \textit{résolu}, v)$ **faire**
  $(n, t, e) \leftarrow \mathsf{ExtraitPremier}(G)$;
  si $t = \textit{vivant}$ **alors**
    suivant *le type de* n **faire**
1   **cas où** n *est terminal :* $\mathsf{Inserer}((n, \textit{résolu}, \min(e, h(n))), G)$;
      cas où n *est Max :*
        soit $(f_1 \ldots f_k)$ *les fils de* n;
2   **pour** i *allant de* 1 *à* k **faire** $\mathsf{Inserer}((f_i(n), \textit{vivant}, e), G)$;
3   **cas où** n *est Min :* $\mathsf{Inserer}((f_1(n), \textit{vivant}, e), G)$
  sinon
    si n *est de type Min* **alors**
      $\mathsf{Inserer}((p(n), \textit{résolu}, e), G)$;
4   Supprimer de G tous les états dont le nœud est un successeur de $p(n)$;
    else
5   **si** n *a un frère suivant* **alors** $\mathsf{Inserer}((\mathrm{fr}(n), \textit{vivant}, e), G)$;
6   **sinon** $\mathsf{Inserer}((p(n), \textit{résolu}, e), G)$;
retourner v ;

L'algorithme (cf. Algorithme 11) traite le meilleur état courant (n, t, e) selon son type. Si l'état est marqué vivant, cela signifie que la meilleure stratégie issue du nœud correspondant$n$ n'est pas encore encore connue et il faut continuer à la developper :

– Si le nœud est terminal (ligne 1) il suffit d'appliquer l'heuristique et l'état est marqué *résolu*.
– Si le nœud est de type Max, chacun des fils de n défini une stratégie partielle et il faut donc insérer un nouvel état pour chacun d'entre eux (ligne 2) ;
– Si le nœud est de type Min, une stratégie complète devra contenir tous les fils de n. On commence par étendre la stratégie courante avec le premier fils de n (ligne 3). Il faudra ultérieurement considérer ses frères (ligne 5).

Si l'état (n, t, e) est marqué résolu, une stratégie complète optimale issue du nœud est connue et il faut faire remonter l'information :

– Si le nœud est de type Min (fils d'un nœud Max), on est revenu sur un état créé à la ligne 2 et une stratégie optimale pour le nœud père est connue. Il faut effacer tous les stratégies concurrentes issues des frères de n (ligne 4) ;
– Si le nœud est de type Max(fils d'un nœud Min), on est revenu sur une un état créé à la ligne 3. Il faut maintenant étendre la stratégie avec le fils suivant du père de n s'il existe (ligne 5). Sinon, le nœud est résolu car tous les fils ont été examinés (ligne 6).

Comme nous l'annoncions initialement, l'algorithme du SSS* est supérieur à α-β[14]. Récemment, un algorithme paramétrable sur la quantité de mémoire (longueur maximum de la liste G) qu'il peut utiliser — l'IterSSS* — a été exhibé (Bhattacharya and Bagchi 1986). Il permet d'obtenir des performances variant entre celles d'α-β (mémoire minimum) et de SSS* (mémoire suffisamment importante). Le tableau suivant indique les performances de cet algorithme en pourcentage de nœuds visités sur différents arbres uniformes de facteurs de branchements et de profondeurs variables :

Profondeur	Branchement	α-β	SSS*	longueur G	IterSSS*
15	2	12,47	8,5	9	12,4
				64	10,36
				192	9,37
				256	8,5
6	5	16,51	11,48	13	15,49
				63	12,68
				95	12,6
				125	11,48
10	3	10,44	6,44	11	10,11
				122	7,75
				243	6,44

15.4 L'algorithme Scout

Nous décrirons rapidement l'algorithme Scout, élaboré par J. Pearl (Pearl 1990) comme outil théorique. Son efficacité est, en général, inférieure à celle d'α-β, pour une consommation mémoire du même ordre. Il peut toutefois lui être supérieur.

Scout repose sur une idée fort simple : si l'on disposait d'un moyen efficace pour comparer (sans nécessairement la déterminer) la valeur minimax d'un nœud à une valeur donnée, une quantité importante de recherche pourrait être évitée. Considérons par exemple un nœud Max n, ayant deux fils : f_1, dont la valeur ν est connue, et f_2. Si l'on sait que la valeur de f_2 est inférieure à ν, il est inutile d'explorer la branche de f_2.

Scout s'appuie donc sur deux procédures simples : la première, appelée Test permet de vérifier si la valeur d'un nœud n est strictement supérieure (ou supérieure ou égale) à une valeur donnée v. Nous continuerons à désigner par h la fonction heuristique.

La seconde, Eval(), utilise Test et applique le principe donné plus haut pour calculer la valeur minimax d'un arbre de jeu. Elle prend en paramètre un nœud n.

Il pourrait sembler que, du fait de la redondance éventuelle des évaluations, lorsque Test ne permet pas la coupure, Scout devrait être très inférieur à α-β, voire

14 Il faut pour cela que les deux algorithmes développent le même arbre, dans un ordre comparable. Ceci nécessite, dans le cas de l'insertion dans G de deux nœuds de même valeur, de mettre en première position le nœud exploré en premier par α-β et entraîne l'emploi de finesse de programmation (ordonnancement fixe des nœuds...) qu'il est inutile de pratiquer lorsqu'on désire employer SSS* pour autre chose qu'une comparaison à α-β.

Procedure Test(n, v, c)

; ; ; n est le nœud à évaluer;
; ; ; v la valeur seuil;
; ; ; c le prédicat fonctionnel de comparaison ($>$ ou $\geq$);

si n *est un nœud terminal* **alors retourner** $c(h(n), v)$;
si n *est Max* **alors**
  soit $S = \{s_1 \ldots s_i\}$ *la liste des fils de* n*;*
  pour j *allant de* 1 *à* i **faire**
    si *Test(*s_j, v, c*)* **alors retourner** *vrai*;
  retourner *faux*;
sinon
  soit $S = \{s_1 \ldots s_i\}$ *la liste des fils de* n*;*
  pour j *allant de* 1 *à* i **faire**
    si $\neg$ *Test(*s_j, v, c*)* **alors retourner** *faux*;
  retourner *vrai*;

Procedure Eval(n)

; ; ; n est le nœud à évaluer;

si n *est un neud terminal* **alors retourner** $h(n)$;
soit $S = \{s_1 \ldots s_i\}$ *la liste des fils de* n*;*
soit $v =$ *Eval(*(s_1)*);*
pour j *allant de* 2 *à* i **faire**
  si s_j *est Max* **alors**
    si *Test(*$s_j, v, >$*)* **alors** $v \leftarrow$ Eval(s_j);
  sinon
    si $\neg$ *Test(*$s_j, v, \geq$*)* **alors** $v \leftarrow$ Eval(s_j);
retourner v;

même à minimax. Une étude mathématique du comportement asymptotique de Scout montre un comportement identique à α-β pour des profondeurs élevées. Dans le cas de certains jeux (jeu de Kalah), il peut même lui être supérieur (jusqu'à 40% d'amélioration à profondeur 5). Bien qu'aucune preuve ne vienne l'appuyer, il semble que Scout soit plus particulièrement adapté aux arbres profonds, ayant un facteur de branchement faible[15].

Le tableau suivant reprend l'ensemble des algorithmes de jeu présentés (minimax, α-β, SSS*, Scout) et indique les performances relatives de chacun d'eux,
– en nombre de feuilles explorées ;

15 Le lecteur intéressé pourra l'appliquer à l'Awale, proche du jeu de Kalah, jeu africain dont le facteur de branchement est inférieur à 6.

– et en temps machine consommé[16].
sur le jeu de l'Othello pour différentes profondeurs p.

p	minimax	Scout	α-β	SSS*
1	3 – 0,37	4 – 0,45	3 – 0,36	3 – 0,36
2	14 – 1,38	21 – 2.09	13 – 1,4	11 – 1,19
3	61 – 6,0	45 – 5,0	37 – 3,9	35 – 3,7
4	349 – 32,5	246 – 23,7	150 – 14,77	95 – 10,0
5	2050 – 185,7	615 – 61,6	418 – 41,4	292 – 19,2
6	13773 – 1213,5	2680 – 254,5	1830 – 172,9	1617 – 117,3

On notera les mauvaises performances de Scout à faible profondeur (plus mauvais que le simple minimax) et la nette supériorité de SSS*. Il faut cependant noter que nous n'avons pu poursuivre les tests avec SSS* au-delà de la profondeur 6 pour des raisons de place mémoire.

15.5 La pratique après la théorie

Les différents algorithmes et résultats présentés ci-dessus sont parfaitement intéressants mais ne sont plus utilisés aujourd'hui sous cette forme. De nombreuses améliorations ont été apportées et il est indispensable de les connaitre pour espérer écrire un programme performant.

15.5.1 Contrôle du temps et α-β

On doit dans la pratique s'intéresser au système de contrôle du temps. Le programme dispose d'un temps total pour jouer l'ensemble de la partie. Il serait possible d'implanter deux techniques pour que l'ordinateur ne consomme pas trop de temps sur un coup. On pourrait tout d'abord accorder un temps donné pour jouer le coup et estimer la profondeur maximale que l'on peut atteindre sans dépasser ce temps. Cette solution présente un inconvénient : il est très difficile de trouver une fonction qui estime le temps que prendra un calcul à une profondeur donnée. On risque de se retrouver en dépassement de temps. On préfère donc utiliser une méthode un peu différente. On fait fonctionner l'algorithme α-β à un niveau minimal (3 par exemple). On compare le temps utilisé avec le temps disponible. S'il reste du temps, on relance la recherche au niveau 4, puis au niveau 5… Cette méthode (habituellement référencée sous le nom de *Depth First Iterative Deepening* (Korf 1985; Slate and Atkin 1978; Marsland and Schaeffer 1990)) présente un autre avantage. On constate aisément qu'un algorithme α-β est d'autant plus efficace qu'il évalue en premier les meilleurs coups. On profite donc du résultat de la recherche à la profondeur n pour classer les coups et faire examiner en premier au programme les meilleurs coups de la profondeur n lorsqu'il effectue la recherche en profondeur $n + 1$.

