Il n'est pas impossible de résoudre le jeu d'échecs

Julien Lemoine [1] et Simon Viennot [2]

1. Introduction

Qu'entend-on par « résoudre le jeu d'échecs » ? Tout joueur d'échec s'est un jour confronté à un problème du type « les blancs jouent et matent en trois coups » (voir Figure 1). Pour un tel problème, quelles que soient les réponses du joueur noir, le joueur blanc arrive à mater en trois coups, ou moins. Le problème est correctement analysé dès lors que toutes les réponses du joueur noir sont prises en compte.

Au fur et à mesure que le joueur humain progresse, il peut étudier des positions de plus en plus complexes, mais la longueur des analyses requises fait que les problèmes dépassent rarement 5 ou 6 coups. L'ordinateur vient ensuite épauler le joueur humain dans son analyse, et pour des positions de fin de partie, on peut alors déterminer si, en supposant un jeu parfait, l'un des deux joueurs est en position de gagner, ou si la partie se terminera par un nul.

La même question peut se poser avec la position de départ. On sait qu'il est théoriquement possible de démontrer l'une de ces trois assertions :

— Blanc peut mater à partir de la position de départ.
— Noir peut mater à partir de la position de départ.
— Blanc et Noir peuvent obtenir le nul à partir de la position de départ.

1. Professeur en collège (LOL) à Mauriac (dans l'académie de Clermont-Ferrand), julien.lemoine@gmail.com.

2. Professeur assistant à JAIST (Japan Advanced Institute of Science and Technology), sviennot@jaist.ac.jp.

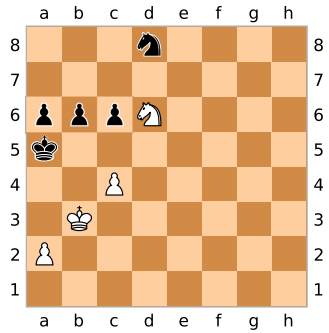

FIGURE 1. Les blancs jouent et matent en trois coups.

On aurait alors *résolu* le jeu d'échecs. Mais la quantité de calculs requise est telle qu'à l'heure actuelle, personne n'y est arrivé – pire, la plupart des spécialistes doute que ce soit un jour possible. Ainsi, dans l'article de 2002 *Games solved : now and in the future* [14], le jeu d'échecs est décrit comme « insoluble par quelque méthode que ce soit ». Interrogé sur le sujet suite à sa résolution du jeu de dames anglaises en 2007, Jonathan Schaeffer, plus prudent, déclare : « une avancée majeure, comme les ordinateurs quantiques, serait nécessaire pour ne serait-ce qu'essayer de résoudre les échecs » [11].

Prenant acte de ces funestes prédictions, nous allons, dans cet article, présenter un rapide tour d'horizon des principales techniques d'étude des jeux combinatoires, dont les échecs font partie. Ainsi, nous observerons qu'en effet, une analyse rapide du problème conduit à être sceptique concernant la possibilité de résoudre le jeu d'échecs ; mais ensuite qu'à l'inverse, une analyse plus poussée conduit à penser que les échecs seront résolus un jour, et même dans un avenir pas si lointain.

2. Les jeux combinatoires

Le jeu d'échecs appartient à un ensemble de jeux partageant des propriétés communes, appelés *jeux combinatoires*, dont voici les deux principales propriétés :

— Il n'y a pas d'intervention du hasard. On ne lance pas de dé comme au 421, on ne tire pas de cartes comme au Poker.

— Ce sont des jeux à *information complète*. Pour choisir son coup, chaque joueur dispose de toutes les informations concernant le jeu pour prendre sa décision. Ceci exclut par exemple le jeu de la bataille navale, où le plateau de l'adversaire est caché.

Ces deux propriétés permettent d'éliminer tout facteur chance, si bien qu'en présence de deux joueurs jouant parfaitement, le résultat du jeu est connu d'avance. Dans un jeu combinatoire, le perdant ne peut pas mettre en cause son manque de

FIGURE 2. Une position de Tic-tac-toe.

chance, il ne peut s'en prendre qu'à lui-même, sauf si, son opposant ayant joué parfaitement de bout en bout, il n'a jamais été en position de remporter la partie.

Parmi les jeux combinatoires les plus connus, outre les échecs, on peut citer les dames, le jeu de Go, ou le Connect-Four (« Puissance 4 »). Les techniques présentées dans cet article sont issues de l'étude informatique des jeux de ce type. Elles seront illustrées par le célèbre jeu de *Tic-tac-toe* [3].

