BibiChess

Lucas Braesch & Romain Burgot

7 juin 2006

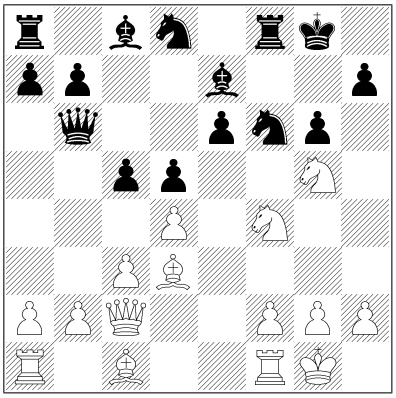


Fig. 1 – TARRASCH – ECKART (1889)
Blanc joue et gagne.

Résumé

On se propose ici d'écrire un programme en `C++`, capable de jouer aux échecs. Aujourd'hui, il existe de nombreux logiciels gratuits, dont les meilleurs culminent à près de 2 800 elo. Bien entendu, nos ambitions sont très inférieures, mais le programme devra au moins jouer des coups «tactiquement correct» et ne pas boguer. Pour prendre un exemple caractéristique, il trouvera **14. ♗×g6 !** dans la position de la Fig. 1, mais n'arrivera probablement pas à construire une telle position tout seul.

Table des matières

♔♕♖♗♘♙

Introduction

Avant de commencer la lecture de ce rapport, le lecteur non-spécialiste des échecs pourra se reporter aux annexes :

- les règles spécifiques du jeu d'échec, brièvement rappelées en annexe : A.1.
- la notation standard des coups, qui sera largement utilisée par la suite : voir A.2.

Par ailleurs, nous tenons à signaler que le programme ne fonctionne pas tout seul, mais au moyen d'une interface externe. Afin de pouvoir le tester, nous renvoyons le lecteur au manuel de l'utilisateur, en annexe B.1.

Cahier des charges

Notre but est d'écrire un programme capable de jouer une partie d'échec complète :

1. Celui-ci devra donc être capable de résoudre des problèmes tactiques (comme la Fig. 1), mais aussi de jouer de façon stratégique : développer ses pièces, contrôler le centre et «avancer» selon un plan.
2. De plus, il faudra que l'utilisateur puisse bénéficier d'une interface conviviale pour affronter le programme, ou le voir affronter d'autres programmes.

3. Enfin, le programme pourra consulter une bibliothèque d'ouvertures pour les premiers coups.

Choix techniques

Plusieurs choix techniques s'imposent alors. D'abord le choix du langage de programmation : nous avons choisi C/C++. Cela signifie donc que le code est écrit en C++, avec certaines parties en C. En voici brièvement les raisons :

1. Nous avons besoin d'un langage orienté objet, parce que c'est pratique et agréable.
2. Cependant, nous avons aussi besoin d'un langage de bas niveau, pour des raisons efficacité. En l'occurrence, BibiChess utilise la «fameuse» méthode des *bitboards* pour la génération des coups. Nous verrons cela plus tard, mais disons juste que nous avons absolument besoin d'opérateur bit à bit sur des entiers 64 bits.

En ce qui concerne l'interface, nous avons choisi de reléguer cette partie à un programme extérieur, avec lequel nous communiquons via un protocole standard, nommé Universal Chess Interface (UCI). De nombreuses raisons justifient ce choix :

1. Coder nous même une interface est une tâche longue et peu intéressante. Nous préférons nous concentrer sur le moteur en tant que tel. De plus, les interfaces existantes évoluent «toutes seules» et nous bénéficions ainsi gratuitement de ces évolutions.
2. En fait, écrire nous même l'interface est non seulement pénible, mais même pas souhaitable. En effet, une interface MFC propre à notre programme souffrirait nécessairement des limitations suivantes :
   (a) L'impossibilité de jouer des tournois automatiques avec d'autres programmes, essentiel pour améliorer le BibiChess.
   (b) Il aurait fallu coder nous même la gestion du livre d'ouverture. Pour commencer il aurait fallu en créer un, donc écrire un compilateur de parties PGN[1]. Heureusement, les interfaces UCI font cela très bien à notre place.
   (c) La portabilité des MFC est très limitée[2], tandis qu'un code ANSI C++ comme le notre devrait fonctionner sur n'importe quel compilateur et n'importe quelle plate-forme[3].

Ce rapport s'articule en 3 parties. Dans un premier temps, nous présentons les algorithmes utilisés (parties 1 et 2). Ensuite, nous verrons les éléments principaux de l'implémentation : la génération des coups et l'évaluation positionnelle. Enfin, nous présenterons brièvement le travail de déboguage effectué tout au long du projet.

[1] Portable Game Notation : la notation standard des parties.
[2] seul Visual C++ compile un programme MFC et seul Windows peut exécuter le code compilé.
[3] ... à un détail près toutefois, la fonction first_bit codée en assembleur Intel pour des raisons d'efficacité.

1 Principe de l'IA

1.1 L'algorithme minmax

1.1.1 Approche théorique

Une partie d'échec ne présente que 3 issues : blanc gagne, noir gagne ou partie nulle. L'arbre du jeu étant fini, on s'aperçoit – par induction rétrograde – qu'il existe des équilibres de Nash parfaits. Cela signifie donc qu'une (exactement) des trois propositions suivantes est vraie :

1. blanc possède une stratégie gagnante,
2. noir possède une stratégie gagnante,
3. blanc et noir possèdent tous les deux une stratégie de partie nulle. Autrement dit, blanc peut au moins forcer la nulle contre toute défense de noir et inversement. Ainsi deux joueurs infiniment intelligents ne feraient que des parties nulles.

Même si ce n'est pas prouvé, il est probable que l'équilibre du jeu d'échec soit une partie nulle. Cela signifie qu'à chaque coup, il existe une correspondance de meilleure réponse, assurant la nulle contre toute défense. Ainsi, dès qu'un des deux joueurs sort de cette correspondance, l'autre peut le sanctionner par une stratégie gagnante.

1.1.2 Évaluation d'une position

Hélas, le seul moyen de connaître la correspondance de meilleure réponse d'un noeud donné de l'arbre est d'effectuer une recherche récursive allant jusqu'aux feuilles de l'arbre, sur lesquelles on peut attribuer des scores $+\infty, -\infty, 0$ (gagner, perdre ou annuler) : c'est la stratégie minmax au sens de la théorie des jeux. Malheureusement, la taille de l'arbre est gigantesque et ceci est tout simplement impossible. On cherchera donc à effectuer un **minmax local**, en appliquant le principe d'induction rétrograde sur un arbre extrait de taille limitée. Pour cela, il faut attribuer à chaque noeud (pas seulement aux feuilles) des scores. Assez naturellement le score d'un joueur sera la somme

- d'un **score matériel** : nombre de points, en comptant classiquement

$$\♙ = 1, \quad \♘ = \♗ = 3, \quad \♖ = 5, \quad \♕ = 9$$

- et d'un **score positionnel** : activité des pièces, sécurité du roi, pions passés, isolés, doublés, arriérés, bon fou, mauvais fou, paire de fou etc...

Autrement dit, on veut que le programme ne réfléchisse pas uniquement en termes tactiques mais aussi en termes positionnels, c'est-à-dire qu'il ait une culture du jeu d'échec et une intuition, de façon à jouer comme s'il avait un plan un peu construit. Enfin, le score de blanc est la différence de son score et de celui des noirs – et inversement. Ainsi, les scores sont symétriques (de somme nulle tout le temps). Un bonus positionnel ou matériel pour blanc sera donc un malus pour noir – et inversement. Par conséquent, l'ordinateur cherchera de manière égale à consolider sa position, ou a détruire celle de l'adversaire (c'est pourquoi il jouera un peu selon un plan).

1.1.3 L'algorithme minmax

Le programme doit donc chercher le meilleur coup en sachant que son adversaire fait de même et ainsi de suite... Le raisonnement doit s'arrêter à une certaine profondeur. Autrement dit :

1. le meilleur coup de A à profondeur d est obtenu en jouant tous les coups possibles et en retenant celui qui minimise le score de B, partant du principe que B applique ce raisonnement à profondeur $d-1$.

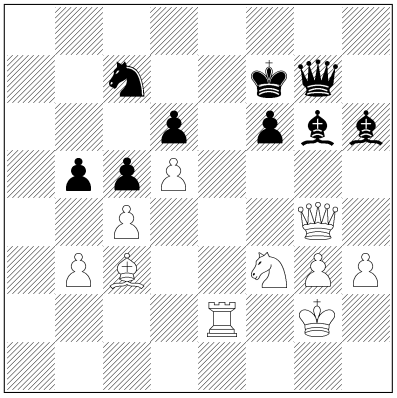
2. le meilleur coup de A à profondeur 1 est obtenu en jouant tous les coups possibles et en retenant celui qui maximise le score de A – ou minimise le score de B – sans tenir compte des réponses de B.

Dans un pseudo-code, volontairement imprécis, voila comment se présente l'algorithme minmax :

```
search(depth)
{
    loop through all moves {
        play move
        if (depth > 1) move_score = -search(depth-1) // récursion sur les noeuds
        else score = static_position_score // évaluation des feuilles
        best = max(best, score)
        undo move
    }
    return best
}
```

1.1.4 L'effet d'horizon

Nous allons voir ici, à travers un exemple caractéristique, que l'algorithme minmax de base peut véritablement jouer des coups stupides. En fait, nous verrons que ce n'est pas la stratégie minmax en tant que telle qui est en cause, mais l'arbre sur lequel nous l'appliquons : on ne peut pas se contenter d'un arbre à profondeur fixée.



Profondeur	Variante	Gain matériel
1	♕×g6+	♗ = +3
2	♕d7+	0
2	c×b5	0
3	♕d7+	♕ = +9
4	♕c8	♘ = +3

FIG. 2 – Inconsistance de l'algorithme minmax.