16 Algorithmes tous réalisés en LE_LISP, utilisant la même fonction heuristique, les mêmes fonctions d'exploration de l'arbre de jeu. Les temps sont sans unité.

position	évaluation	meilleur coup	distance à la position terminale

Figure 15.4 – Informations stockées dans une table de transposition

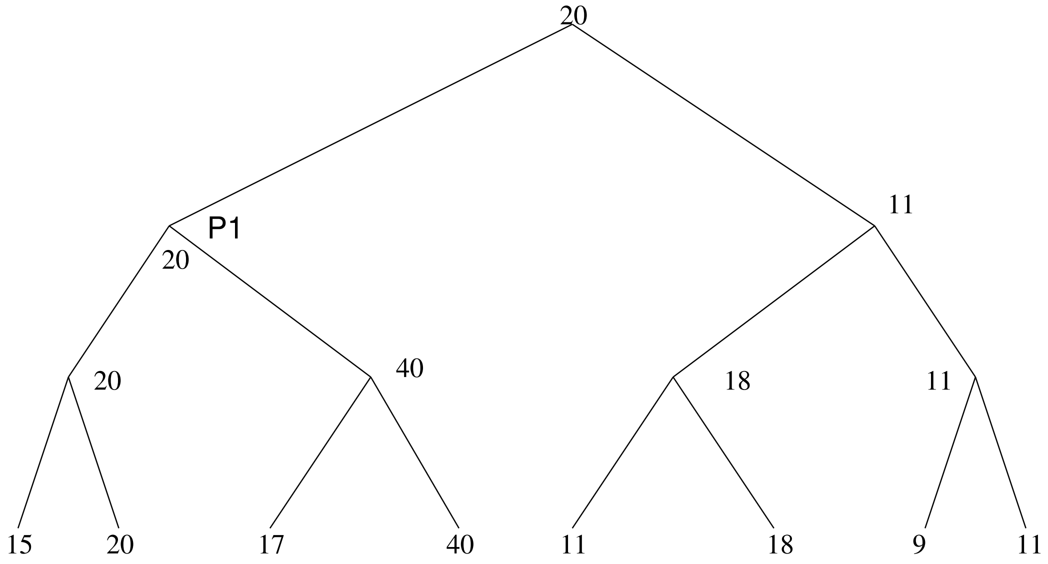

Figure 15.5 – Exemple d'arbre minimax

15.5.2 Tables de transposition

Le principe des tables de transposition est le suivant : pour chaque position rencontrée dans la recherche, l'algorithme minimax retourne une évaluation. Il s'agit de conserver dans une table la valeur de chacune des positions rencontrées de façon à pouvoir réutiliser directement cette valeur lors des recherches suivantes. Les informations stockées pour une position sont représentées sur la figure 15.4. Nous allons voir cela sur un exemple pratique. Soit l'arbre minimax représenté sur la figure 15.5. Les informations stockées pour la position P_1 sont représentées figure 15.6. Supposons que dans une recherche future l'ordinateur rencontre à nouveau la position P_1. Si la distance à la racine de P_1 est plus grande dans la table de transposition que dans la recherche actuelle, l'ordinateur ne poursuivra pas l'évaluation : il se contentera de recopier la valeur stockée dans la table de transposition. En revanche, si la distance à la racine est plus petite dans la table de transposition que dans l'évaluation courante, le programme terminera normalement l'évaluation de la position et remplacera

position	évaluation	meilleur coup	distance à la position terminale
P_1	20	Gauche	2

Figure 15.6 – Valeurs stockées pour la position P_1 de l'arbre minimax représenté sur la figure précédente

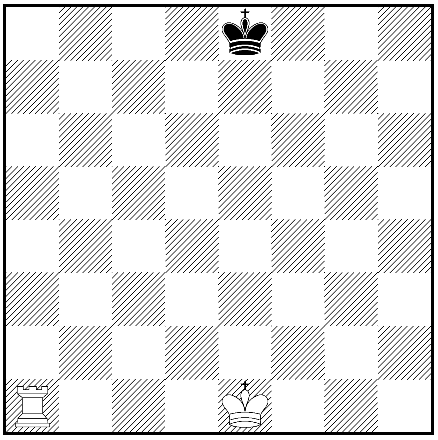

Figure 15.7 – Position

Pièce	couleur	position	valeur de hash-coding (choisie aléatoirement)
Tour	blanc	a1	10101011110110110101010110101101
Tour	blanc	a2	01000101001011001110100111101010
...	...	...	...
Roi	blanc	e1	11010110101001001010001011010110
...	...	...	...
Roi	noir	e8	00100101010110110100100010101000
...	...	...	...

Table 15.1 – Valeurs de hash-coding

l'ancienne évaluation de la table de transposition par celle qu'il aura calculée, puis stockera la nouvelle valeur de la distance à la racine.

Si les tables de transposition fonctionnaient suivant ce principe, un problème se poserait rapidement : comment stocker tous les éléments de la table de transposition (et en particulier la position de chacune des pièces sur l'échiquier) dans la mémoire du calculateur, et comment retrouver rapidement une position. On utilise une structure de données connue sous le nom de *table de hachage*[17] ; au lieu d'utiliser la position en entier pour retrouver la valeur de cette position, on utilise un nombre qui caractérise cette position. Nous allons examiner brièvement sur un exemple une méthode permettant de construire ce nombre pour une position aux échecs.

On construit une table contenant, pour chaque pièce de chaque couleur sur chacune des cases de l'échiquier, un nombre binaire de 32 chiffres choisi de façon aléatoire (voir exemple table 15.1). Pour une position donnée, on construit alors le code de hachage en faisant un « ou-exclusif[18] » entre chacune des valeurs associée à chacune des pièces. Ainsi, pour la position représentée sur la figure 15.7, la valeur sera :

17 En anglais, *hash table*. Voir par exemple (Cormen *et al.* 1990).

18 L'addition de nombres binaires sans retenue (qui correspond à un *ou exclusif* bit à bit) consiste à additionner chaque chiffre en utilisant comme règle : $0 \oplus 0 = 0$, $0 \oplus 1 = 1$ et $1 \oplus 1 = 0$.

Tour blanche en a1	10101011110110110101010110101101	$\oplus$
Roi blanc en e1	11010110101001001010001011010110	$\oplus$
Roi noir en e8	00100101010110110100100010101000	$=$
	01011000001001001011111111010011	

Le grand intérêt de cette méthode est qu'elle permet de calculer la valeur de hachage d'une position après déplacement d'une pièce, simplement en ajoutant à la valeur de hachage de la position originale le code de la pièce sur la nouvelle case et le code de la pièce sur l'ancienne case. C'est une propriété du « ou-exclusif ».

Un des problèmes des tables de hachage est que rien ne nous garantit que deux positions différentes ne vont pas avoir la même valeur de hachage, ce qui risquerait de nous amener à des conclusions fausses sur la valeur d'une position. Cette probabilité de collision est d'autant plus faible que la taille de la table est importante.

Il reste maintenant à stocker un élément de la table de hachage. On ne peut utiliser la valeur de hachage comme index de tableau car elle est généralement trop grande (un tableau avec un index sur 32 bits occuperait quatre milliards de positions mémoire, ce qui est déraisonnable). On peut par exemple utiliser les 16 premiers bits de la valeur de hachage comme index du tableau. On prend alors le risque que des positions aient des valeurs de hachage différentes et le même index. Il y a alors *conflit*. Il existe différentes méthodes pour gérer ce type de problème, la plus simple étant de remplacer la plus ancienne des deux valeurs par la plus récente[19].

15.5.3 α-β avec mémoire

La méthode des tables de transposition permet d'améliorer de façon spectaculaire la vitesse de recherche d'un calculateur, car elles permettent l'implantation de la version la plus classiquement utilisée de l'α-β, l'α-β avec mémoire. Sa structure est un petit peu plus complexe que celle de l'α-β standard, car il fait appel aux tables de transposition pour stocker les bornes supérieures et inférieurs des positions déjà évaluées. Nous le présentons ici dans sa version la plus aisée à comprendre (algorithme 14).

On peut aisément se demander quel peut-être l'intérêt d'un tel algorithme. Il est de deux ordres. D'une part, dans les jeux où certaines positions se présentent plusieurs fois dans l'arbre, il permet de ne pas recalculer le nœud comme indiqué précédemment. Mais son principal avantage apparait lorsqu'on l'utilise avec des fenêtres de recherche réduite, en particulier avec les algorithmes de la famille MTD(f).

15.5.4 Fenêtre réduite et MTD(f)

On se rappelle que lors du début de l'algorithme α-β, les bornes α et β sont initialisées à $+\infty$ et $-\infty$. Or, dans 90% des positions, il est rare que la valeur d'une position évolue beaucoup. On peut donc initialiser les bornes α et β à la valeur de la position courante $\pm\epsilon$ (ou ϵ est une valeur bien choisie dépendant du jeu et/ou de la fonction d'évaluation). Ainsi, tous les coups dans l'algorithme α-β qui seront en dehors de cette fenêtre seront immédiatement élagués. La méthode est efficace si l'hypothèse

19 On appelle cette structure *hash cache table* en anglais. Notons qu'il existe des stratégies plus évoluées.

Algorithme 14: l'α-β (version minimax avec mémoire)

Alpha-Béta(n, α, β);
si n *est dans la table* **alors**
  si $n.bas >= \beta$ **alors retourner** $n.bas$;
  si $n.haut <= \alpha$ **alors retourner** $n.haut$;
  $\alpha \leftarrow \max(\alpha, n.bas)$;
  $\beta \leftarrow \min(\beta, n.haut)$;
si n *est terminal* **alors retourner** $h(n)$;
si n *est de type Max* **alors**
  soit $g \leftarrow -\infty$;
  soit $j \leftarrow 1$;
  soit $(f_1 \dots f_k)$ *les fils de* n;
  tant que *($j \leq k$)* **faire**
    $g \leftarrow \max(g,$Alpha-Béta$(f_j, \alpha, \beta))$;
    si $g >= \beta$ **alors** Sortir;
    $\alpha \leftarrow \max(\alpha, g)$;
    $j \leftarrow (j + 1)$;
sinon
  soit $g \leftarrow +\infty$;
  soit $j \leftarrow 1$;
  soit $(f_1 \dots f_k)$ *les fils de* n;
  tant que *($j \leq k$)* **faire**
    $g \leftarrow \min(g,$Alpha-Béta$(f_j, \alpha, \beta))$;
    si $g <= \alpha$ **alors** Sortir;
    $\beta \leftarrow \min(\beta, g)$;
    $j \leftarrow (j + 1)$;
si $g <= \alpha$ **alors** $n.haut \leftarrow g$;
si $g >= \beta$ **alors** $n.bas \leftarrow g$;
si $g > \alpha$ *et* $g < \beta$ **alors**
  $n.bas \leftarrow g$;
  $n.haut \leftarrow g$;
retourner g

initiale est juste (c'est-à-dire qu'on ne peut effectivement pas gagner ou perdre plus de ϵ). Si tel n'est pas le cas, on le détecte aisément car la valeur retournée est alors hors de la fenêtre initiale $]\alpha\ ,\ \beta[$. Il faut alors relancer l'algorithme en modifiant la fenêtre de recherche ; mais si l'on s'est donné la peine de stocker dans la table de transposition pour chaque position déjà exploré les informations acquises lors de la passe précédente on gagne un temps précieux en ne réexaminant pas certains nœuds.

C'est un poussant ce raisonnement à l'extrème, et en s'inspirant de l'algorithme Scout, qu'est né l'algorithme MTD(f) (algorithme 15). Cet algorithme présente de

Algorithme 15: MTD(f)

```
;;; root est la racine de l'arbre;
MTD(root, f);
soit g ← f;
soit h ← +∞;
soit b ← −∞;
tant que h > b faire
    si g = b alors β ← g + 1 sinon β ← g;
    g ←Alpha-Béta(root, β − 1, β);
    si g < β alors h ← g sinon b ← g
retourner g
```

nombreuses particularités qui le rendent particulièrement intéressant. Il emprunte en effet certains traits à la plupart des algorithmes que nous avons déjà rencontrés. Son principe est simple : il appelle autant de fois que nécessaire un α-β pour construire un encadrement de la valeur de la position. Quand les deux bornes haute et basse se rencontrent, l'évaluation est terminée et la valeur de la position est connue. L'efficacité de l'algorithme vient de l'élagage violent qu'il effectue à chaque évaluation, car l'α-β avec mémoire est appelé avec une fenêtre de taille 0. Il est d'autant plus efficace que la valeur du paramètre f dont dépend MTD(f) est proche de la valeur réelle de la position. On utilise donc en général comme valeur de f la valeur retourné par l'algorithme lors d'une itération précédente. Remarquons que la technique consistant à utiliser un α-β avec une fenêtre de taille 0 est équivalent à la procédure TEST de l'algorithme SCOUT.

De récents travaux (Plaat *et al.* 1996) ont montré d'autre part que MTD($+\infty$) était exactement équivalent à SSS^*, c'est à dire qu'il développe les mêmes nœuds dans le même ordre, répondant en cela aux nombreuses questions qui se posaient depuis l'article fondateur de SSS^*. On en déduit en particulier que SSS^* est moins efficace que MTD(f) pour une valeur de f bien choisie.

15.5.5 Paralléliser un algorithme α-β

Comme pour tous les programmes utilisant des techniques de recherche dans des arbres, la puissance de calcul est un facteur déterminant. Un moyen traditionnel d'augmenter cette puissance consiste à tenter de rendre l'exécution du programme parallèle, c'est-à-dire d'utiliser plusieurs machines (ou plusieurs processeurs) pour exécuter simultanément des parties du programme.

Il est clair que l'on peut aisément paralléliser un algorithme α-β. En effet, il suffit de distribuer la recherche dans l'arbre sur l'ensemble des processeurs. Mais on ne peut pas le faire n'importe comment. On doit choisir entre quatre types de technique (nous retrouverons ce type de mécanisme lorsque nous parlerons de parallélisme en programmation logique) :

Système distribué contre système en étoile : Dans un système en étoile[20] un des processeurs, le processeur maître, est chargé de diriger la résolution. Ainsi, dans le cas qui nous intéresse, le processeur maître envoie à chacune des machines esclaves une position et lui demande de renvoyer sa valuation α-β. Il peut, s'il le désire, interrompre le travail d'un processeur s'il estime ce travail inutile, à la suite d'informations qu'il aurait reçues d'autres processeurs. Ce type de mécanisme est simple à mettre en œuvre, car le contrôle de l'exécution reste séquentiel, toutes les des décisions étant prises par un seul processeur. D'autre part, le nombre d'informations transitant entre les processeurs reste relativement faible.

Dans un modèle totalement distribué, aucun processeur n'a de statut particulier. Chaque processeur choisit au départ une position à évaluer (au hasard et après un délai aléatoire différent de façon à limiter les conflits) et diffuse sur le réseau à l'intention des autres son choix. Au cours de la résolution, il diffuse également, à l'intention de tous les autres processeurs, l'ensemble des informations qui lui paraissent intéressantes (valuations α-β de position par exemple). Ce type de modèle est plus fiable dans la mesure où la défaillance (ou l'absence) d'une machine est plus facilement surmontée. En revanche, il est beaucoup plus complexe à réaliser. En particulier, la gestion des conflits doit être faite avec soin : comment faire si deux processeurs se rendent compte qu'ils ont choisi la même branche ? Une méthode envisageable consiste à faire attendre chaque machine pendant une durée aléatoire, avant de leur faire choisir à nouveau une branche à évaluer. Enfin, un autre inconvénient est l'augmentation du débit des données qui va circuler sur le réseau de communications entre processeurs.

Grosse ou petite granularité : [21] Lorsque l'on distribue la résolution dans l'algorithme α-β, on doit choisir le niveau à partir duquel chaque processeur va commencer une évaluation α-β séquentielle classique. Si nous choisissons de faire commencer l'évaluation au niveau 1 de l'arbre (grosse granularité), nous ne pourrons pas utiliser plus de processeurs qu'il n'y a de positions différentes au niveau 1 (facteur de branchement). Ainsi, pour Othello, il y a en moyenne 16 coups jouables à chaque tour. Si nous disposons de n processeurs avec $n > 16$, nous aurons en moyenne $n - 16$ processeurs inemployés. Si nous distribuons la recherche à partir du niveau 2 (256 positions), nous serons en mesure d'occuper simultanément 256 processeurs. En revanche, le nombre d'informations qui cir-

20 Appelé aussi modèle maître-esclaves.

21 Le terme de granularité semble adapté à la représentation intuitive que l'on se fait de la parallélisation d'une tâche. On peut considérer la tâche comme un bloc monolithique. La paralléliser va consister à la réduire en sous-tâches plus petites. Si nous découpons la tâche en gros blocs, on parlera de grosse granularité (peu de sous-tâches, et chaque sous-tâche est importante). Si nous décomposons la tâche en de multiples petites sous-tâches, on parlera de petite granularité.

culeront sur le réseau sera beaucoup plus important et continuera à augmenter au fur et à mesure que nous rendrons la granularité plus petite.

Il faut donc choisir sa stratégie en fonction des capacités des machines et du réseau dont on dispose. Examinons en détail le fonctionnement d'un système à forte granularité :

1. Au départ, la machine maître est la seule à connaître la position à évaluer (niveau 0). À partir de cette position, elle établit les 15 premières positions que l'on peut générer au niveau 1 et demande à chacun des processeurs esclaves de calculer la valuation α-β pour une de ces positions.
2. Chaque processeur esclave va, au fur et à mesure que l'évaluation de la position que lui a confiée la machine progresse, renvoyer à la machine maître les évaluations partielles au niveau 1. La machine maître note chacune des évaluations partielles pour chacun des processeurs, en sachant qu'une évaluation partielle de niveau 1 ne peut que diminuer (principe minimax).
3. À un instant donné, une des machines esclaves termine la résolution de sa position et renvoie à la machine maître l'évaluation définitive de cette position de niveau 1. La machine maître effectue alors la séquence d'opérations suivante :
   (a) Elle note ce coup comme le meilleur connu et cette évaluation comme la meilleure évaluation courante et l'appelle p_0.
   (b) Elle note que ce processeur est disponible.
   (c) Elle ordonne à tous les processeurs dont l'évaluation partielle de niveau 1 est inférieure à p_0 de s'arrêter : en effet, ils ne pourront trouver que des valeurs inférieures à leur valeur courante (principe α-β-minimax). Elle note que tous ces processeurs sont disponibles.
   (d) Elle répartit sur les processeurs disponibles les positions de niveau 1 encore non traitées (la $16^{\text{ième}}$, $17^{\text{ième}}$…).
4. Chaque fois qu'une machine lui communiquera une évaluation partielle inférieure à l'évaluation p_0 elle lui communique l'ordre de s'arrêter et lui donne à évaluer une des positions encore disponibles.
5. Chaque fois qu'une machine lui envoie une évaluation définitive de niveau 1, et que cette évaluation est supérieure à p_0, la machine maître répète les opérations ((a)… (d)).
6. La machine maître poursuit cet algorithme jusqu'à ce que tous les processeurs esclaves soient arrêtés et que toutes les positions de niveau 1 aient été examinées.

Nous avons ainsi décrit une implantation parallèle de l'algorithme α-β[22].

22 Il existe d'autres implantations parallèles de l'algorithme α-β beaucoup plus efficaces. Le problème principal de cette implantation est que dans une résolution par un mécanisme α-β, la réponse au premier coup prend généralement 50% du temps de résolution. Donc répartir également chacun des coups du premier niveau est clairement une erreur. L'algorithme PVSA (Principal Variation Splitting Algorithm) améliore le système en parallélisant les réponses pour le premier coup à chaque niveau de l'arbre. On peut se reporter à (AKL *et al.* 1982; Schaeffer 1989a; Marsland and Schaeffer 1990; Kuszmaul 1995) pour des méthodes encore plus efficaces.

15.5.6 Pour conclure ?

La recherche dans le domaine des algorithmes de jeu est d'une richesse et d'une vigueur assez extraordinaire et nous n'avons fait qu'effleuré le domaine. Nous n'avons en particulier pas parlé des alternatives au paradigme minimax, comme les algorithmes de type B^* ou BPIP (Best Play for Imperfect Player). On peut se reporter avec profit à l'excellente thèse de Jean-Christophe Weill, qui a de plus le grand mérite d'être en Français (Weill 1995).

15.6 Othello

Nous nous proposons de développer en détail les différentes phases de la programmation du jeu d'Othello. La programmation d'Othello[23] doit être divisée en trois phases : l'ouverture, le milieu de partie et la fin de partie. Nous allons nous intéresser successivement à ces trois différents types de jeu.

15.6.1 Les ouvertures

Il existe dans le jeu d'Othello un certain nombre d'ouvertures répertoriées. Il s'agit de séquences connues que l'on doit systématiquement utiliser en début de partie, sous peine de se retrouver en position inférieure. Le jeu d'Othello étant beaucoup plus jeune que le jeu d'échecs, la théorie des ouvertures est beaucoup moins développée. On ne s'intéresse à l'heure actuelle qu'à des séquences allant jusqu'à six coups maximum. D'autre part, les séquences recommandées ne sont souvent choisies que pour des raisons statistiques (on sait que, dans un championnat du monde donné, les meilleurs joueurs ont eu leurs meilleurs résultats avec une certaine séquence : c'est donc qu'il s'agit d'une bonne ouverture).

Les ouvertures ne sont pas stockées sous forme de séquence de coups, mais sous forme de positions indexées. Cela signifie que l'ordinateur ne stocke pas des séquences de positions, chaque séquence correspondant à une ouverture, mais stocke séparément chacune des positions avec le coup à jouer si cette position se rencontre à un instant donné. Pourquoi cette méthode ? Simplement pour éviter que l'ordinateur ne se perde dans une interversion de coups. Afin de retrouver rapidement une position dans l'ensemble des positions stockées, on les indexe en général par un certain nombre de clés (nombre de pions de chaque couleur…).

Cette technique a été amélioré considérablement par Michael Buro pour son programme LOGISTELLO, qui est actuellement le meilleur programme mondial. Buro utilise des techniques tout à fait originales pour améliorer en permanence la bibliothèque de son programme. On pourra se reporter avec profit à (Buro 1997) pour plus de détails.

---

23 On peut se demander pourquoi présenter Othello. Plusieurs raisons à cela : c'est un des jeux où les ordinateurs ont obtenu les performances les plus probantes ; un des auteurs de cet ouvrage est également l'auteur du programme d'Othello présenté dans ces quelques lignes, (programme qui battit il y a quelques années un bon joueur français et reste cependant assez loin des tous meilleurs programmes), et connaît donc assez bien les techniques utilisées ; enfin, c'est un jeu relativement connu et dont les règles se comprennent rapidement.