Étude d'un exemple. Sur la FIG. 2, nous voyons clairement l'inconsistance du minmax. En effet, les scores à profondeur n et $n+1$ n'ont généralement rien à voir entre eux :

1. A profondeur 1, le «meilleur coup» au sens minmax est **1. ♕×g6+ ?**. On évalue alors directement la position, alors que blanc a une dame en prise !
2. A profondeur 2, le minmax comprends enfin que **1. ♕×g6+ ?** est réfuté par **1. . . . ♕×g6**. En fait, toute capture des blancs est directement re-capturée. Ainsi le minmax préfèrera **1. ♕d7+**, qui force **1. . . . ♔f8** ou **1. . . . ♔g8** : ici, le score minmax est donc nul.
3. A profondeur 3, le minmax préfèrera : **1. ♕d7+ ♔f8; 2. ♕×g7+**. Il pensera donc gagner une dame sec, sans jamais envisager les re-captures **2. . . . ♔×g7** ou **2. . . . ♗×g7**.
4. Ce n'est donc qu'a profondeur 4 qu'il trouve le bon coup **3. ♕c8 !**

Conclusion. Plutôt que de s'appesantir d'avantage sur ce genre d'exemple (qui ne manque pas), cherchons plutôt à en tirer des conclusions, afin de définir correctement l'arbre à explorer. Il ne s'agit donc pas d'un arbre à profondeur équilibrée, à cause d'un effet d'horizon manifeste :

on ne peut pas évaluer n'importe quelle position. Ainsi, évaluer une position dans laquelle il reste des séquences de captures, ou une position en échec, donne des résultats pour le moins douteux.

1.2 Extensions tactiques

1.2.1 Quiescent search

Pour résoudre le problème d'instabilité de la recherche minmax, nous venons de voir pourquoi il est impératif de n'évaluer que des positions «calmes», c'est-à-dire sans captures ni échec. En effet, la moindre des choses, avant même de penser à évaluer quelque score positionnel que ce soit, est de résoudre les suites de captures et d'échecs. La solution classique à ce problème (voir [5]), appelée *quiescent search*[4] effectue, en partant des feuilles de l'arbre :

1. une recherche à profondeur infinie sur les captures,
2. en étendant les échecs qui en résultent.

Dans un pseudo code un peu simplifié, voici ce que cela signifie :

```
quiesce()
{
    score = eval()
    if we're in check {
        move_list = all legal moves   // check escapes
        if none return -MATE          // no escape -> we're mate
    } else move_list = captures only

    loop trough the move_list {
        play(move)
        score = max(score, -quiesce())
        undo(move)
    }
    return score
}
```

Il faut bien noter la première ligne `score = eval()`, qui a pour effet de laisser le choix au programme d'initier la séquence de captures, ou bien de se contenter du score actuel. Effectivement, le meilleur coup n'est généralement pas une capture. On constatera aussi que le *quiescent search* peut trouver des mat. Par exemple, après **1. f4 e5; 2. f×e5 d6; 3. f×d6 ♗×d6; 4. ♘c3 ?** [4. ♘f3]; l'algorithme trouve un mat a profondeur 1, grâce aux extensions du *quiescent search* (Fig. 3).

En effet, après **4. ... ♕h4+ !** le *quiescent search* trouve directement la suite de captures et d'échecs **5. g3 ♕×g3+; 6. h×g3 ♗×g3#**. De la même façon, **5. ... ♗×g3+; 6. h×g3+ ♕×g3#** donne le même résultat. Enfin, n'oublions pas que le minmax standard ne trouve le mat qu'à profondeur 5, ce qui nécessite environ $40^5 \approx 100$ millions de coups à calculer. De plus, le «meilleur» coup à profondeur 1 – au sens minmax – est **4. ... ♗×h2 ?**.

1.2.2 Extensions de recherche

Toujours dans la même idée (résoudre en priorité les variantes tactiques), il faut rajouter des extensions au minmax normal : échec, re-capture, coup forcé, poussée de pion en septième rangée. Ainsi, l'arbre de recherche utilisé (minmax + quiescent search) devient assez déséquilibré. Concrètement, l'algorithme minmax du 1.1.3 se réécrit ainsi, pour tenir compte des extensions et du quiescent search :

[4] *Quiescent* est synonyme de *quiet* en anglais : silencieux. Les «boules Quies» sont un bon moyen mnémotechnique.

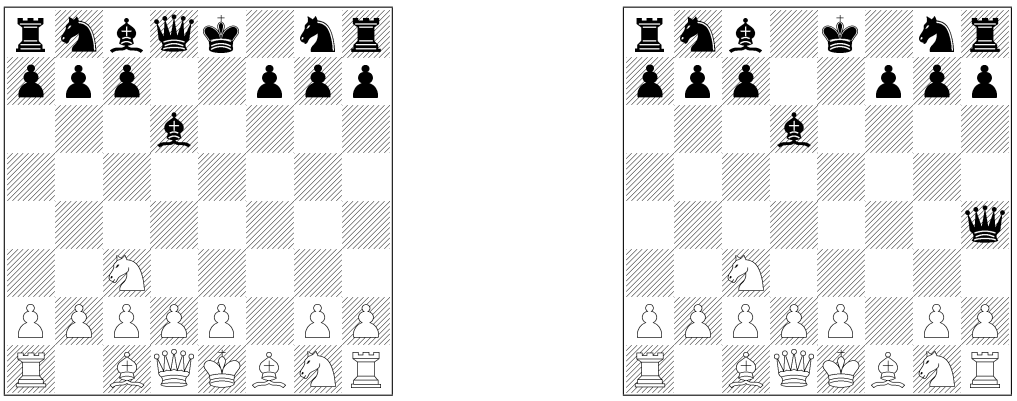


FIG. 3 – Résolution de mat par le *quiescent search*.

```
search(depth)
{
    if (no legal move) return in_check ? -MATE : 0 // mate or stalemate
    if (in check, recapture, 1 legal move, pawn-push to 7th rank) depth++
    loop through all moves {
        play move
        score = depth > 1 ? -search(depth-1) : -quiesce()
        undo move
        best = max(best, score)
    }
    return best
}
```

1.2.3 Retour sur l'exemple 1.1.4

Nous reprenons ici l'exemple de la FIG. 2, afin de mieux comprendre dans quelle mesure les extensions tactiques précédentes améliorent la qualité de l'algorithme. En utilisant les extensions de recherche (re-capture, échec, poussée de pion en septième rangée) et le quiescent search (captures et position en échec), voici les résultats obtenus :

1. A profondeur 1, nous obtenons déjà : **1. ♕d7+ ♔g8; 2. ♕×d6 b×c4; 3. b×c4** et les blancs gagnent un pion.
2. A profondeur 2 : **1. ♕c8 ♕f8; 2. ♕×c7+ ♔g8** et les blancs gagnent un cavalier.

On obtient ainsi le bon coup à profondeur 2, ainsi qu'a toute profondeur supérieure. L'amélioration apportée par les extensions tactiques est donc énorme, d'autant plus que l'algorithme est désormais stable. Nous avons largement réduit l'effet d'horizon qui donnait des résultats délirants, comme le score +9 obtenu par le minmax à profondeur 3 – **1. ♕d7+ ♔f8; 2. ♕×g7+**.

2 Améliorations de l'algorithme

Jusqu'à présent, nous avons cherché à bien définir l'arbre sur lequel il convient d'appliquer la stratégie minmax. Rappelons que la construction de cet arbre se résume à un principe de bon sens : résoudre les séquences tactiques en priorité, avant toute évaluation. Le problème qui se pose maintenant est la taille – énorme – de l'arbre à parcourir. Bien entendu, la complexité de l'algorithme est en $\mathcal{O}(b^n)$, où n est la profondeur et b est le facteur de branchement de l'arbre (i.e. nombre moyen de branches par noeud). Nous allons voir qu'il existe des techniques de *pruning*[5], permettant de réduire sensiblement le nombre b. Ces techniques de *pruning* peuvent être classées en deux catégories :

1. Les méthodes de *pruning* théoriquement fondées, c'est-à-dire dont on peut démontrer qu'elles obtiennent le même résultat que le minmax non optimisé – en réduisant l'arbre à parcourir. Ces méthodes se résument essentiellement à l'algorithme PVS (voir 2.1) et les Hash Tables (voir 2.2).
2. Les méthodes heuristiques, théoriquement non fondées, mais empiriquement valables, que nous verrons en 2.3.

2.1 L'algorithme PVS

2.1.1 Un exemple

L'Alpha-beta pruning est une astuce qui réduit énormément la taille de l'arbre. Pour l'instant, l'algorithme minmax recherche chaque réponse possible, pour tous les noeuds. Etant donné le facteur de branchement moyen (de l'ordre de 40), cette méthode est donc trop laborieuse. L'Alpha-beta pruning est basée sur l'élimination des menaces non-crédibles. Imaginons que le programme soit occupé à chercher tous les coups à la racine de l'arbre. Pour l'instant, il a passé les 6 premiers coups, et le meilleur score est 15 points[6]. Il cherche alors le 7^{eme} et considère les réponses adverses. Il va donc boucler sur ces réponses adverses et trouver, disons, les scores suivant :

$$-20, -22, -15, -16, -11, -18, -20, -30, -70, -75$$

Maintenant le meilleur de ces scores est -11, ce qui donne $-(-11) = 11$ au niveau de la racine. Ce score est moins bon que les 15 points trouvés précédemment. Pourtant, le programme a perdu son temps à chercher les réponses adverses numéro 6 à 10. En fait, dès qu'une réponse adverse marque -11 points, nous savons que ce coup ne pouvait pas battre le score 15 précédent. Quels que soient les scores adverses suivant (ici $-18, -20, -30, -70, -75$) nous savons que ce noeud n'est qu'une menace non crédible (le premier joueur n'a pas intérêt à y aller).

2.1.2 Alpha-beta pruning

Concrètement, à chaque noeud est associé un intervalle $[\alpha, \beta]$ qui correspond à la fourchette de score admissible (tout score extérieur à $[\alpha, \beta]$ est réfuté en amont de l'arbre). A partir de là, le code devient tout simple[7] :

```
search(depth, alpha, beta)
{
    if (no legal move) return in_check ? -MATE : 0 // mate or stalemate
    if (in check, recapture, 1 legal move, pawn-push to 7th rank) depth++
    loop through all moves {
        play move
        score = depth > 1 ? -search(depth-1,-beta,-alpha) : -quiesce(-beta,-alpha)
```

[5] En anglais, *to prune a tree* = élaguer un arbre, donc *pruning* = élagage.

[6] En pratique, un pion vaut 100 points, ce qui permet de n'utiliser que des entiers pour coder le score.

[7] On ne manquera pas de constater que la technique s'applique de la même façon au quiescent search, ce qui explique l'utilisation d'une fourchette $[-\beta, -\alpha]$ dans la fonction quiesce.

```
        undo move
        alpha = max(alpha, score)
        if (alpha >= beta) return alpha
    }
    return alpha
}
```

Concrètement, l'amélioration est liée à : `if (alpha >= beta) return alpha`. On appelle cela un β-cutoff. S'il y a typiquement 40 coups légaux à chaque noeud et que le β-cutoff apparaît, en moyenne, au b^{eme}, alors la complexité passe de 40^n à b^n, ce qui n'est pas du tout négligeable.