15.6.2 Les finales

Le jeu d'Othello est intéressant dans la mesure où il s'agit d'un jeu à information totale à *nombre de coups précisément connus*. On sait en effet que la partie se terminera au plus tard (et en général exactement) au $60^{\text{ième}}$ demi-coup[24]. En fonction de la puissance de calcul disponible, il est possible de lancer une recherche exhaustive un certain nombre de demi-coups avant la fin (le $60^{\text{ième}}$ coup). Dans ce cas, la fonction d'évaluation est extrêmement simple : il suffit de faire la différence entre son propre nombre de pions et le nombre de pions de l'adversaire. De plus, il s'agit d'une heuristique qui évalue parfaitement le caractère perdant ou gagnant de la position (évidemment !). L'algorithme de recherche est relativement original, puisque l'on considère que le plus efficace dans cet exercice est le $NegaC^*$, un algorithme proche de MTD(f) mais pour lequel on recherche l'encadrement de la valeur de la position par bisection successive de l'intervalle (Weill 1995).

Cette caractéristique du jeu d'Othello explique en partie la force des programmes d'ordinateur : un humain est totalement incapable d'exécuter un calcul total sur une semblable profondeur sans commettre d'erreurs. Ainsi, un ordinateur jouant à Othello peut se retrouver dans une situation difficile une vingtaine de coups avant la fin de la partie et retourner (au propre et au figuré) complètement la position en sa faveur simplement par la force brute[25].

15.6.3 Le milieu de partie

Le milieu de partie est le royaume de l'algorithme α-β et de la fonction d'évaluation. Nous allons examiner en détail la construction d'une fonction d'évaluation trés simple pour Othello. Cette fonction sera une fonction statique dans la mesure où le programme n'est pas capable d'apprentissage. Nous allons nous apercevoir que la construction de la fonction n'est pas chose complètement triviale, même si le jeu est simple et le but raisonnable (il ne s'agit pas de construire un programme champion du monde).

Tout d'abord, on tente d'attribuer à chaque case du jeu une valeur tactique. Cette valeur tactique doit représenter l'intérêt qu'a l'ordinateur à occuper une case ou, au contraire, à la laisser à son adversaire. Une fonction d'évaluation élémentaire consisterait donc à additionner les valeurs des cases que l'on possède et à soustraire la valeur des cases que possède l'adversaire. Une possibilité de valuation des cases est donnée dans la figure 15.8.

On peut rapidement justifier ce tableau en disant que la possession d'un coin est capitale, alors que les cases qui entourent le coin sont à éviter car elles donnent à l'adversaire accès au coin. La possession des cases centrales est importante, car elle augmente généralement les possibilités de jeu. Les bords sont également des points d'appui solides.

Une fois cette évaluation effectuée, il faut la corriger sur-le-champ. En effet, si nous possédons un coin, la valeur des trois cases environnantes doit être considérée

24 À distinguer des échecs où une partie peut durer moins de 10 coups ou plus de 100.
25 On est loin des principes cognitifs fondateurs de l'IA !

500	-150	30	10	10	30	-150	500
-150	-250	0	0	0	0	-250	-150
30	0	1	2	2	1	0	30
10	0	2	16	16	2	0	10
10	0	2	16	16	2	0	10
30	0	1	2	2	1	0	30
-150	-250	0	0	0	0	-250	-150
500	-150	30	10	10	30	-150	500

Figure 15.8 – Exemple de valuation de cases pour Othello

comme positive puisqu'elles ne donnent plus accès au coin, qui est déjà occupé. De même, les cases du bord qui sont reliés au coin par une chaîne continue de pions de la même couleur sont beaucoup plus intéressantes car désormais imprenables. Un autre facteur important de la position à Othello est le nombre de cases libres contiguës à ses propres pions. Il faut essayer de le minimiser car plus ce nombre est petit et moins l'adversaire a, en général, de coups disponibles.

15.6.4 Othello et apprentissage

Le programme que nous venons de présenter garantit un niveau de jeu intéressant, mais est cependant très loin des meilleurs programmes d'Othello. Tous les meilleurs programmes utilisent aujourd'hui des techniques d'apprentissage qui peuvent être de plusieurs sortes. Ce type de mécanisme n'est pas nouveau, puisqu'il était déjà utilisé par Arthur Samuel (Samuel 1959) dans son programme de jeu de dames.

La plupart des fonctions d'évaluation sont de la forme $\Sigma c_i t_i$ où les t_i sont les caractéristiques de la position et les c_i les coefficients qui mesurent l'importance à accorder à telle ou telle caractéristique[26]. Au fur et à mesure de l'apprentissage, les coefficients c_i doivent être modifiés de façon à enregistrer l'information acquise pendant les parties.

Comment modifier les coefficients ? Il est clair que l'on doit augmenter la valeur des coefficients qui prédisent correctement l'issue du coup et diminuer la valeur de ceux qui fournissent des informations incorrectes. Le problème dans un programme de jeu est qu'il faut être capable, après la partie, d'analyser les coups qui se sont révélés bons ou mauvais, de façon à modifier la fonction d'évaluation. On peut implanter ce mécanisme en supposant que toute séquence de coups qui a amené à une bonne situation est une bonne séquence et que les coefficients qui ont encouragé cette séquence doivent être renforcés et ceux qui tendaient à l'éviter, diminués (méthode de la carotte et du bâton). Quant à mesurer en quoi une situation est bonne, c'est

26 Nous simplifions ici à l'extrême. En fait, nombre de programmes utilisent des fonctions d'évaluation contenant des termes non linéaires.

facile lorsque l'on est proche de la fin de la partie (car on peut aisément faire des analyses exhaustives), mais en milieu de partie, on doit se fier à la fonction d'évaluation elle-même qui fournit, en quelque sorte, son propre feedback. Ainsi, à chaque coup, Arthur Samuel, dans son programme, comparait la valeur courante de la fonction d'évaluation S_c avec la valeur S_p affectée à cette position lors du calcul au coup précédent (soit deux demi-coups, et donc deux niveaux auparavant). Il en tirait alors les conséquences suivantes :

- Si $S_p > S_c$, il estimait que les termes qui apportaient une contribution négative dans S avaient des coefficients c_i trop faibles et ceux qui apportaient une contribution positive des coefficients trop élevés.
- Si $S_p < S_c$, il en tirait des conclusions directement opposées.

Il ne reste alors plus qu'à modifier les coefficients d'évaluation[27].

Une autre technique classique consiste à faire jouer le programme contre une copie de lui-même avec certains coefficients de la fonction d'évaluation modifiés. Si la nouvelle copie l'emporte, alors on conserve la nouvelle valeur des coefficients, sinon on conserve l'ancienne et on essaie une autre modification.

L'*apprentissage de base*, consiste quant à lui à mémoriser, chaque fois que l'on effectue l'évaluation d'une position, la position ainsi que la valeur associée. Si par la suite, dans une recherche, cette même position est rencontrée, on ne tente pas de l'analyser avec la fonction d'évaluation mais on récupère la valeur stockée en mémoire, qui est évidemment plus précise puisqu'elle a elle-même été évaluée par une recherche α-β profonde. L'inconvénient principal de cette méthode est qu'elle réclame de très importantes capacités de stockage.

La plupart des programmes d'Othello utilisent quand à eux un apprentissage sur des formes standards. Le principe en est simple : on estime (en simplifiant) que la valeur d'une position d'Othello ne dépend que de la structure des 4 bords, des 4 coins 3×3 et du nombre de libertés des pions. On peut alors décider, par exemple, de stocker systématiquement tous les résultats de toutes les parties jouées, et en attribuer le mérite aux structures rencontrées sur les bords et dans les coins au cours de ces parties.

Un programme utilisant correctement de tels concepts devraient figurer dans les dix ou vingt meilleurs programmes mondiaux. Notons pour conclure que le problème de la suprématie homme/machine pour Othello a été réglé de façon à peu près définitive en 1997 où un match opposant LOGISTELLO au champion du monde humain Takeshi Murakami s'est terminé par le score sans appel de 6 à 0 pour le programme.

15.7 Les échecs

Les échecs sont considérés comme le jeu des Rois et surtout le Roi des jeux. Dès les débuts de l'IA, c'est sur les échecs que s'est porté l'effort le plus important et ce sont les échecs qui ont donné lieu à des luttes epiques que se sont livrés les

27 Il peut se produire un problème si la position à laquelle nous nous trouvons n'a pas été étudiée au coup précédent parce que l'algorithme α-β l'a élaguée. Normalement, ce type de situation ne se produit que si l'adversaire fait une erreur, et peut donc être négligé.

meilleures équipes universitaires comme (HITECH mené par Hans Berliner[28], ou DEEP THOUGHT, mené par Feng-Hsiung Hsu[29]) devenus depuis DEEP BLUE avec le succès que l'on sait. Il faut sans doute craindre que la victoire de DEEP BLUE face à Gary Kasparov en mai 1997 n'ait des repercussions que l'on pourrait qualifier de néfastes : le plus grand défi que l'on s'était lancé (la victoire d'un programme sur le champion du monde humain dans le cadre de partie à cadence de tournoi et sur six parties) a été réalisé et il est probable que les efforts se porteront davantage dans d'autres directions. D'autre part, même si la lutte entre les plus importantes firmes commerciales restent réels, le niveau des programmes disponibles pour micro ordinateurs est maintenant tellement élevé qu'ils battent systématiquement 99% des joueurs. On accordera donc plus d'importance à la qualité de l'interface avec l'utilisateur qu'à la force pure du programme. Mais les grandes avancées liées aux échecs demeureront.

Les pères fondateurs de l'IA, Simon et Newell, menèrent, ainsi que le psychologue hollandais de Groot et le psychologue et Grand Maître soviétique Nicolaï Kroguious une analyse de la méthode de réflexion des grands champions d'échecs. Nous allons rapidement parler des résultats de ces études avant d'aborder les techniques de programmation proprement dites. Deux raisons à cela : tout d'abord les résultats mis en évidence par de Groot (Groot 1965) et Kroguious (Kroguious 1986) sont très intéressants car ils sont applicables à la plupart des raisonnements humains dans tous les domaines ; enfin, ces résultats n'ont en rien influencé les programmes d'échecs (du moins les meilleurs) qui utilisent des méthodes de raisonnement totalement différentes aujourd'hui, preuve qu'il est bien difficile d'appliquer des techniques humaines pour faire résoudre des problèmes à des machines.

15.7.1 Les études des psychologues

Nous allons ici évoquer les travaux effectués par le psychologue hollandais de Groot[30] et ceux du psychologue et GMI soviétique Kroguious. de Groot (Groot 1965) put travailler avec un échantillon tout à fait remarquable de six GMI (dont un champion du monde, Max Euwe[31]), quatre MI, deux champions des Pays-Bas et dix experts. De par l'exceptionnelle qualité de ces joueurs, l'étude de de Groot est incomparable.

---

28 Hans Berliner présente la caractéristique intéressante d'être un très fort joueur d'échecs, puisqu'il fut champion du monde d'échecs par correspondance. Il s'intéresse à la programmation du jeu d'échec depuis de nombreuses années et passa son PhD en 1974 sur ce sujet. Pour une description de HITECH, voir (Berliner 1990).

29 Pour une description de DEEP THOUGHT, voir (Hsu *et al.* 1990).

30 Cette partie est largement développée, et remarquablement présentée dans (Laurière 1986).

31 Max Euwe, né en 1901 à Amsterdam, fut un très grand théoricien des échecs et remporta le titre mondial en 1935 face à Alexandre Alekhine (profitant, il est vrai, de la mauvaise santé d'Alekhine qui avait, comme souvent, un peu trop forcé sur l'alcool et le tabac). Il le reperdit dès le match revanche, Alekhine, s'étant mieux préparé, l'écrasa très largement. Il tenta de reconquérir son bien en 1948, le titre étant vacant après le décès d'Alekhine, mais ne termina que cinquième du tournoi hexagonal. Il échoua encore plus nettement au tournoi des candidats de 1953 ne terminant qu'avant-dernier. Il se consacra par la suite à la rédaction d'études théoriques, prit part à des études sur le comportement des joueurs d'échecs, et fut élu président de la FIDE en 1970, poste qu'il conserva jusqu'à sa mort. Sa part dans l'étude de de Groot est considérable, d'autant que Max Euwe, ancien professeur de mathématiques, avait un esprit clair et précis dans ses analyses.

De Groot et ses élèves[32] analysèrent la façon dont les joueurs d'échecs perçoivent une position. Pour ce faire, ils présentèrent à chaque joueur une position sur un échiquier pendant cinq secondes puis enlevèrent les pièces de l'échiquier et leur demandèrent de les replacer. Ils s'intéressèrent alors à deux facteurs : le nombre de pièces replacées correctement et la façon de replacer les pièces. Ils remarquèrent un certain nombre de points intéressants :

- Les GMI parvenaient à replacer l'intégralité des pièces dans près de 40% des cas alors que les experts n'y sont jamais parvenus, tout en obtenant des scores honorables.
- L'ensemble des individus testés qui savaient jouer aux échecs replaçaient les pièces par groupe logique. Ainsi, on replaçait en même temps l'ensemble des pièces qui attaquaient le roque et celles qui le défendaient. Les individus qui ne savaient pas jouer replaçaient les pièces de façon non corrélée et obtenaient des scores désastreux.
- Les grands maîtres avaient en fait réalisé une mémorisation de la position qui était proche d'une analyse. Ils se la rappelaient comme un ensemble de lignes d'attaque et de défense, et non comme un ensemble de pièces.
- Les grands maîtres reconstruisaient la position par référence à des positions de partie déjà connues qui ressemblaient à celle-ci. Ils étaient capables de dégager les grandes lignes du jeu qui avait mené à la position considérée.
- Placés devant des positions aléatoires (non plus tirées de parties réelles, mais composées de pièces posées au hasard sur l'échiquier), tous les individus, joueurs, non joueurs et GMI obtenaient des scores comparables et très mauvais.

D'autres travaux furent menés pour étudier la façon dont les joueurs d'échecs « à l'aveugle[33] » voient mentalement la position. Ainsi un article publié en 1946 par Pina Morini dans la revue italienne *Minerva Medica* estimait qu'il existait différents types de mémorisation de la position, visuelle pour certains et linguistique pour d'autres (qui relisent mentalement la liste des coups joués). En revanche, l'analyse d'une position se fait toujours par bloc logique et par analogie avec des positions connues. Kroguious fait la même remarque en remarquant les effets nocifs du découpage de la position en bloc de pièces. Il signale par exemple le cas suivant.

> « Dans la partie (figure 15.9) Romanovsky-Kasparian (Léningrad 1938), les blancs sont en mauvaise posture. Kasparian, avec les noirs, voulut forcer le mat en créant un réseau à l'aide du Cavalier et de la Dame, et en oublia le reste de l'échiquier. Il annonça mat en trois coups : 52... ♕e1+ ; 53 ♔h2, ♖xh3+ ; 54 ♗xh3, ♘f3 ? ? ?. Romanovsky très gêné, lui expliqua que le Cavalier noir était cloué par la Dame blanche : « Tout d'abord il ne le crut pas, ce n'est qu'en lui montrant du doigt la diagonale a1-h8 qu'il se rendit compte de son erreur. »

32 Comme le fait remarquer (Vicente and Brewer 1993), de Groot n'a réalisé lui-même qu'une seule expérience, celle consistant à faire replacer à tous les cobayes les pièces d'un échiquier dans une configuration valide.

33 Jouer à l'aveugle consiste à jouer aux échecs sans voir l'échiquier. Certains joueurs parviennent même à jouer plusieurs parties simultanément sans voir. Ainsi, le grand maître polonais Najdorf joua, en 1947, 42 parties simultanées à l'aveugle, en gagna 36, fit 4 nulles et connut 2 défaites. Le record a été largement battu depuis. Tous les bons joueurs d'échecs ont la capacité de jouer à l'aveugle.

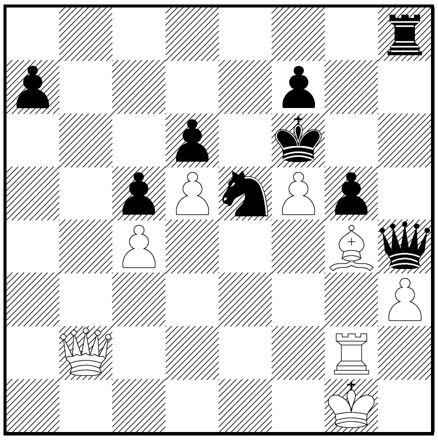

Figure 15.9 – Isolation de sous-blocs dans une position

> Kasparian, tout entier à sa bataille, se désintéressa de certaines pièces, et finit même par violer les règles du jeu. »

Ainsi, il est clair que la mémorisation de la position est fortement liée à la connaissance du sujet, et qu'elle est faite par des schémas de type connexionniste, qui n'ont pas de lien direct avec les techniques de mémorisation des ordinateurs.

De Groot s'intéressa ensuite à la façon de raisonner de chacun de ses cobayes. Il leur présenta des positions et leur demanda quel coup ils joueraient et pourquoi. Il en dégagea les enseignements suivants :

– La plupart des positions sont tout d'abord « pensées » par analogie avec des positions déjà rencontrées. Ces analogies permettent d'isoler les coups qui peuvent permettre un gain ou éviter une défaite, toujours en rapport avec des stratégies *globales* à long terme.
– La recherche du joueur est très sélective. L'aptitude principale des grands maîtres est leur capacité à identifier rapidement le, ou les bons coups, à analyser sérieusement, précisément par analogie. Une fois ces coups identifiés, tous les autres coups sont ignorés après une analyse superficielle.
– La profondeur du raisonnement est faible. Aucun grand maître n'examine une position à plus d'une dizaine de demi-coups de profondeur (alors que les ordinateurs les plus puissants peuvent examiner jusqu'à quinze ou vingt demi-coups).
– La fonction d'évaluation utilisée est floue ; elle contient des concepts difficiles à identifier clairement, comme la notion de *mobilité utile*, de *pression d'attaque*, ou de *position active* ... Ces critères remplacent l'analyse en profondeur qu'un joueur humain ne peut effectuer en raison de la limitation de ses capacités mentales. Très souvent l'évaluation est aussi effectuée par référence à des schémas connus, faciles à expliciter dans certains cas (position centralisée des Tours, Fous de la « bonne » couleur pour jouer une finale) mais parfois plus indistincts, voire liés au joueur lui-même.
– Le joueur peut effectuer des changements de plan en fonction des analyses qu'il effectue. Ces changements de plan l'amènent alors à reconsidérer les coups à examiner en profondeur.

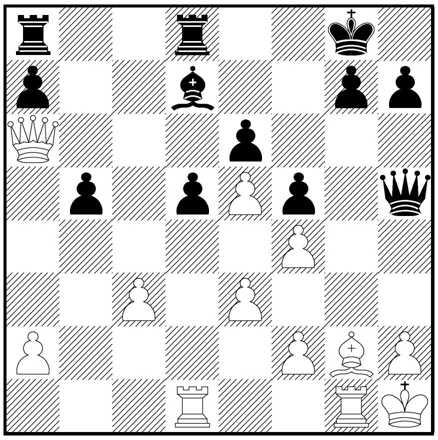

Figure 15.10 – Existence d'images résiduelles à long terme

Les études de Kroguious sont plus orientées vers une analyse « utilitaire » de la psychologie du joueur d'échecs. Capitaine de plusieurs équipes soviétiques[34], il cherche à dégager quels sont les paramètres qui interviennent dans le jugement du joueur d'échecs et renforcent ou dégradent la qualité de son analyse. Kroguious met en évidence l'existence d'images résiduelles, à la fois à long terme et à court terme. L'image résiduelle à long terme permet de reconnaître dans une position courante une référence à une position déjà vue, et d'appliquer ainsi immédiatement le même type de plan. Il cite ainsi l'exemple de la figure 15.10.

> « Dans cette position, Bogolioubov avec les blancs trouva instantanément la combinaison 22. ♗xd5,exd5 ; 23. ♖xg7+,♔xg7 ; 24. ♕f6+, ♔g8 ; 25. ♖g1+, ♕g4 ; 26. ♖xg4+, fxg4 ; 25. f5 .
>
> La clef semble impossible à découvrir si l'on ne se réfère qu'aux principes généraux. Cette combinaison est le produit d'une association d'idées. Inconsciemment, Bogolioubov s'est inspiré d'une partie connue (Bird - Morphy, Londres 1858) (figure 15.11).
>
> Ici, les noirs jouèrent 17... ♖xf2 ; 18. ♗xf2, ♕a3 avec une attaque gagnante.
>
> La solution intuitive trouve son origine dans la comparaison entre la position de la partie et une idée familière au joueur. »

Remarquons tout de suite que la similitude que signale Kroguious est loin d'être évidente.