2.1.3 Tri des coups et SEE

Pour minimiser ce nombre b, il faut donc que les meilleurs coups se situent au début de la liste. Ainsi les β-cutoff apparaissent plus tôt et les coups sont cherchés avec des fenêtres $[\alpha, \beta]$ plus étroites. La qualité du tri est un point crucial de l'algorithme, comme le souligne [5] : *«The way to speed up a chess program very rarely lies with extremely fast move generation, nor does it lie with rapid check testing. The best way to achieve immediate and noticeable speed increases is to improve the move ordering»*. Notre tri est largement inspiré de [6] :

1. «bonnes» captures : il s'agit de captures initiant une séquence de recaptures gagnante.
2. échecs, déplacements vers le centre, puis autres déplacements.
3. «mauvaises» captures : captures initiant une séquence de recaptures perdante.

SEE. Pour départager les bonnes des mauvaises captures, nous utilisons une technique classique appellée SEE[8]. Le SEE évalue simplement toutes les suites de recaptures possibles, afin d'estimer la valeur immédiate d'une capture : il s'agit d'une approximation rapide du quiescent search. La question que cherche à résoudre le SEE est donc la suivante : la séquence de recaptures qu'initie une capture est-elle perdante ou gagnante pour le camp qui fait la capture initiale ?
Afin de répondre à cette question, les coups qui seront envisagés par le SEE ne seront que des recaptures sur la case initiale de l'attaque – que nous appellerons «cible». On construit les listes de pieces qui attaquent et défendent la case «cible» de l'échiquier, puis les joueurs vont jouer chacun leur tour :

1. Si le joueur actif possède encore une pièce pouvant capturer la case cible : il réalise cette capture avec la pièce de plus faible valeur. On doit ensuite ajouter à la liste des attaquants, défenseurs, une pièce éventuelle qui peut désormais accéder à la case «cible».
2. Si le joueur actif est incapable de re-capturer la case cible, alors l'algorithme s'arrête, et l'on dépile la séquence de capture en sens inverse, afin de savoir à chaque étape s'il vaut mieux pour le joueur actif faire la re-capture ou non.

Il y a tout de même une autre issue à la boucle sur les joueurs : si l'un des joueurs fait une capture avec le roi et que son adversaire possède encore une pièce pouvant le re-capturer, il est évident que la capture faite avec le roi était illégale. Voyons maintenant deux exemples, afin de comprendre comment fonctionne le SEE et quelles sont ses faiblesses.

Si l'on examine la promotion en dame du pion d (Fig. 4 à gauche), la séquence de captures sera la suivante : **1. d8=Q+** ♘×**d8; 2. e×d8+** ♘×**d8; 3.** ♗×**d8+**. En effet, reprendre à la tour A8 ne sert à rien puisqu'elle sera ensuite capturée par la dame ou la tour, sans possibilité de re-capture par le roi. Le score renvoyé par le SEE sera donc la valeur de deux cavaliers, moins celle de deux pions. Considérons maintenant l'exemple de la promotion en E8. Il n'y aura aucune séquence de capture (car reprendre à la tour ferait échanger un pion contre une tour), et le score renvoyé sera la valeur d'une dame moins celle d'un pion.

[8] Static Exchange Evaluation : Évaluation Statique des Échanges.

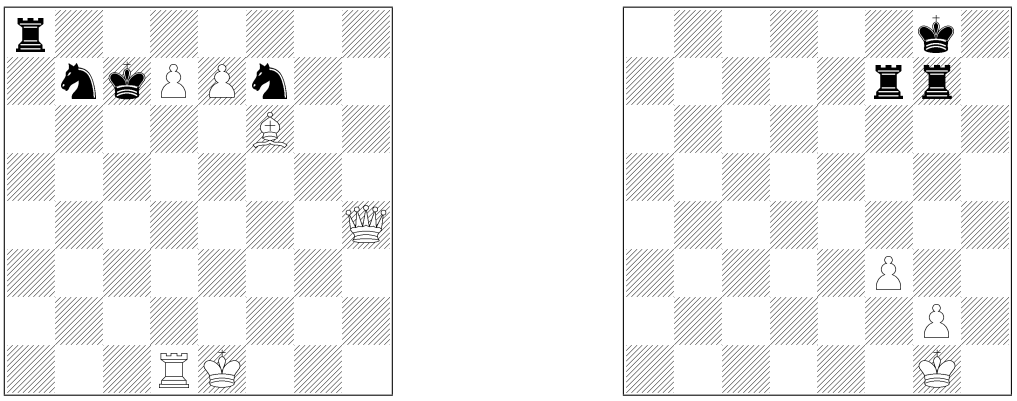


FIG. 4 – Exemples de SEE.

Il s'agit là de cas où l'analyse du SEE est relativement correcte. Voyons maintenant, sur la FIG. 4 à droite, un exemple qui met le SEE en défaut. Le score renvoyé par la SEE pour la prise du pion F par la tour sera désavantageux pour noir, car le SEE ne prête pas attention aux clouages, donc il va considérer que le blanc pourrait prendre la tour au pion G. Le score serait le même si le roi blanc était remplacé par une dame blanche. Bien que ce deuxième coup soit légal, il entraînerait la perte de la dame, chose qui n'est pas vu par le SEE dans la mesure où cela se produit ailleurs que sur la case de re-capture F3.

Le SEE n'est aucunement une évaluation exacte des échanges comme nous venons de le voir. Mais c'est cette simplification du problème qui fait sa force. Considérant des règles plus simples que celles des échecs, il peut résoudre rapidement les séquences de capture sur une case unique, sans que nous ne générions les coups légaux.

2.1.4 Le PVS

Le *Principal Variation Search* (PVS) est un raffinement de l'*alpha-beta pruning*. Le postulat de départ est que l'on dispose d'une bonne fonction de tri des coups. En particulier, le premier est fréquemment le meilleur. Dans ce cas, il n'est souvent pas nécessaire de chercher les coups suivant, de façon aussi exhaustive que le premier. Plus précisément, on se fiche de savoir leur score exact, tant que celui-ci est $\leq \alpha$. Ainsi, nous cherchons le premier coup correctement, et les suivant sont d'abord cherchés avec une fenêtre $[\![\alpha, \alpha + 1]\!]$:

```
if (first_move)   // do the first one thoroughly
    score = -search(depth-1, -beta, -alpha)
else {  // try zero window first
    score = -search(depth-1, -(alpha+1), -alpha)
    if (score > alpha && beta > (alpha+1))  // fail high: current move improves alpha
        score = -search(depth-1, -beta, -alpha) // PVS re-search
}
```

Si notre hypothèse s'avère exacte et que le meilleur coup est le premier de la liste, alors la recherche des coups suivant sera beaucoup plus rapide. En revanche, si un des coups suivant améliore α, un β-cutoff sur la fourchette $[\![\alpha, \alpha+1]\!]$ – en principe rapide – indique que ce coup est meilleur que α. Comme son score est tronqué supérieurement (par $\alpha + 1$) il faut le re-chercher correctement, c'est-à-dire dans toute la fourchette $[\![\alpha, \beta]\!]$.

En principe, le gain de temps lié aux recherches «fructueuses» sur des fenêtres étroites $[\![\alpha, \alpha+1]\!]$ contrebalance la perte de temps occasionnelle des *PVS re-search*. Dans la pratique, le premier coup testé est celui de la Hash Table (voir 2.2), s'il y en a un.

2.2 Hash Tables

2.2.1 Motivations

Transpositions. L'exploration d'un arbre de coups aux échecs, même avec un algorithme PVS, est un calcul très redondant. En effet, diverses trajectoires (de la racine aux feuilles) se recoupent, passant ainsi par les même positions. Il serait donc bien pratique, lorsque l'on a déjà cherché la position courante à la profondeur requise, de renvoyer directement le résultat préalablement stocké, ce qui permettrait d'élaguer l'arbre d'avantage.

Approfondissement itératif. Il existe aussi une autre bonne raison de stocker le travail effectué lors de la recherche : un programme joue avec une contrainte de temps, et pas de profondeur. En effet, il est difficile de prédire qu'avec 10 secondes, il atteindra telle ou telle profondeur selon le niveau de complexité de la position. Il faut nécessairement augmenter la profondeur de façon itérative, en commençant disons par 2 (ce qui est toujours très rapide). Dès qu'il ne reste plus de temps, il faut renvoyer le coup trouvé à la dernière profondeur explorée entièrement. De manière assez contre-intuitive (voir [5]), chercher les profondeurs $2, 3, 4, 5$ en remplissant au fur et à mesure une *hash table*, est plus rapide que de chercher directement la profondeur 5.

2.2.2 Principe d'une Hash Table

L'idée d'une hash table est assez simple : on stocke en mémoire tout ce que l'on calcule (position, profondeur, score et meilleur coup trouvé). Enfin, avant de chercher quelque position que ce soit, on consulte la hash table, pour voir si la position n'a pas déjà été cherchée à la profondeur requise (ou plus loin). Cela permet d'éviter le calcul redondant des transpositions : diverses trajectoires (de la racine aux feuilles) se recoupent souvent (i.e. passent par la même position). Statistiquement, il a été montré que ce phénomène augmente avec la profondeur.

D'un point de vue pratique, stocker et chercher une position doit être une opération rapide et les entrées d'une hash table doivent utiliser le moins de mémoire possible. Nous utilisons donc un tableau, de taille 2^N, avec recherche et insertion **immédiates**[9]. Cela suppose donc de disposer d'un ordre sur les positions : il faut construire une application – aussi injective que possible – qui à une position associe un nombre, facilement manipulable en `C++` : on appelle cela une signature.

Il existe plusieurs algorithmes de signature génériques (i.e. pour tout type de données), tels que CRC 32 ou MD5, utilisés typiquement pour vérifier qu'un fichier n'a pas été corrompu. Cependant, pour les programmes d'échec, il existe une méthode très simple et plus spécifique : les clefs de Zobrist sur 64 bits (voir [1]). Il s'agit, en effet, de la méthode utilisée par tous les bons programmes.

2.2.3 Clef de Zobrist

Il s'agit d'une méthode simple et efficace de signature des positions. L'«efficacité» signifie, bien entendu, que la probabilité que deux positions distinctes aient même clef de Zobrist est infime et peut être négligée. Pour commencer, on remplit un tableau à 3 entrées (pièce, case, couleur) par des nombres aléatoires sur 64 bits :

```
unsigned __int64 zobrist[piece][square][color]
```

Partant de `Z=0`, on boucle sur les 64 cases en effectuant l'opération

```
Z ^= zobrist[piece][square][color]
```

dès que l'on rencontre une pièce `pièce` de couleur `color` sur la case numéro `square`. Toujours dans la même idée, on génère 64 nombres aléatoires pour la case de prise en passant, 2 pour le

[9] La position d'un clef est donné par ses N bits de poids faible.

tour de jeu et 16 pour les 2^4 combinaisons possibles de permissions de roquer. De la même façon, tout ce qui doit l'être est mis dans la clef `Z`, à base de l'opération XOR : et voila notre clef.

Une propriété intéressante des clefs de Zobrist est la possibilité des les calculer de façon dynamique. Partant de la règle de calcul

```
(x ^ y) ^ y = x ^ (y ^ y) = x ^ 0 = x
```

on peut mettre à jour la clef, sans re-parcourir tout l'échiquier. Par exemple, partant de la position de départ de clef `Z`, après **1. e4**, il n'y a qu'à enlever le pion blanc de la case e2 pour le mettre en e4 :

```
Z ^= zobrist[Pawn][E2][White] ^ zobrist[Pawn][E4][White]
```

Il faut aussi mettre à jour, de la même façon, la permission de roquer, la case de prise en passant (désormais e3) et le tour de jeu (c'est maintenant à noir de jouer). Cependant, pour des raisons de simplicité et de robustesse, nous n'utilisons pas cette astuce.

2.3 Heuristiques d'élagage

2.3.1 Null-move pruning

Nous avons vu que le PVS permet de réduire sensiblement la taille de l'arbre à explorer. Cependant, il ne s'agit pas d'une limite absolue d'élagage : il est possible d'aller plus loin, si l'on s'autorise quelques «erreurs». Le *null-move pruning* est une méthode utilisée avec beaucoup de succès, dans tous les bons programmes d'échecs modernes.

Cet algorithme ne fait qu'appliquer un concept simple, que les joueurs humains utilisent naturellement et sans le savoir. L'idée est la suivante : si vous choisissez de passer votre tour – laissant l'adversaire jouer 2 coups consécutifs – et que le score adverse est toujours mauvais, alors votre position est clairement supérieure. En termes plus précis maintenant : si le joueur actif passe son tour et le score adverse calculé à une profondeur réduite $d - R$ est inférieur à alpha, alors on renvoie simplement cette valeur, sans chercher exhaustivement le sous-arbre considéré à profondeur d. En fait, nous ne faisons que prédire un β-cutoff sans vraiment le vérifier. BibiChess utilise cette méthode avec une réduction systématique de $R = 3$: nous avons pris la même valeur que [7].

Cependant, cette méthode peut s'avérer dangereuse dans les cas de *zugzwang*[10]. C'est pourquoi BibiChess n'effectue pas de *null-move* lorsque le joueur actif n'a plus de pièce, c'est-à-dire juste un roi et des pions. Cette condition de sécurité est toutefois un peu faible. Les meilleurs programmes prennent plus de précautions, en ce qui concerne les null-move :

1. la réduction de profondeur R augmente avec la profondeur du sous arbre à chercher.
2. cette même réduction baisse lorsque l'on est en finale avancée.
3. dès qu'un β-cutoff est prédit par un null-move, on lance un algorithme de vérification de zugzwang – connu sous le nom de *verification search*.

Afin de ne pas alourdir l'exposé, nous ne parlerons pas de cet algorithme, que BibiChess n'utilise pas.

2.3.2 Delta pruning

Le *delta pruning* est une méthode d'élagage approximatif, qui s'applique au niveau du *quiescent search*. Nous n'avons pas trouvé de référence théorique à ce sujet, mais deux codes source [6] et [7], qui procèdent tous les deux différemment. Nous avons choisi la méthode appliquée par [6]. Le delta pruning ne s'applique bien sûr pas lorsque la position est un échec. Pour tous les autres noeuds du *quiescent search*, on boucle sur les coups et :

[10] On appelle ainsi, une position dans laquelle le joueur actif a intérêt à passer son tour – ce qui n'est bien sûr pas possible. Ce phénomène se produit souvent en finale roi+pions.

1. On effectue une évaluation optimiste du quiescent search, en ajoutant le SEE du coup et l'évaluation statique de la position.
2. Si cette valeur trop faible, disons $\leq \alpha - \delta$, alors on saute ce coup sans le développer récursivement.

Nous avons choisi $\delta = 100 = \♙$, ce qui est une valeur peu élevée, donc un peu risquée.

2.3.3 Futility pruning

Le *futility pruning* s'effectue à un coup du quiescent search. L'idée est assez simple : s'il n'y a pas échec, le coup qui doit être cherché n'est pas un coup tactique – capture échec promotion –, et le score actuel augmenté d'une certaine marge de sécurité est $\leq \alpha$, alors ce coup est très probablement mauvais. Dans ce cas, au lieu de le chercher à profondeur 1, on le cherche à profondeur 0, c'est-à-dire directement avec le quiescent search.

Bien entendu, cette méthode n'est pas fondée théoriquement, et peut manquer des variantes tactiques intéressantes. En pratique, le gain apporté par cette méthode compense largement ce défaut. Après tout, si l'on gagne disons une profondeur, alors on évite les erreurs tactiques liées au *futility pruning* et on gagne en tactique.

La méthode s'étend au noeuds situés à 2 coups du *quiescent search*. Le principe est le même, mais cette fois, il faut prendre une marge de sécurité plus importante. Si le score augmenté de la marge de sécurité est toujours $\leq \alpha$, alors on réduit la profondeur de 1. Ainsi, au lieu de chercher le coup à profondeur 2, on le cherche à profondeur 1.

3 Implémentation

Dans cette partie, nous ne présentons pas tout le code, mais uniquement des morceaux choisis. Il s'agit en l'occurrence de la génération des coups par la méthode des bitboard, ainsi que l'évaluation positionnelle – évoquée en 1.1.2.

3.1 Les BitBoards

3.1.1 Choix des structures

Dans un premier temps, nous avons programmé la génération des coups d'une façon simple et naturelle. Il s'agissait d'utiliser un tableau de 64 éléments correspondant aux 64 cases de l'échiquier. La génération des coups se faisait nécessairement de la façon suivante :

1. on boucle sur les 64 cases de l'échiquier.
2. lorsque l'on trouve une pièce (de la bonne couleur), on boucle sur l'ensemble de ses directions possibles.
3. enfin, si la pièce est une tour une dame ou un fou, il faut en plus boucler dans la direction considérée – en s'arrêtant lorsque l'on rencontre une pièce ou le bord de l'échiquier.

Ainsi, nous avons 3 boucles imbriquées ce qui est très lent. Le même schéma se répète d'ailleurs dans l'évaluation de la mobilité (voir 3.2.2). Il ne faut pas perdre de vue en effet, que le programme passe la majeure partie de son temps dans la génération des coups et dans l'évaluation positionnelle. Nous avons ainsi obtenu une version de mi-parcours du programme, que nous avions appelé NewbieChess – à cause de son faible niveau. Afin d'augmenter ses performances, mais aussi pour l'exercice technique de programmation, nous avons choisi de réécrire entièrement la génération des coups ainsi que l'évaluation, en utilisant une méthode bien plus efficace et infiniment plus élégante.

BibiChess utilise une méthode sophistiquée pour la génération de coups légaux, ainsi que toute sorte de calculs relatifs à l'échiquier. Il s'agit en gros, de représenter l'échiquier par des nombres 64 bits, ce qui permet de faire toute sorte de choses avec des opérateurs *bitwise*, soit `& | ^ ~ << >>`. En particulier, des quantités de boucles sont évitées, et tout simplement remplacées par des calculs immédiats à base d'opérateurs bitwise.

3.1.2 Représentation de l'échiquier

Commençons par le commencement, il faut d'abord numéroter les cases de l'échiquier :

A8	B8	C8	D8	E8	F8	G8	H8
A7	B7	C7	D7	E7	F7	G7	H7
A6	B6	C6	D6	E6	F6	G6	H6
A5	B5	C5	D5	E5	F5	G5	H5
A4	B4	C4	D4	E4	F4	G4	H4
A3	B3	C3	D3	E3	F3	G3	H3
A2	B2	C2	D2	E2	F2	G2	H2
A1	B1	C1	D1	E1	F1	G1	H1

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

FIG. 5 – Numérotation des cases de l'échiquier

ainsi que les pièces et les couleurs :

```
enum { White, Black };
enum { Knight, Bishop, Rook, Queen, King, Pawn, Empty };
```

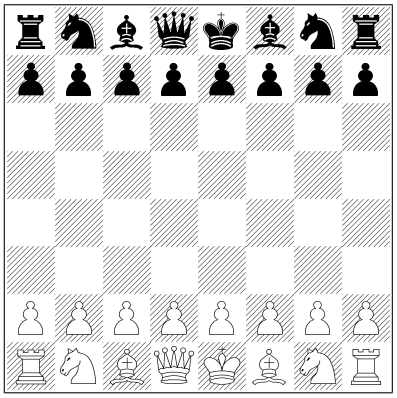
En clair, les cases sont numérotées de bas en haut et de gauche à droite. Les rangées sont numérotées de 0 à 7 (de bas en haut) et les colonnes de 0 à 7 (de gauche à droite). Ainsi, pour toute case `A1 <= sq <= H8`

```
file(sq) = sq & 7    et    rank(sq) = sq >> 3
```

nous donnent respectivement la colonne et la rangée de `sq`. Maintenant l'échiquier est représenté par des nombres 64 bits de la façon suivante :

```
typedef unsigned __int64 U64;
U64 Board::b[2][64];
```

Chaque élément du tableau `b` doit être considéré comme un ensemble : les opérateurs `& | ^ ~` correspondent alors aux opérations ensemblistes d'intersection, de réunion, de différence symétrique et de complémentation. Par exemple, `b[White][Knight]` représente l'ensemble des cavaliers blancs sur l'échiquier, etc... Ainsi, la position de départ, encodée en notation bitboard, fait l'objet de la Fig. 6. Pour simplifier les notations, nous utilisons un tableau pré calculé `U64 bit[64]`, dont le i-ème élément est tout simplement `bit[i] = U64(1) << i`.



```
b[White][Knight] = bit[B1] | bit[G1];
b[White][Bishop] = bit[C1] | bit[F1];
b[White][Rook]   = bit[A1] | bit[H1];
b[White][Queen]  = bit[D1];
b[White][King]   = bit[E1];
b[White][Pawn]   = 0xFF << A2;
b[Black][Knight] = bit[B8] | bit[G8];
b[Black][Bishop] = bit[C8] | bit[F8];
b[Black][Rook]   = bit[A8] | bit[H8];
b[Black][Queen]  = bit[D8];
b[Black][King]   = bit[E8];
b[Black][Pawn]   = 0xFF << A7;
```

Fig. 6 – Position initiale et représentation bitboard correspondante.