Les études de de Groot et Kroguious sur les joueurs d'échecs sont donc particulièrement intéressantes car elles recouvrent en fait les techniques de raisonnement des humains face à la plupart des problèmes qui leur sont posés : la base de connaissances est fondamentale et l'essentiel de la résolution se fait par référence à des exemples

34 Il fut le « patron » de l'équipe d'Anatoly Karpov au championnat du monde de Lyon (1990).

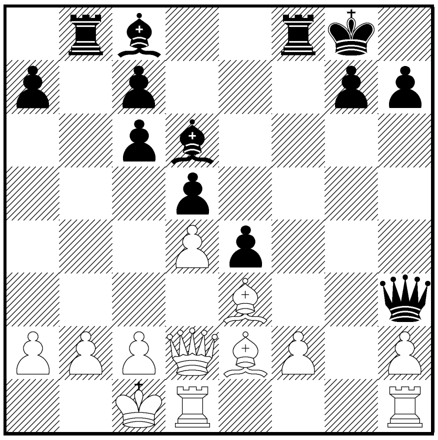

Figure 15.11 – Existence d'images résiduelles

déjà rencontrés[35]. Ceci est rendu possible par la formidable capacité du cerveau humain à stocker et trier les informations, et à retrouver instantanément (ou presque) l'ensemble des informations relatives à un objet donné, pourvu que l'on fournisse un identificateur suffisamment clair de l'objet. Ce qu'il y a parfois de plus remarquable encore, c'est que nombre d'informations peuvent disparaître, mais certaines associations apparemment sans valeur subsister[36]. Il faut d'ailleurs noter la fantastique quantité d'informations que le cerveau emmagasine tout au long d'une existence. Il faut également souligner que la capacité à mémoriser les informations, si elle est importante, n'est pas primordiale. Ce qui est réellement important c'est la capacité que l'on a à récupérer efficacement ces informations quand on est confronté à certains indices. Tous les enseignants savent bien que faire comprendre quelque chose est important, mais qu'il existe certains élèves qui, bien qu'ayant compris, restent incapables d'appliquer la méthode apprise dès que le problème est légèrement modifié dans sa présentation[37]. La technique est mémorisée, mais le cerveau est incapable de reconnaître l'analogie des situations. Il nous est impossible, à l'heure actuelle, de modéliser[38] comment le cerveau détecte les analogies[39]. Ce fut la cause de l'échec du GPS de Newell, Simon et Shaw et reste le problème majeur à résoudre aujourd'hui. Si on savait le faire, on serait non seulement capable de construire des ordinateurs reproduisant le raisonnement humain, mais on serait peut-être aussi capable de rendre les

35 C'est finalement l'histoire du polytechnicien à qui l'on demande de faire chauffer de l'eau chaude après lui avoir expliqué comment faire chauffer de l'eau froide : il suffit de laisser refroidir l'eau chaude et *l'on est ramené au problème précédent* : la phrase magique du taupin résolvant un problème.

36 C'est ainsi qu'un mot, une odeur, une image peuvent déclencher des sensations de « déjà vus » sans qu'il soit possible d'identifier complètement à quoi cette sensation se rapporte exactement.

37 Un professeur de mathématiques prétendait même présenter à chaque interrogation le même problème sous un aspect différent et avoir à la fin de l'année une fraction d'individus toujours incapable de le résoudre.

38 À l'exception peut-être des réseaux de neurones formels, mais le modèle semble tout de même bien pauvre.

39 Pour exprimer un avis personnel, je pense, comme Herbert Dreyfus, que cela rend, pour l'instant, irréaliste toute tentative de construction de programme reproduisant le fonctionnement d'un raisonnement humain.

humains plus intelligents[40] en développant les capacités de leur cerveau à raisonner correctement[41].

Après ces digressions revenons à notre sujet principal : la programmation des échecs.

15.7.2 Les ouvertures

La programmation de l'ouverture est certainement la chose la plus simple à réaliser. Les techniques à utiliser sont exactement les mêmes que pour la programmation des ouvertures pour Othello. Cependant, le nombre de positions aux échecs rend le problème un peu plus complexe. Il faut savoir que l'encyclopédie des ouvertures doit dépasser les 2000 pages.

Il s'agit plus d'un travail de bénédictin que de la réalisation exaltante d'algorithmes évolués. Notons une fois de plus la nécessité de ne pas stocker les séquences de coups, mais bien les positions en les indexant de façon correcte, afin de ne pas être trompé par une inversion de coups. Certains programmes se font encore duper. On peut aisément le vérifier avec la séquence suivante :

1.	e2–e4	e7–e6
2.	♘g1–f3	c7–c5

Il suffit alors de demander à l'ordinateur le coup qu'il jouerait face à la position résultante. S'il répond immédiatement d4, c'est qu'il a reconnu l'inversion de coup qui a transformé une partie française en une sicilienne. La séquence classique pour atteindre cette position sicilienne est en effet : 1. e4,c5 ; 2.♘f3,e6. Le coup e6 en premier par le joueur noir fait au contraire croire à un ordinateur primaire que l'on va jouer une partie française, auquel cas il attend comme séquence standard : 1. e4,e6 ; 2. ♘f3,d5.

Ne trouvant pas cette séquence standard (c5 au lieu de d5), il est perdu s'il stocke ses ouvertures sous forme de séquences de coups et non sous forme de positions indexées, car il ne reconnaît aucune séquence standard et est incapable de reconnaître une position (sicilienne).

Une technique classique consiste, pour accélérer la recherche, à indexer la position sur le numéro du demi-coup où on peut la rencontrer. On peut, bien entendu, écrire un programme qui, connaissant les séquences d'ouverture, génère la structure de données appropriée pour faire de la reconnaissance de position, la réalisation d'une telle structure par un être humain étant réellement fastidieuse.

En fait, une technique plus efficace encore consiste à réaliser un programme qui utilise conjointement les deux méthodes : mémorisation des séquences d'ouverture pour une plus grande rapidité et mémorisation des positions pour la généralité. Il n'aurait recours à la seconde méthode que lorsque la première ne lui permettrait pas de conclure efficacement.

Hormis ces détails techniques, la programmation des ouvertures ne pose aucun problème difficile.

---

40 Ce qui justifie la phrase de Patrick Winston : « Rendre les ordinateurs intelligents peut aider à rendre les hommes plus intelligents. »

41 Je crains fort hélas que les capacités des cerveaux à réaliser des analogies soient plus du domaine de la biologie que du domaine des modèles informatiques ou psychologiques.

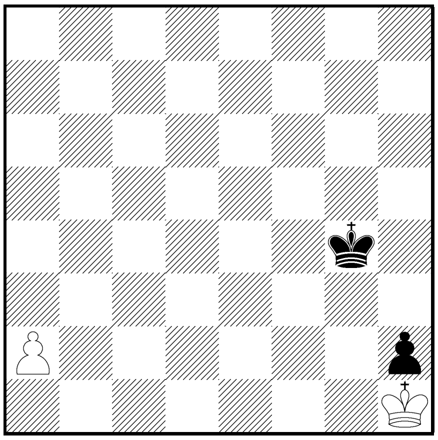

Figure 15.12 – Effet horizon dans une finale

15.7.3 Les finales

Les finales furent longtemps le talon d'Achille des programmes d'échecs avant de devenir un de leur point relativement fort.

Quel problème avait-on à résoudre ? Les ordinateurs sont efficaces en milieu de partie en raison de leur formidable habileté tactique, liée à leur grande capacité de calcul et à leurs algorithmes de recherche efficaces. Mais dans une finale, la capacité à calculer doit être bien différente. Prenons un exemple trivial. Si nous sommes dans une finale Roi et pion contre Roi et pion, et que la disposition des pièces est la suivante : pion blanc en a2, Roi noir en g4, pion noir en h2, Roi blanc en h1 avec trait au blanc (figure 15.12).

La partie peut être considéré comme terminée. En effet, le Roi noir ne peut empêcher le pion blanc d'aller à Dame en a8 et le pion noir en h2 ne peut aller à Dame, bloqué qu'il est par le Roi blanc en h1. C'est l'analyse que réalise immédiatement tout joueur, même débutant. Le bon coup est évidemment a4. Mais un ordinateur raisonnant avec un algorithme α-β à profondeur fixe va considérer les choses différemment. Tout d'abord, il voit qu'il peut prendre le pion h2 sur le champ, et qu'en revanche s'il ne le prend pas, il ne pourra pas le prendre au coup suivant si les noirs jouent Th3, il y a donc manque à gagner. D'autre part, il lui faudrait analyser la position à 10 demi-coups de profondeur pour s'apercevoir que le pion blanc ne peut aller à Dame que s'il est avancé sur le champ. Cette profondeur étant déraisonnable pour beaucoup d'ordinateurs, il sera aveuglé par sa gloutonnerie et jouera |Rxh2|, la partie se terminant alors par une nulle (le Roi noir capturera le pion blanc sans difficulté au bout de 10 demi-coups). Cet exemple nous montre la nécessité d'*inclure dans le programme des règles particulières permettant de déterminer aisément la valeur d'un pion dans une finale sans être contraint à des analyses en profondeur trop importantes*.

Un autre exemple typique de problème de finales est le suivant. On considère la position : Roi noir en f6, pion noir en g6, Roi blanc en g2, Fou blanc en f5 Cavalier blanc en g5 et Fou blanc en h1 (figure 15.13).

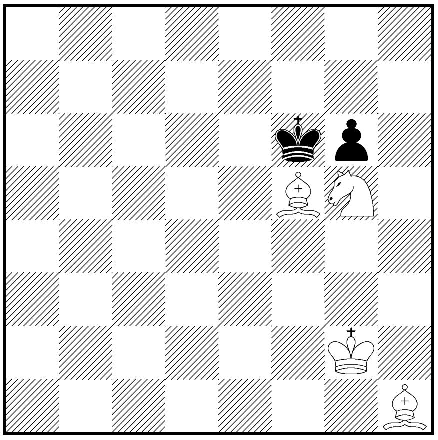

Figure 15.13 – Reconnaissance d'une finale Fou-Cavalier

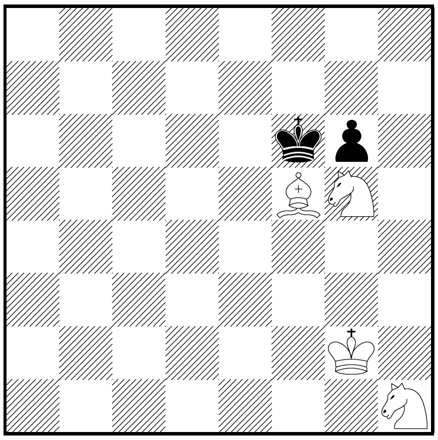

Figure 15.14 – Reconnaissance de la finale Cavalier-Cavalier

Le programme ne peut défendre simultanément son Fou en f5 et son Cavalier en g5. Il jouera donc ♗xg6, gagnant un pion avant de perdre le Fou. Il se retrouve alors à jouer une finale Roi, Fou, Cavalier contre Roi qui est gagnante. Jusque-là pas d'erreur. Remplaçons maintenant le Fou blanc en h1 par un Cavalier blanc(figure 15.14).

Le problème est apparemment inchangé, la meilleure solution semble encore de jouer |Fxg6| pour gagner un pion avant de perdre une pièce. C'est d'ailleurs ce que font tous les programmes de l'ancienne génération et nombre de joueurs d'échecs novices. Il s'agit pourtant d'une erreur grave. En effet, la finale Roi, Cavalier, Cavalier contre Roi est toujours nulle. Le coup correct ici est |Fd3|, pour sauver le Fou. Après |Rxg5|, les blancs élimineront sans difficulté le pion noir restant avec leur coalition Roi, Fou, Cavalier puis materont le Roi noir. Mais là encore, il faut savoir sacrifier le court terme pour le long terme et surtout il faut savoir *reconnaître certains types de positions particulières*.