Nous concentrons maintenant l'exposé des bitboard sur la génération des coups légaux. Bien entendu, il y a des tas de fonctions qui utilisent les bitboard, mais la génération des coups légaux est la partie la plus intéressante. C'est véritablement là que l'astuce prend tout son intérêt.

3.1.3 Génération des coups pseudo-légaux

Nous appelons pseudo-légal un coup presque légal, à ceci près que l'on a pas encore testé s'il nous mettait en échec. Autrement dit, ce sont des coups géométriquement légaux : un cavalier se déplace en «L», un fou se déplace en diagonale sans traverser personne etc... Nous signalons, pour faciliter la lecture du code, que nous les appelons *lazy moves* dans les noms de variables et commentaires – en anglais. Nous allons voir comment, grâce aux bitboard, la génération des coups pseudo-légaux est une opération très rapide. Selon le type de pièce, bien entendu, nous procédons différemment : voyons tout cela, en allant du plus simple (roi et cavaliers) au plus compliqué (fous et dames).

Roi et Cavaliers. On utilise ici des bitboards pré calculés `knight_moves[64]` et `king_moves[64]`, qui fournissent directement l'ensemble des cases attaquées par un roi ou un cavalier, en fonction de la case qu'il occupe. Par exemple, un roi en `E1` attaque les cases `{D1,F1,D2,E2,F2}` et un cavalier en `B1` attaque les cases `{A3,C3,D2}`, donc

```
king_moves[E1]   = bit[D1] | bit[F1] | bit[D2] | bit[E2] | bit[F2];
knight_moves[B1] = bit[A3] | bit[C3] | bit[D2];
```

Maintenant, pour générer les coups pseudo-légaux des cavaliers blancs, on procède comme suit :

```
int fsq, tsq;   // from square, to square
U64 fss = b[White][Knight], tss;  // from squares, to squares
while (fss)
{
    next_bit(fsq, fss);       // fsq parcours les cavaliers blancs
    tss = knight_moves[fsq]   // cases atteignables par le cavalier considéré
        & ~friends;           // et qui ne contiennent pas de pièces "amies"
    while (tss)
    {
        next_bit(tsq, tss);   // tsq parcours les cases attaquées par ce cavalier
        ...     // générer ici le coup de cavalier fsq -> tsq
    }
}
```

Bien sûr, c'est encore plus simple dans le cas du roi : il n'y a pas la boucle extérieure sur `fss`, puisque chaque joueur n'a qu'un seul roi.

Pions. Pour générer les poussées de pions blancs d'une ou deux rangées, nous devons juste les avancer de 8 ou 16 cases, étant donné la numérotation choisie. Ainsi, avec le bitboard pré calculé `rank_mask[8]` correspondant aux rangées (numérotées de 0 à 7), voila comment nous pouvons générer les poussées de pions blancs :

```
U64 fss = b[White][Pawn], tss;  // from squares, to squares
tss = (fss << 8) | (((fss & rank_mask[1]) << 16) & ~(all_pieces << 8));
tss &= ~all_pieces;             // all_pieces contient toutes les pièces
```

En particulier, vous remarquerez que l'interdiction d'écraser une pièce en poussant un pion s'écrit `tss &= ~all_pieces`, et l'interdiction de sauter par dessus une pièce pour une double poussée correspond au `... & ~(all_pieces << 8)`. Nous obtenons ainsi les cases d'arrivé dans le bitboard `tss`. Ensuite, il faut boucler sur `tss` et voir d'où peut provenir la poussée correspondante (il y en a nécessairement 1 ou 2 pour chaque bit de `tss`). Enfin, pour générer les coups il faut distinguer les promotions des autres poussées, avec un test du genre

```
if (rank_mask[7] & bit[tsq])  // teste l'appartenance de tsq à rank_mask[7]
    // générer les promotions fsq ->tsq = Q, N, R, B
else
    // générer une poussée normale fsq -> tsq
```

Les captures se gèrent de façon similaire, avec une «formule» de calcul des *to-squares* `tss` :

```
can_capture = enemies | (ep_square ? bit[ep_square] : 0);
fss = b[White][Pawn];
tss = (((fss & ~file_mask[0]) << 7) & can_capture)
    | (((fss & ~file_mask[7]) << 9) & can_capture);
```

Rangées. Pour les déplacements en ligne, on commence par récupérer l'occupation de la ligne. Il s'agit de 8 bits, dans l'ordre de lecture sur l'échiquier. Par exemple, l'occupation de la rangée 3 dans la position de la Fig. 7 à gauche, est `b01000101`. Vous remarquerez bien que ce nombre est écrit en notation binaire inversée, c'est-à-dire en commençant par le bit de poids faible. Nous mettrons dans ce cas un préfixe `b`, pour rester cohérent avec la notation standard (du moins en assembleur) suffixe, qui commence par le bit de poids fort. Autrement dit, `b01000101 = 10100010b = 0xA2`.
Pour récupérer l'occupation de la ligne contenant la case `fsq` d'une tour, rien de plus facile !

```
U64 occupancy = (all_pieces >> (rank(fsq) << 3)) & 0xFF;
```

Une fois que l'on dispose de cette occupation, on aimerait bien savoir directement où peut aller (en ligne) une tour en `fsq`, étant donné cette occupation. Nous utilisons pour cela un tableau pré calculé :

```
U64 rank_slides[8][256];
```

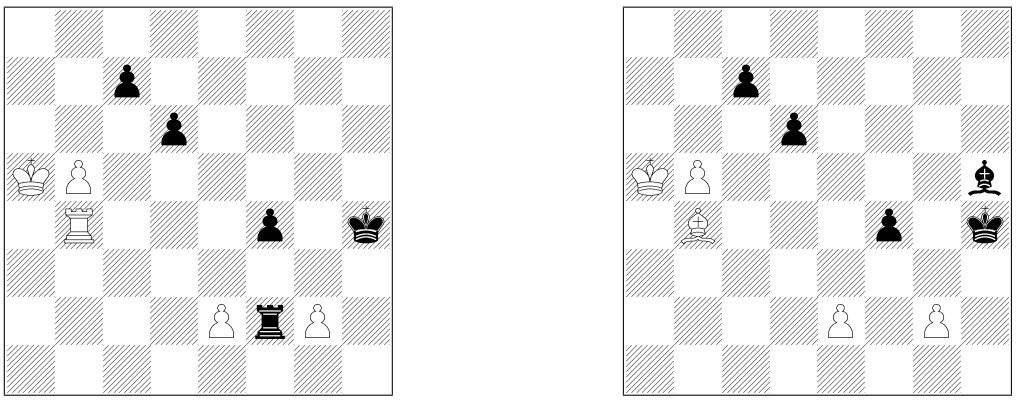


FIG. 7 – Génération des coups de tour (à gauche) et de fou (à droite).

Concrètement nous avons, pour la tour en `B4` (toujours sur la FIG. 7 à gauche), nous avons :

```
rank_slides[file(B4)][b01000101] = b10111100 = 0x3D
```

Enfin, il n'y a plus qu'a décaler ce `b10111100`, pour le remonter sur la rangée 3. Plus généralement, on a (avec des notation désormais standard...) :

```
tss = rank_slides[file(sq)][occupancy] << (rank(sq) << 3);
```

Colonnes. En ce qui concerne les déplacements en colonne, on procède de la même façon, sauf que l'occupation d'une colonne est plus compliquée à obtenir. Il faut appliquer une symétrie à `all_pieces`, par rapport à `a1h8`. Par exemple, toujours sur la FIG. 7 à gauche, nous procédons ainsi pour récupérer l'occupation de la colonne `F`, afin de générer les coups de la tour noire en `F2` :

all_pieces	all_pieces_sym	>> 8*file(F2)	& 0xFF
0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0	0 1 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 1	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0	0 0 0 1 1 0 0 0	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0	0 1 0 1 0 0 0 0	0 1 0 1 0 0 0 0

FIG. 8 – Occupation de la colonne F, sur la FIG. 7 à gauche.

On récupère donc l'occupation de la colonne `F` correspondant à un sens de lecture de bas en haut. Cette fois, la donnée précalculée qu'on utilise s'appelle `U64 file_slides[8][256]`. Le résultat est alors donné sur la colonne `A`; il faut donc le décaler pour se remettre en `F`. Concrètement,

```
file_slides[rank(F2)]
  [b01010000]             << file(F2)
--------------------  --------------
0 0 0 0 0 0 0 0       0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0       0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0       0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0       0 0 0 0 0 0 0 0   =>  tss = bit[F1] | bit[F3] | bit[F4]
1 0 0 0 0 0 0 0       0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0       0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0       0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0       0 0 0 0 0 1 0 0
```

Diagonales. Toujours dans la même idée, il faut récupérer l'occupation de la diagonale considérée, et utiliser une table précalculée. Cette fois, le problème se complique encore, puisqu'il y a deux types de diagonales (sens `A1H8` ou `A8H1`) et qu'elles ont toutes des longeurs différentes, comme le montre la (Fig. 9).

```
            A1                                  A8
          A2  B1                              B8  A7
        A3  B2  C1                          C8  B7  A6
      A4  B3  C2  D1                      D8  C7  B6  A5
    A5  B4  C3  D2  E1                  E8  D7  C6  B5  A4
  A6  B5  C4  D3  E2  F1              F8  E7  D6  C5  B4  A3
A7  B6  C5  D4  E3  F2  G1          G8  F7  E6  D5  C4  B3  A2
A8  B7  C6  D5  E4  F3  G2  H1  H8  G7  F6  E5  D4  C3  B2  A1
  B8  C7  D6  E5  F4  G3  H2          H7  G6  F5  E4  D3  C2  B1
    C8  D7  E6  F5  G4  H3              H6  G5  F4  E3  D2  C1
      D8  E7  F6  G5  H4                  H5  G4  F3  E2  D1
        E8  F7  G6  H5                      H4  G3  F2  E1
          F8  G7  H6                          H3  G2  F1
            G8  H7                              H2  G1
              H8                                  H1
```

Fig. 9 – Diagonales dans le sens A1H8 (à gauche) et A8H1 (à droite).

Pour récupérer l'occupation d'une diagonale, il nous faut donc le bitboard `all_pieces` renuméroté dans le sens `A1H8` et `A8H1`, comme sur la Fig. 9. Toujours à l'aide d'un décalage et d'un filtrage des bits de poids faible (`>>` et `&`), on récupère les occupations voulues. Enfin, les tables pré calculées suivantes nous donnent directement les *to-squares* `tss` que l'on cherche à calculer :

```
U64 diag_a1h8_slides[64][256], diag_a8h1_slides[64][256];
```

On terminera donc par `tss = diag_a1h8_slides[square][occupancy]`, après avoir calculé l'occupation de la diagonale considérée.

Dans l'exemple de la figure Fig. 7 à droite, pour générer les coups du fou blanc en `B4`, nous procédons ainsi :

1. les occupations des diagonales f8-b3 et a5-e1 sont `b001010` et `b11000`.
2. les bitboards précalculés :

   `diag_a1h8_slides[B4][b001010]` et `diag_a8h1_slides[B4][b11000]`

   nous donnent alors les cases sur lesquelles le fou `B4` peut aller.

3.1.4 Accélération du générateur

Implémentation Robuste. Pour générer les coups légaux, la méthode la plus simple est de générer les coups pseudo-légaux, de tous les jouer (et les défaire), afin de voir lesquels sont illégaux – par auto-échec – et lesquels mettent l'adversaire en échec. Nous avons en effet besoin de savoir directement quel coup donne échec, pour la partie tactique du code `Computer::search` – qui effectue des extensions de recherche, voir 1.2.2. C'est ainsi que nous avons commencé, jusqu'à ce que l'on profile le code pour s'apercevoir du temps perdu dans les fonctions `Board::play`, `Board::undo` et `Board::in_check` – jouer, défaire les coups et tester la mise en échec. Pour fixer les idées, disons qu'avec 40 coups pseudo-légaux dans une position donnée, nous effectuions 40 fois `play` et `undo` et 80 fois `in_check`. De mémoire, il me semble que la génération des coups pseudo-légaux n'occupait que 2.3% du temps de parcours d'un arbre moyen ! N'est-ce pas dommage de se casser la tête avec des bitboards pour tout gâcher ainsi ?

Actualiser dynamiquement ce qui doit l'être. En se laissant guider par les résultats du profile de `Visual C++`, nous avons vite compris que l'on perdait trop de temps dans le calcul

de la clef de Zobrist et des 3 transformées géométriques de l'occupation globale de l'échiquier, soit `all_pieces_sym`, `all_pieces_a1h8` et `all_pieces_a8h1`. En effet, chacun de ces calculs nécessite une boucle de 64 itérations, et ce à chaque utilisation de la fonction `play` – soit environ 40 fois plus que de noeuds (non terminaux) de l'arbre exploré, d'après la remarque précédente. Nous avons donc décidé de calculer dynamiquement la clef de Zobrist, voir 2.2.3 pour plus d'explications. De la même façon, au lieu d'effectuer les transformations géométriques sur tout le bitboard `all_pieces`, nous ne faisons que des actualisations concernant les bits effectivement déplacés. En tout et pour tout, cette amélioration multiplie, en moyenne, par 2.6 la vitesse d'exploration d'un arbre de coup. Ce gain est d'autant plus appréciable qu'il est peu coûteux, en termes de lignes de code supplémentaires.

Filtrage des coups pseudo-légaux. Jouer, tester deux fois la mise en échec et défaire les coups pseudo-légaux, est une méthode bien trop laborieuse. Pour éviter cette perte de temps, voila comment nous procédons :

1. Si le joueur actif est en échec, nous utilisons l'implémentation robuste précédemment décrite. Seule une faible proportion des noeuds de l'arbre sont des échecs. Ainsi, un code spécifique pour ce cas n'est pas vraiment nécessaire. Cependant, les bons programmes d'échec disposent tous d'une fonction `generate_check_escapes` qui génère directement les coups légaux lorsque l'on est en échec.
2. Dans tous les autres cas, soit une immense majorité des noeuds de l'arbre, nous commençons par calculer 2 bitboards : `pinned_pieces` et `rvl_check`, avec la fonction `Board::find_pins`. Il s'agit respectivement de l'ensemble des pièces clouées et des pièces pouvant révéler un échec à découvert. A partir de là, nous bouclons sur les coups pseudo-légaux :
   (a) Les prises en-passant et roques sont des cas particuliers, pour lesquels aucune optimisation n'est effectuée. On utilise donc la méthode laborieuse (`play`, `undo` et `in_check`). N'oublions pas toutefois que ces coups sont très rares dans l'arbre des coups.
   (b) Pour les coups de roi, on teste la légalité avec un `square_attacked(move.tsq,!turn)`, qui vérifie directement si la case d'arrivée est attaquée par l'adversaire. La mise en échec de l'adversaire par un coup de roi (autre qu'un roque) ne peut résulter que d'une attaque à découvert, cachée par le roi déplacé. On teste donc si la case de départ `move.fsq` appartenait à `rvl_check` : si non alors pas d'échec, et si oui, alors le coup est un échec si et seulement si le sort du rayon de clouage correspondant.
   (c) Pour les autres coups, c'est-à-dire le cas général, l'illégalité ne peut résulter que d'un auto-échec à découvert. En termes plus didactiques, un tel coup est légal si et seulement si la pièce déplacée n'est pas clouée ou bien si elle reste sur son rayon de clouage. En ce qui concerne la mise en échec de l'adversaire, un échec ne peut être que direct ou révélé[11]. Pour un échec direct, on regarde si le roi adverse est attaqué par la pièce que l'on dépose[12] sur la case d'arrivée `move.tsq`. Pour un échec à découvert, on teste l'appartenance de `move.fsq` à `rvl_check` : si non, le coup ne donne pas échec; si oui, le coup donne échec si et seulement si `move.tsq` sort du rayon de clouage correspondant.

Grâce à ce calcul des pièces clouées et pouvant révéler un échec, nous avons encore gagné un facteur 4.3 e, moyenne. Au total, les optimisations du générateur l'accélèrent donc d'un facteur $4.3 \times 2.6 = 11.2$!

[11] Ces deux cas ne sont pas mutuellement exclusifs : un échec à la fois direct et révélé est précisément un double échec.

[12] Qui est, en général la pièce déplacée... sauf en cas de promotion !

3.2 Evaluation positionnelle

3.2.1 Introduction

Aux échecs, deux domaines sont généralement distingués : la tactique et la stratégie. Alors que la tactique correspond à la capacité du joueur à calculer exactement les séquences de coups gagnantes, la stratégie considère plutôt la construction de long terme du jeu : développer ses pièces et «avancer» selon un plan. Par plan, on entend aussi bien des choses simples, comme le contrôle du centre, que des choses plus compliquées comme la construction d'une attaque sur le roi averse, ou échanger les pièces afin de profiter d'un avantage dans la structure de pion.

L'un des buts de l'évaluation, doit être d'inciter à la construction de plans crédibles, mais ce n'est pas tout, l'évaluation comprend finalement toute la culture échiquéenne que va posséder le moteur. C'est notamment grâce à elle que le moteur va choisir comment développer ses pièces : par exemple, on peut attribuer un malus à chacune des pièces qui est restée à la position initiale, et qui n'est donc pas développée. Mieux, on peut attribuer un bonus lorsqu'une pièce attaque le centre, ainsi le moteur sera non seulement incité à développer ses pièces, mais encore, il préfèrera le faire de façon à contrôler le centre, évitant de placer ses cavaliers sur les bords de l'échiquier.

La difficulté majeure est de construire un ensemble de règles, attribuant des bonus et des malus, qui soit cohérent et le moins redondant possible. Dans l'exemple donné précédemment, un coup de développement de cavalier pourrait apporter un gain relatif à la permission de roquer qu'il permet d'obtenir, un autre pour l'attaque du centre, et un malus s'il se trouve sur le bord. Et le gain final prendra donc en compte chacune des règles que l'on aura énoncées au moteur. Il est donc essentiel de garder à l'esprit les interactions entre ces règles et d'essayer, autant que possible de bien séparer les domaines afin d'éviter les incohérences et les redondances.

3.2.2 Les composantes de l'évaluation

L'évaluation que nous avons programmé se découpe en trois domaines distincts : l'activité des pièces, la structure de pion et la sécurité du roi.

Concernant l'activité des pièces, voici les éléments principaux que nous évaluons :
- La position d'une pièce sur l'échiquier. Par exemple les cavaliers ne devraient pas être sur les bords, le roi préfère ne pas être au centre sauf en fin de partie, et les pions contrôlant le centre sont appréciés.
- La mobilité des pièces. On compte le nombre de cases non-attaquées par les pions adverses sur lesquels la pièce pourrait venir, et pour chaque case on attribue un bonus relatif à la pièce.
- Les schémas. Ils correspondent à l'interaction d'une pièce avec d'autres, appartenant à l'adversaire ou non. Cela comprend par exemple le fait d'avoir une tour bloquée en H à cause d'un roi en F et des trois pions en position initiale F,G,H. Mais aussi le fou blanc bloqué en H7 par un pion adverse en G6.
- Le reste, qui ne correspond ni aux pions, ni au roi, mais que nous n'avons pas su classer dans les catégories précédentes. Cela comprend par exemple le fait de placer une tour en septième rangée ou sur une colonne ouverte.

Nous pouvons déjà voir que certaines de ces règles entrent en conflit ou sont redondantes : avoir une tour sur une colonne ouverte permet non seulement d'accroître la mobilité de la tour, mais aussi d'obtenir le bonus spécifique. Cela n'est pas un problème dans le cas présent, puisque cela est identifié, et l'on réduit donc simplement le bonus spécifique puisque l'aspect mobilité sur une colonne ouverte est déjà pris en compte.

Par «structure de pion», nous entendons les pions doublés, isolés (malus), candidats ou passés (bonus). Mais d'autres notions de la culture échiquéenne pourraient mériter d'être implémentées, tels que les pions faibles, arriérés, les îles ou les chaînes. Pour les pions passés, qui constituent le

but ultime pour un pion, l'évaluation identifie elle-même si l'on peut considérer que la promotion ne pourra être empêchée. Cela n'est bien entendu réalisé que dans les cas simples suivants : l'adversaire n'a plus d'autre pièce que son roi et ses pions, et son roi n'est pas dans le «carré du pion», ou bien notre roi défend à la fois le pion et sa case de promotion. Dans les deux cas la promotion est assurée, et l'on peut attribuer un bonus aussi important que la valeur d'une dame moins celle du pion passé. Cela permet de plus dans ces cas spécifiques de gagner quelques profondeurs dans la vision du jeu, à moindre frais.