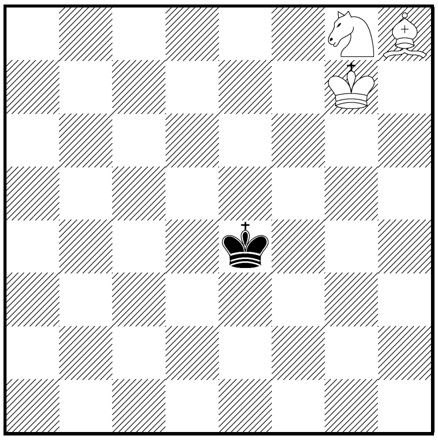

Figure 15.15 – Établir un plan

Signalons un dernier point capital dans beaucoup de finales : *la nécessité d'établir un plan*. Considérons la position suivante : Roi noir en e4, Roi blanc en g7, Fou blanc en h8, Cavalier blanc en g8 (figure 15.15).

Il s'agit d'une typique finale Roi, Fou, Cavalier contre Roi. Cette finale est gagnante. Encore faut-il mater en moins de 50 coups[42]. Nombre de joueurs de club ont échoué dans une semblable finale et tous les programmes de l'ancienne génération échouaient également. Le mat ne peut être atteint que par un jeu précis et un plan parfaitement établi. Un problème encore plus difficile est posé par la finale Roi, Cavalier, Cavalier contre Roi et Pion qui peut être gagné dans certains cas[43]. Mais la réalisation du mat en 50 coups est excessivement difficile sans plan précis.

Nous avons donc vu la nécessité d'introduire trois éléments fondamentalement nouveaux dans les programmes de finale :

– des règles particulières permettant de calculer instantanément la valeur de certaines pièces (pion passé, règle du carré, notion de « bon Fou »...) ;
– savoir reconnaître des types de positions classiques ;
– savoir appliquer des plans précis adaptés à des cas précis.

Les progrès réalisés ces cinq dernières années par les programmes en résolution de finales sont extrêmement spectaculaires[44]. Tous les exemples que nous avons présentés ci-dessus sont maintenant résolus par des programmes que l'on peut trouver dans le domaine public sur internet. Deux facteurs ont largement contribué à la progression des ordinateurs dans les finales : les tables de transposition (que nous avons déjà présentées), et l'analyse rétrograde que nous allons maintenant détailler.

---

42 Rappelons qu'il existe aux échecs une règle qui stipule qu'une partie est nulle s'il se produit une séquence de cinquante coups sans prise de pièce ou de pion.

43 C'est une des bizarreries des échecs. La finale Roi, Cavalier, Cavalier contre Roi est nulle, mais la finale Roi, Cavalier, Cavalier contre Roi et Pion peut être gagné, le pion noir étant utilisé par les blancs pour bloquer une case de fuite du Roi noir.

44 Une des critiques que l'on peut adresser à la présentation des techniques de jeu dans (Laurière 1986) est qu'elle date sérieusement par rapport aux résultats actuels.

Il s'agit d'une méthode permettant de constituer des base de données indiquant à l'ordinateur quel est le coup à jouer dans une position donnée pour arriver au gain de la façon la plus rapide. La constitution de semblables bases s'est beaucoup développée dans les dix dernières années. Ainsi, il existe une base de données permettant de jouer parfaitement des finales comme (Roi + Cavalier + Fou) contre (Roi + Cavalier), une finale que même un champion du monde aurait des difficultés à gagner[45].

Pour construire de semblables bases de données, les ordinateurs utilisent une méthode appelée *analyse rétrograde*. Prenons comme exemple la finale (Roi + Dame) contre (Roi). Il existe, si on néglige les symétries, $64 \times 64 \times 64 = 262144$ positions de départ. Certaines de ces positions sont impossibles (les deux Rois adjacents par exemple) : elles sont éliminées. Dans les positions restantes, on recherche celles où les noirs sont mats : ce sont les positions où le Roi noir est déjà en échec et où aucun des coups dont il dispose ne permet de le soustraire à cet échec. À partir de là, on détermine aisément les positions de mat en un coup. Ce sont les positions qui permettent au blanc d'atteindre en un coup une position de mat. On peut ainsi déterminer récursivement les positions de mat en $n + 1$ coups à partir des positions de mat en n coups.

La constitution de bases de données de ce type est une technique bien particulière, relativement éloignée de la programmation classique des jeux, mais fort intéressante. Les problèmes à 5 pièces (sans pion) ont été exhaustivement résolus[46], et l'on travaille activement sur les problèmes à 6 pièces. En Octobre 1991, Larry Stiller (avec l'aide de Noam Elkies et Burton Wendroff) a construit la plus longue finale gagnante connue à ce jour. La position initiale est représentée sur la figure 15.16. La résolution complète demande 223 coups. Il est clair que ce type de finale est inaccessible à l'être humain. La question est de savoir si les règles doivent être modifiées pour en tenir compte.

15.7.4 Le milieu de partie

Nous allons retrouver les deux grands protagonistes du milieu de partie : la fonction d'évaluation et l'algorithme α-β. Mais nous introduirons également quelques techniques spécifiques ou, en tout cas, surtout utilisées pour les échecs.

La fonction d'évaluation aux échecs est extrêmement complexe. La base d'une fonction d'évaluation est la valeur accordée aux pièces. On peut dire que les valeurs que la plupart des programmes accordent aux pièces sont les suivantes :

Dame	:	9
Tour	:	5
Fou	:	3
Cavalier	:	3
Pion	:	1

45 L'ordinateur a d'ailleurs prouvé que les règles actuelles du jeu d'échecs étaient, dans un certain sens, « incorrectes ». En effet, la règle stipule qu'une partie est déclarée nulle si aucune prise de pièces ou aucune promotion n'a eu lieu en cinquante coups. Or, il existe des cas où la finale (Roi + Cavalier + Fou) contre (Roi + Cavalier) ne peut être gagnée qu'en soixante-dix-sept coups.

46 Pour être tout à fait précis, Larry Stiller calcula exhaustivement les 80 positions les plus intéressantes en 227 minutes sur une CONNECTION-MACHINE (TMC 1991) à 32000 processeurs (une demi-CONNECTION-MACHINE en fait).

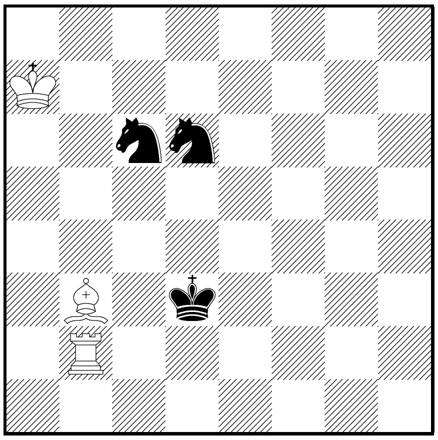

Figure 15.16 – Position initiale de la plus longue finale connue

Il reste maintenant à raffiner la fonction d'évaluation. Tout un ensemble de paramètres interviennent. En voici quelques uns :
- mettre son Roi en sécurité (Roquer, éviter d'affaiblir le roque par des avancées de pions) ;
- conserver la paire de Fous ;
- dominer le centre ;
- garder un maximum de mobilité ;
- centraliser les Cavaliers ;
- placer les Fous sur des diagonales ouvertes ;
- placer les Tours sur des colonnes ouvertes ;
- doubler les Tours sur les colonnes ouvertes ;
- conserver une bonne structure de pions (pas de pions doublés, conserver des pions liés, éviter les pions arriérés) ;
- obtenir des pions passés soutenus (capital en vue de la finale) ;
- conserver le Fou de la « bonne » couleur (un Fou est de la bonne couleur lorsqu'il peut attaquer les cases où sont bloqués les pions de l'adversaire. C'est un concept fondamental pour avoir un avantage en finale.) ;
- favoriser les échanges si le passage en finale est favorable.

Ce n'est qu'un petit aperçu de ce que doit être une bonne fonction d'évaluation aux échecs.

Notons cependant un point capital : *la notion de plan stratégique d'ensemble est absente.* On peut, certes, introduire dans la fonction d'évaluation quelques éléments pseudo-stratégiques, comme l'attaque sur le roque adverse « à la baïonnette[47] », mais cela reste très limité. Il s'agit bien là du principal défaut des programmes d'ordinateurs. Cela est particulièrement sensible lorsqu'ils passent de la séquence d'ouverture qu'ils jouent « par cœur » au milieu de partie. En effet, à chaque ouverture est associée une idée stratégique et la suite de la partie doit développer cette idée. Ainsi une sicilienne doit être jouée de façon agressive par les blancs qui doivent attaquer

47 Technique consistant à attaquer le roque adversaire avec les pions.

rapidement le petit roque noir, alors que la chance des noirs se situe en général à l'aile dame. Or un programme est incapable de saisir ces subtilités stratégiques et appliquera la même fonction d'évaluation quelle qu'ait été l'ouverture. Tous les programmes d'ordinateur, même les meilleurs, manquent de « liant » dans leur jeu, ce qui rend leur style facilement reconnaissable et les laisse encore vulnérables. On ne connaît actuellement aucune méthode pour remédier à cet état. Simplement, la force tactique des programmes leur permet de compenser leur faiblesse stratégique.

La technique de recherche dans l'arbre de jeu est un algorithme α-β tel que nous l'avons décrit au paragraphe précédent. La technique de contrôle de temps de jeu et de reclassement des coups avant d'aborder des recherches plus profondes est également identique. Il faut cependant introduire certaines techniques indispensables aux échecs[48] :

L'élagage de futilité : Nous savons que l'algorithme α-β élague une branche au niveau n dès que l'évaluation partielle de cette branche est inférieure (ou supérieure si le niveau est pair) à l'évaluation définitive du niveau $n-1$. Supposons par exemple que l'évaluation du niveau $n-1$ soit 3 et qu'une évaluation partielle au niveau n soit 3.01. Certes, l'évaluation n'est pas inférieure, mais il est clair qu'elle ne sera pas non plus franchement meilleure. On peut donc élaguer cette branche. Cette technique est appelée *élagage de futilité*.

Le coup meurtrier : Supposons que lors de la recherche α-β, l'algorithme note que le coup de premier niveau p_1 est détruit par la riposte de d_1 (c'est-à-dire que si l'adversaire répond d_1 après p_1 le score de la position s'effondre). Cela signifie que d_1 est un *coup meurtrier*. Lors de l'examen du coup de premier niveau suivant p_2, la première riposte que l'algorithme α-β devra examiner sera précisément (si cela est possible) d_1 car si le coup d_1 a été meurtrier pour p_1, il a de fortes chances de l'être également pour p_2. De tels coups sont liés, aux échecs, à la notion de menace et la même menace reste souvent valable face à de nombreux coups. En procédant de cette façon, on améliore encore l'efficacité de l'élagage de l'algorithme α-β.