Dans la «sécurité du roi», nous avons comptabilisé les attaques sur les cases adjacentes au roi, en n'attribuant des bonus que si plusieurs pièces participent à cette attaque. Aucun point n'est attribué à la défense, par conséquent, lorsqu'une attaque de l'adversaire lui procure trop de points, les seules réponses admises sont d'interposer ses pièces, ou de capturer les pièces attaquant le roi. Cette partie de l'évaluation est la plus sensible et la plus difficile. Notre solution est relativement douteuse, mais s'efforce de prendre en compte cet aspect du jeu.

Finalement, nous ne devons pas perdre de vu que beaucoup de règles sont spécifiques à un contexte positionnel, afin de ne pas attribuer à tort et à travers ces bonus. Le plus important d'entre eux étant ce que l'on nomme habituellement phase de jeu (ouverture, milieu de partie, finale). Entre autres, un roi positionné au milieu de l'échiquier est pénalisant à l'ouverture mais bénéfique en finale; un pion passé a lui bien plus de valeur en finale qu'en ouverture. Le plus important est sûrement que la structure de pion doit être d'autant plus prise en compte que l'on approche de la finale.

3.2.3 Les phases de jeu

Concernant les phases de jeu, il n'existe aucun consensus sur leur définition. Il nous a donc fallu en trouver une. Nous avions initialement défini ces phases de façon stricte et irréversible, et attribué uniquement les bonus relatifs à la phase en cours. Cela pose un problème de continuité dans l'évaluation : si les bonus sont très différents d'une phase à l'autre, le moteur avait l'impression qu'un seul échange lui permettait de passer dans la phase suivante, et pouvait ainsi améliorer nettement son évaluation positionnelle sans que cela n'aie vraiment de sens pour la partie en cours. De plus nous étions désireux de construire des plans, or l'effet sus-cité ne le fait pas, vu son effet très ponctuel.

Une meilleure utilisation des phases de jeu, dont nous avons repris le principe dans le code source de Fruit se base sur l'observation suivante : lorsque chaque joueur possède encore toutes ses pièces, alors nous sommes évidemment dans l'ouverture; lorsque les joueurs n'ont plus que leur roi et des pions, alors il s'agit de la finale. On peut alors concevoir qu'entre ces deux situations extrêmes rien ne soit parfaitement identifié, et que la meilleure façon de procéder consiste à utiliser une évaluation pour l'ouverture, une évaluation pour la finale, et à en faire une somme pondérée par la quantité de pièces (autres que les pions) encore sur l'échiquier.

D'une part cela évite les discontinuités d'évaluation entre les phases, d'autre part cela va ajouter une dimension à la stratégie du moteur. Il est désormais capable d'identifier qu'il a tout intérêt à échanger ses pièces contre celles de l'adversaire, dès que sa structure de pion est meilleure. En effet, ce faisant il va accroître la pondération accordée à l'évaluation en contexte «finale», qui lui est encore plus favorable que celle du contexte «ouverture». Dans cette situation, et même après quelques échanges, le moteur sera toujours désireux de continuer à échanger les pièces, s'assurant ainsi que sa structure de pion gagnante lui permettra de faire une promotion.

Enfin, n'oublions pas qu'il est extrêmement difficile de provoquer l'apparition d'un comportement précis. Les modifications de l'évaluation sont donc à faire très précautionneusement et il est important de chercher à observer leur effet par l'observation de nombreuses parties. Dans la mesure où l'évaluation fait intervenir de nombreux paramètres, plus ou moins arbitraire, il est essentiel de le faire afin de choisir des valeurs raisonnables tenant compte de l'ensemble des

règles. On pourrait aussi être désireux de rendre optimaux ces paramètres, à l'aide d'une méthode systématique telle que les algorithmes génétiques. Il faut néanmoins garder à l'esprit que l'évaluation d'un niveau ELO (qui serait le critère à optimiser) est une chose très longue, rendant cette idée extrêmement difficile à mettre en pratique, voir impossible. De toute façon, dans un premier temps, il est toujours nettement plus instructif d'avoir les commentaires d'un bon joueur, juste sur quelques parties.

4 Tests et résultats

4.1 Validation du générateur de coups

Le générateur de coups légaux est assez délicat à coder, mais surtout à débuguer. Comme toujours, le code ne marche jamais du premier coup et le bug n'est pas où on l'attend. Il faut donc procéder avec méthode. Dans un premier temps, il faut s'assurer que – dans tous les cas – `Board::undo` défait correctement ce que `Board::play` a fait. C'est d'abord dans cet unique but, que nous avons écrit les fonctions d'import/export des positions en notation FEN. A cela, il faut ajouter des tests «à la main» pour chaque type de coups (déplacement, capture, échec, prise en passant, roque, promotion).

La preuve par Perft. Une fois les fonctions `Board::play` et `Board::undo` validées, il faut s'attaquer à la génération des coups légaux. Nous avons d'abord débugé «à la main» cette fonction, et nous avons même eu l'impression qu'elle marchait, jusqu'à ce que nous entendions parler de la fonction `perft` et du projet SMIRF [2]. La fonction `perft` est un étalon, contre lequel on peut vérifier son générateur de coups. Plus précisément, le nombre de coups à différentes profondeurs et dans diverses positions a déjà été calculé.

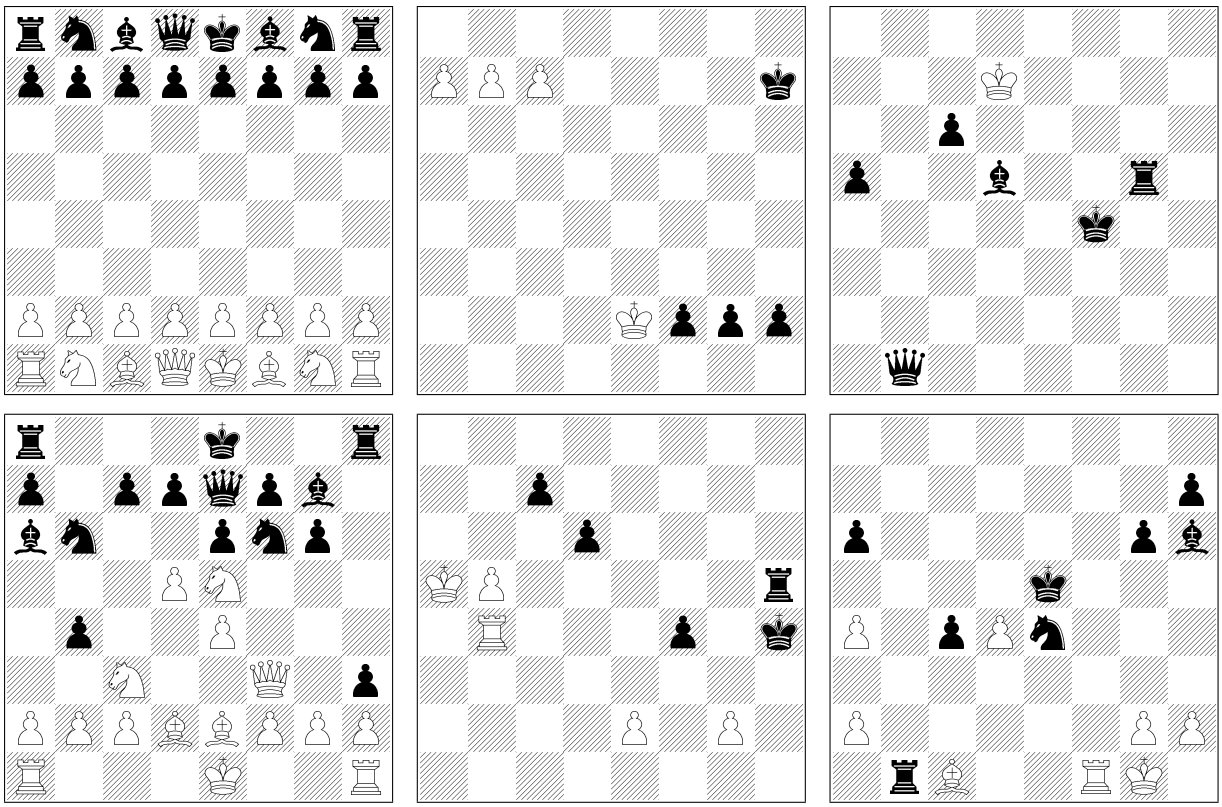


FIG. 10 – Positions analysées par le projet SMIRF

En particulier, les résultats sont décomposés en : captures, prise en-passant, échec, mat, promotion et roque. Les positions n'ont pas été choisies au hasard, mais pour couvrir tout l'éventail des possibilités. Ce n'est que grâce à ces chiffres comparatifs que nous avons pu localiser quelques bug résiduels, liés généralement à la prise en passant ou au roque. Cette méthode systématique est intéressante, parce qu'il est difficile de tout envisager «à la main».

4.2 Performances

Nous avons effectué quelques test sur diverses positions, afin de mesurer les améliorations qu'apportent chaque composante de l'algorithme. Pour ne citer qu'une position, nous reprenons

la FIG. 1. Plus précisément, nous effectuons dans cette position une recherche à profondeur 5 – qui donne bien **14. ♗×g6 !** – en activant progressivement les diverses composantes de l'algorithme :

Algorithme	**Temps (sec.)**	**Gain**
α, β standard	14.86	1
+ PVS	4.95	3.0
+ Hash Table	2.90	5.1
+ Null Move	2.01	7.4
+ Futility Pruning	1.73	8.6

Ainsi dans la position considérée, l'algorithme va 8.6 fois plus vite avec PVS + Hash Table + Null Move + Futility Pruning, qu'un α, β standard. De plus, ce chiffre ne tient pas compte de l'amélioration essentielle apportée par le tri des coups utilisant le SEE : celui-ci est toujours appliqué. Même si nous ne l'avons pas chiffrée précisément, l'amélioration due au SEE nous a semblé flagrante dès que nous l'avons testée.

Conclusion

Finalement, notre programme répond de façon satisfaisante au problème posé initialement. Il sait en effet résoudre des problèmes tactiques, mais aussi avancer stratégiquement dans une partie. Cependant, BibiChess est loin d'être parfait et beaucoup de choses peuvent être améliorées : le quiescent search, l'évaluation, les algorithmes de pruning. Quoiqu'il en soit, le résultat obtenu est plutôt motivant, aussi nous continuerons à développer le programme par la suite.