Utilisation du temps de réflexion de l'adversaire : Après avoir joué un coup, il est bon de retenir l'intégralité de la branche. En effet, on peut relancer la recherche *α-β pendant le temps de réflexion de l'adversaire* sur la position telle qu'elle se produirait si l'adversaire joue la riposte que nous estimons la meilleure pour lui. De cette façon, si l'adversaire joue effectivement ce coup, nous aurons déjà effectué une grande partie de l'évaluation de la position.

Retour à l'équilibre : Si l'algorithme α-β travaille à une profondeur fixe, il s'expose à de sérieux ennuis. Supposons en effet que l'algorithme α-β descende à une profondeur 3 et qu'il trouve une séquence du genre : je déplace mon Cavalier (premier niveau), on me le prend (deuxième niveau), je prends la Dame adverse (troisième niveau). Bien qu'il enregistre la perte de son Cavalier, le résultat est globalement positif puisque le programme comptabilise la prise de la Dame adverse. Mais supposons que le quatrième coup (non examiné puisqu'il s'est arrêté à la profondeur 3) est la prise de sa Dame. Le bilan global de l'opération est là franchement négatif. Pour éviter ce type de comportement,

---

48 Ces techniques peuvent s'appliquer, avec plus ou moins de bonheur ou d'utilité, à d'autres jeux.

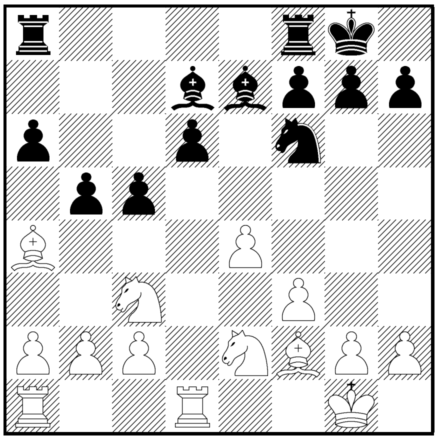

Figure 15.17 – Effet horizon lié à un coup intermédiaire

tous les algorithmes utilisent une stratégie dite *de retour à l'équilibre*. Une fois atteint leur profondeur maximale d'évaluation, ils poursuivent le calcul de la position terminale en continuant à examiner toutes les positions (et seulement les positions) découlant de prises ou d'échecs, jusqu'à ce qu'il n'y en ait plus. Ils peuvent ainsi éviter ce type de mauvaise évaluation.

Recherche secondaire : Un autre problème qui guette les ordinateurs est *l'effet horizon*. Voici un exemple classique dû à Hans Berliner (cf. figure 15.17).

Le Fou blanc en a4 est perdu. En effet, il ne pourra échapper au pion b5 ou au pion c5 (qui va venir en c4). La séquence qui provoquera la perte du Fou est ♗b3 suivi de c4 par les noirs et le Fou est perdu. Mais un algorithme α-β tente simplement de repousser cette éventualité au-delà de son horizon de recherche (d'où le nom d'*effet horizon*). Il va donc tenter de jouer une séquence de coups à riposte forcée qui repoussera hors de son champ de vision la séquence « déplaisante ». Dans le cas présent, il prévoit par exemple (profondeur 5) : 1. e5,d×e5 (obligatoire car sinon le Cavalier en f6 tombe), 2.♘d5,♘×d5 (on ne peut jouer 2. . . .,b×a4 car alors 3.♘×e7+), 3.♖×d5.

Arrivé à cette position, le programme applique son mécanisme de retour à l'équilibre. Mais il ne voit rien de dangereux car après 3. . . .,b×a4 il voit 4. ♖×d7 et récupère le Fou. En fait, son raisonnement est incorrect car les noirs intercalent le coup intermédiaire ♗e6 et le jeu se poursuit en fait par : 3. . . .,♗e6 ; 4.♗b3,c4 et le Fou est perdu.

Le programme, pour trouver la solution correcte, devrait effectuer la recherche à la profondeur 7, ce qui est prohibitif pour un ordinateur moyen. Certains écrivaient, il y a quelques années, que ce type de situation était ingérable par un ordinateur. En fait, la plupart des bons programmes de jeu utilisent aujourd'hui une technique appelée *recherche secondaire* qui leur permet de trouver la solution du problème précédent en quelques secondes. Il s'agit simplement de relancer la recherche α-β à une profondeur supérieure, mais uniquement sur la branche (voire uniquement sur la position terminale de la branche) qui contient le coup sélectionné. Il s'agit, en quelque sorte, d'une vérification de la validité du coup choisi. Comme l'évaluation n'est faite que sur la position terminale sélectionnée, le temps nécessaire est raisonnable. Certes, ce mécanisme ne permet pas de faire disparaître l'effet horizon, mais il en limite considérablement les effets[49].

L'algorithme SEX : L'algorithme SEX (Search EXtension) (Levy *et al.* 1989) a aussi pour but de limiter les problèmes d'effet horizon. Dans un mécanisme de recherche minimax classique, la profondeur de recherche est fixée au départ à n et à chaque descente d'un niveau dans l'arbre, on diminue de 1 la valeur de n. Lorsque n atteint 0, la recherche est terminée.

Avec l'algorithme SEX, on utilise une variable *SX* qui prend une valeur nettement plus grande que la profondeur à laquelle on veut chercher (une valeur typique est dix fois la profondeur, si n vaut 3, *SX* vaut 30), mais à chaque descente d'un niveau dans l'arbre, on diminue *SX* d'une quantité variable en fonction de l'intérêt du coup joué. Ainsi, un coup « moyen » soustrait 10 à *SX*, une prise ou un échec est un coup intéressant, et on diminue peu *SX* (2 ou 3) ; en revanche, une retraite de pièces est un coup peu intéressant et on diminue beaucoup *SX* (jusqu'à 35). La recherche est terminée quand *SX* devient négatif ou nul.

De cette façon, les coups intéressants sont étudiés de façon plus profonde, et les coups « peu intéressants » sont rapidement abandonnés.

L'heuristique du « coup nul » : Appelé *Null Move Heuristic* en anglais, cette technique consiste à supposer que, dans une situation donnée, un joueur exécute deux coups successifs[50] (son adversaire ne joue pas, d'où le nom de « coup nul »). Si la situation ainsi générée n'est pas clairement favorable, alors le premier coup est élagué de l'arbre de recherche. Cette technique permet d'améliorer l'efficacité de l'élagage mais provoque parfois des élagages intempestifs, comme un match du programme Mephisto au tournoi ACM de 1991 le montra…

Signalons, pour conclure, un des inconvénients de la méthode minimax. Si l'ordinateur se trouve dans une situation perdante et qu'il a le choix entre deux coups, dont le second est légèrement plus mauvais que le premier mais tend un piège à l'adversaire, il sera incapable de s'en apercevoir. Le principe minimax choisira obligatoirement le premier, alors qu'il garantit presque assurément la perte de la partie, alors qu'un joueur humain tentera le tout pour le tout et jouera le second coup. La notion de piège est bien difficile à formaliser, cependant Donald Michie proposa, il y a quelques années, une intéressante méthode qu'il est bon de rapporter. Supposons que nous ayons un arbre à deux niveaux que nous parcourons par une méthode minimax. L'ordinateur cherche la valeur S_1 du premier coup de niveau 1. À partir de la position P_1 correspondant à ce coup, son adversaire peut atteindre trois positions P_{11}, P_{12} et P_{13} dont les scores sont respectivement S_{11}, S_{12}, S_{13}. Théoriquement, l'ordinateur devrait remonter comme valeur au niveau 1 le minimum de ces trois valeurs. Michie propose de raffiner la méthode en prenant comme valeur du niveau 1 non pas

49 Sur les techniques dites de *Singular Extension*, voir (Anantharaman *et al.* 1990).

50 Pour une description précise, voir (Goetsch and Campbell 1990).

le minimum, mais la combinaison linéaire :

$$p_{11} \times S_{11} + p_{12} \times S_{12} + p_{13} \times S_{13}$$

p_{11}, p_{12}, p_{13} représentent les probabilités que l'adversaire a de jouer les coups S_{11}, S_{12}, S_{13} d'après l'ordinateur.

Comment évaluer ces probabilités ? Michie a mis au point une méthode pour les échecs, qu'il appelle *modèle de discernement*. Il estime que le discernement d'un joueur face à une position donnée est donné par la formule empirique :

$$d = (C/1000 + 1)3^{1/(n+0.01)}$$

C représente le classement ELO[51] du joueur que l'on affronte et n le nombre de coups dont on a calculé une valuation. Pourquoi cette formule ? Il est clair qu'un joueur joue d'autant mieux qu'il est plus fort. D'autre part, nous savons que plus il y a de coups différents possibles et plus le choix est difficile, surtout s'ils ont des valeurs proches. Michie estime alors que la probabilité qu'un joueur de discernement d face à une position joue un coup menant à un score estimé v sera :

$$p = Ad^v$$

où A est un coefficient bien choisi pour normer la probabilité. Il serait intéressant de tester la méthode de Michie, mais aucun programme ne semble l'utiliser aujourd'hui.

Enfin, et d'après[52] Feng-Hsiung Hsu (responsable du projet DEEP THOUGHT), les programmes d'échecs à l'heure actuelle n'utilisent pas de technique d'apprentissage[53] :

> « Une grave erreur doit être évitée : DEEP THOUGHT n'apprend pas. La fonction d'évaluation que nous utilisons est linéaire par rapport à presque tous ses termes (de 100 à 150). Nous ne faisons qu'une optimisation du modèle de mérite dans un espace multi-dimensionnel, en utilisant un mécanisme de régression linéaire. Ce processus produit des résultats intéressants, même s'il n'est pas clair qu'il produise toujours les effets désirés. »

Pour plus d'informations sur les programmes qui jouent aux échecs, on ne peut que conseiller (Levy and Newborn 1991; Marsland and Schaeffer 1990).

---

51 Le classement ELO d'un joueur est une procédure internationale permettant d'attribuer une force à un joueur d'échecs. La méthode de calcul de ce nombre de points est complexe. Disons que l'on gagne des points chaque fois que l'on bat ou que l'on annule contre un joueur plus fort que soi, et que l'on en perd dans le cas contraire (présentation fort sommaire et quelque peu inexacte). Pour donner une idée, le champion du monde actuel a un ELO de l'ordre de 2800, un GMI tourne autour de 2500, un MI autour de 2200, la moyenne de l'ensemble des joueurs d'échecs est d'environ 800. Les meilleurs programmes commerciaux actuels ont une force de l'ordre de 2500/2600 points.

52 Correspondance privée.

53 Sans qu'il soit toujours simple de préciser ce qui est ou n'est pas de l'apprentissage. Il est souvent très tentant, et dans tous les domaines, d'utiliser ce genre de mot, à la sémantique « attirante » et mal définie, pour séduire les financeurs potentiels. Ceux-ci devraient, comme dans les restaurants, se méfier des menus aux noms pompeux et aller faire un tour en cuisine…