BibiChess est aujourd'hui disponible sur Internet – `www.uciengines.de` – et des listes indépendantes lui donnent un niveau d'environ 2 000 points elo.

A Pré requis

A.1 Règles du jeu

Commençons d'abord par rappeler brièvement les règles du jeu. Les déplacements des pièces seront supposés connus. Nous rappelons ici toutes les «autres règles», parfois méconnues :

- **Le roque.** Il y a deux roques possibles : petit roque (côté roi) et grand roque (côté dame). Pour pouvoir roquer, il faut que le roi et la tour concernée n'aient jamais bougés, que le roi ne soit pas en échec et qu'il ne passe pas par un échec pour roquer. Bien sûr il faut aussi que l'espace séparant le roi et la tour soit vide.
- **La prise en passant.** Apres une double poussée de pion, la case intermédiaire est capturable par un pion adverse au coup suivant (et seulement au coup suivant).
- **La promotion.** Dès qu'un pion arrive sur la dernière rangée, il se transforme en pièce (au choix : cavalier, fou, tour, dame).
- **Fin normale du jeu.** Le jeu s'arrête lorsque le joueur qui doit jouer n'a plus de coups légaux : s'il est en échec il perd par *échec et mat*, sinon il est *pat* et la partie est nulle. La partie est aussi déclarée nulle si aucun des deux joueurs n'a de matériel suffisant pour mater (par exemple, roi contre roi + cavalier).
- **Fin de parties perpétuelles.** La *règle des* 3 *coups* stipule que lorsqu'une position donnée s'est retrouvée 3 fois dans la partie, cette partie est nulle. De façon similaire, la *règle des* 50 *coups* déclare la partie nulle après 50 coups *réversibles* consécutifs : les coups irréversibles étant les captures et poussées de pions.

A.2 Notation des coups

Il existe essentiellement deux «bonnes façons» de noter les coups, selon l'utilisation qui en est faite.

Notation algébrique. La façon la plus simple de noter les coups – sans ambiguïté – est de concaténer la case de départ et la case d'arrivée. Il faut juste signaler, en cas de promotion, la pièce choisie. Ainsi, un début de partie classique ressemble à ça : `e2e4 c7c5`, `g1f3 d7d6` (...) `e7e8q`. Bref, ce genre de notation est facile à générer informatiquement, mais difficile à lire pour l'utilisateur. Ainsi, notre programme utilise cette notation dans le seul but de communiquer en bas niveau avec une interface (voir B), mais affiche ses coups à l'utilisateur en notation SAN.

Notation SAN. C'est cette notation qui est utilisée dans le présent document. Il s'agit cette fois de concaténer :

1. la pièce déplacée $\in \{♘, ♗, ♖, ♕, ♔\}$
2. un `'x'` pour une capture
3. la case d'arrivée
4. une indication en cas de promotion : `'=Z'` pour une promotion de pièce $Z \in \{♘, ♗, ♖, ♕\}$
5. un indicateur `'x'` en cas d'échec, et `'#'` pour un échec et mat.

Ainsi, le même début de partie ressemble à ça : **1. e4 c5; 2. ♘f3 d6** (...) **40. e8=Q+**, ce qui est bien plus parlant pour l'utilisateur. Dans certains cas, cette notation est ambiguë. Par exemple, dans le cas de la Fig. 1, **14. ♘×e6** peut désigner **14. ♘f2×e6** ou **14. ♘g5×e6**. On utilise alors les notations **14. ♘f×e6** ou **14. ♘g×e6**. De la même façon, si les deux cavaliers ne sont pas distinguables par leur colonne, mais par leur rangée, on utilise le numéro de rangée.

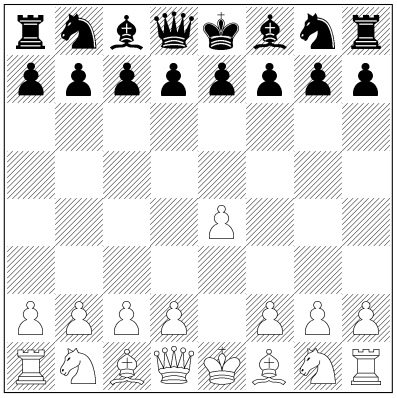
Enfin, les roques sont notés `O-O` (petit roque) et `O-O-O` (grand roque). Bien sûr, d'un point de vue informatique, un coup SAN est une chaîne de caractère, donc les jolies icônes ♘♗♖♕♔ sont remplacées par les vilaines abréviations `NBRQK` (kNight ♘, Bishop ♗, Rook ♖, Queen ♕, King ♔).

A.3 Notation FEN des positions

La position de l'échiquier est caractérisée par : l'agencement des pièces, le tour de jeu, les 4 permissions de roquer, la case éventuelle de prise en-passant[13]. Par exemple, la position de la Fig. 1 se note

```
r1bn1rk1/pp2b2p/1q2pnp1/2pp2N1/3P1N2/2PB4/PPQ2PPP/R1B2RK1 w - -
```

Il s'agit en effet, de lire l'échiquier de haut en bas et de gauche à droite, de noter les pièces (`NBRQK` pour blanc et `nbrqk` pour noir) et de compter les espaces vides – d'où les chiffres qui apparaissent. Ensuite, le `'w'` signifie que c'est à blanc de jouer. Enfin, puisqu'il n'y aucune permission de roquer et pas de case de prise en passant (le dernier coup étant **13. ... g6**) `'-'` est répété deux fois. Un dernier exemple (pour le roque et la prise en passant) : après **1. e4** dans la position initiale :



```
rnbqkbnr/
pppppppp/
8/
8/
4P3/
8/
PPPP1PPP/
RNBQKBNR
b KQkq e3
```

Fig. 11 – Exemple de notation FEN.

`e3` est bien sûr la case de prise en passant et les quatre permissions de roquer sont au vert (`'KQ'` pour le roque blanc côté roi et dame, idem pour noir en minuscules). En pratique cette notation nous est très utile, ne serait-ce que pour faire des copier-coller entre l'interface [4], le code en `C++` et le code LaTeX de ce rapport (notations de coups et diagrammes).

B Interface UCI

B.1 Manuel de l'utilisateur

Pour tester le programme avec une interface graphique, il faut :

1. Télécharger l'interface Arena [4] sur `www.playwitharena.com`
   Dans Arena Downloads (sur la gauche) choisir la version 1.1 setup2, et l'installer en français. Au message demandant si l'on souhaite 'Set Nalimov Path...' répondre 'Nein'.
2. Dans Arena, faire `Module => Nouveau Module => UCI`.
   Sélectionner le fichier exécutable du programme, le module portera le nom de ce fichier.
3. Dans Arena, faire `Module => Gérer`.
   Dans l'onglet Choix, placer le module en premier module actif et faire `Ok`.

Pour utiliser le livre d'ouvertures installé par défaut avec la version 1.1, il faut :

1. Dans Arena, faire `Biblio. => Gérer`.
   Dans l'onglet Données, faire `Nouveau`.
   Sélectionner little-mainbook.abk

[13]...ainsi que le numéro du dernier coup et le compteur des 50 coups, mais passons.

2. Dans Arena, faire `Modules => Gérer`.
   Dans l'onglet Détails,
   sélectionner le module qui porte le nom du fichier exécutable,
   Dans le sous-onglet (sur la droite) Bibliothèques, cocher 'Utilise bibliothèque, de Arena avec ce module',
   faire `Appliquer`, puis `Ok`.

Pour changer les options paramétrables du moteur, il suffit de faire `Ctrl+1` dans Arena. Toute modification de ces options est mémorisée pour la prochaine utilisation, mais elles peuvent être réinitialisées à leurs valeurs par défaut.

B.2 Le protocole UCI

Le protocole UCI est simple à manier, puisqu'il fonctionne par des entrées/sorties en mode texte, c'est-à-dire à base de `cin >>` et de `cout <<`. Pour une définition exhaustive du protocole UCI, nous renvoyons à [3]. Juste pour vous donner une vague idée du protocole, voici les informations échangées par le programme et l'interface, correspondant à l'exemple ci-dessus (en mode console) :

```
// Chess Engine <=> Interface
<= uci                        // Permet de débuter la conversation
=> id name BibiChess 0.5      // Le moteur se présente: donne son nom,
=> id author Lucas et Romain // ainsi que celui de ses auteurs.
=> option name Specimen type check default true // Puis déclare ses options paramétrables
=> option name Specimen2 type spin default 100 min 0 max 500
=> uciok                      // Mets fin aux présentations

<= setoption name Specimen2 value 200 // Arena restitue les options changées.
<= isready // "Es tu prêt?"
=> readyok // La réponse à "isready" sera "readyok" dès que possible

<= ucinewgame // L'interface prévient le moteur qu'il doit
<= isready    // se préparer à une nouvelle partie,
=> readyok    // puis attend que le moteur soit prêt.

<= position startpos moves e2e4 // position de départ, puis les coups joués depuis
<= go wtime 120000 btime 120000 // Donne le signal pour débuter la recherche, ainsi que
                    // les informations relatives au contrôle du temps (ici 2 minutes)

=> info depth 2 score cp -4 pv c5 // Le moteur transmet des informations
=> info depth 3 score cp 0 pv c5  // intermédiaires afin que l'interface
=> info depth 4 score cp -8 pv c5 // puisse les afficher, le score est
=> info depth 5 score cp -4 pv c5 // indiqué en centième de pion
=> bestmove c7c5 // Le moteur a fini sa recherche, il indique le meilleur coup trouvé

<= position startpos moves e2e4 c7c5 g1f3 // énumération des coups depuis le début
<= go wtime 114398 btime 114753           // temps disponible
... // info intermédiaire, comme précédemment
=> bestmove d7d6
```

Références

[1] Clés de Zobrist : `www.seanet.com/~brucemo/chess.htm` – page perso d'un certain Bruce Moreland, que je remercie.

[2] Projet SMIRF : calcul détaillés des premières valeurs de la fonction Perft, dans différentes positions, `www.chessbox.de`

[3] Définition du protocole UCI : `www.shredderchess.com`

[4] Arena, une interface UCI gratuite : `www.playwitharena.com`

[5] Chess Programming Theory : `www.frayn.net/beowulf/theory.html`

[6] GNU Chess : `http://ftp.gnu.org/pub/gnu/chess/`

[7] Fruit 2.1 : `http://wbec-ridderkerk.nl/html/download.htm`