De par leur nature, les jeux combinatoires se prêtent particulièrement bien à la programmation. L'efficacité des programmes est corrélée à la complexité du jeu : pour un jeu simple, comme le Tic-tac-toe, on n'a aucune difficulté à déterminer informatiquement une stratégie optimale. Pour un jeu plus complexe comme le Connect-Four, il a fallu attendre 1988 pour qu'un programme calcule que le premier joueur dispose d'une stratégie gagnante [1].

En 2007, ce fut au tour des dames anglaises [4] d'être résolues : si les deux joueurs jouent de manière optimale, la partie se termine par un nul [9]. Il s'agit du jeu combinatoire le plus complexe à avoir été résolu à l'heure actuelle.

Pour le jeu d'échecs, encore plus complexe, les ordinateurs battent désormais les meilleurs joueurs humains, mais sans disposer de la stratégie optimale. Enfin, pour le jeu de Go, considéré comme le jeu combinatoire classique le plus complexe, les ordinateurs sont encore d'un niveau inférieur à celui des meilleurs joueurs humains, mais ont atteint récemment le niveau de très bons joueurs [8].

3. Les échecs : état des lieux

Intéressons-nous plus particulièrement à la programmation du jeu d'échecs.

Les premiers développements théoriques sur la création d'un programme capable de jouer aux échecs datent de l'immédiat après-guerre. On peut par exemple citer la description détaillée, par Alan Turing, d'un programme d'un niveau de débutant, capable de jouer des coups pas trop ridicules, et dont il avait simulé le calcul à la main pour jouer contre une collègue [13].

Au début des années 1960 apparaît le premier programme à disposer d'un jeu crédible, *Kotok-McCarthy*, développé au MIT. En 1966 a lieu le premier match entre programmes, et *Kaissa*, développé à Moscou, bat *Kotok-McCarthy* lors d'un match de 9 mois dont les coups étaient échangés par télégraphe.

3. Jeu souvent désigné improprement comme *Morpion* : le Morpion consiste à aligner 5 symboles sur une grille illimitée.

4. *Checkers* ou *english draughts*, il s'agit d'une version des dames se déroulant sur un plateau 8×8.

Les progrès continuent, et les programmes commencent à se confronter à des joueurs humains. *Chess* de l'université Northwestern atteint un niveau Elo de 2000 en 1977, *Cray Blitz* atteint 2200 en 1981, et *HiTech* atteint 2500 en 1985. *HiTech* est également le premier programme à battre un grand maître en 1988.

Dans les années 1990, les programmes commencent à surpasser les meilleurs joueurs mondiaux, d'abord en Blitz, puis dans des conditions de temps normales, avec la victoire très médiatisée de *Deep Blue* contre Garry Kasparov en 1997. Si, pendant encore quelques années, les meilleurs joueurs humains ont pu rivaliser avec les ordinateurs, depuis la défaite de Kramnik contre Deep Fritz en 2006, il est acquis que les ordinateurs ont dépassé les humains.

Tous ces programmes ont un point commun : ils ont été développés pour fonctionner en temps limité, et recherchent un bon coup, plutôt que le meilleur coup théorique dont l'obtention est en général trop coûteuse en temps. Dans la suite, au contraire, nous discutons de la possibilité de développer un programme capable de jouer, à chaque coup d'une partie, le meilleur coup.

4. Arbre de jeu

Dans un article de 1950 resté célèbre [10], Claude Shannon donne une estimation du nombre de parties différentes possibles au jeu d'échecs : il obtient le nombre affolant de 10^{120}, et encore, son estimation est très optimiste, le nombre réel étant probablement bien plus important. Face à un nombre de cette ampleur, très supérieur au nombre estimé d'atomes dans l'univers (de l'ordre de 10^{80}), il apparaît bien présomptueux d'espérer résoudre le jeu d'échecs. Et de fait, le nombre de Shannon est régulièrement utilisé comme argument massue pour clouer le bec à tout impudent qui oserait évoquer cette possibilité ; or, ce nombre ne représente pas une bonne mesure de la difficulté du jeu d'échecs.

Pour comprendre à quoi correspond exactement ce nombre, il faut introduire la notion d'*arbre de jeu*, classiquement utilisée dans l'étude des jeux combinatoires. Un arbre de jeu est un *graphe*, c'est-à-dire un objet mathématique composé de *sommets* reliés par des *arêtes*. Dans le cas de l'arbre de jeu, les sommets sont appelés *nœuds*, et représentent les positions. Deux nœuds sont reliés par une arête si on obtient une position à partir de l'autre en un coup (on parle de *position fille*, ou de *nœud fils*). Les arbres sont représentés à l'envers de leur homologue végétal : la *racine*, la position de départ, est placée en haut. Puis, sur la ligne juste en-dessous, on place les positions filles de la racine, et les petites-filles encore en-dessous.

Prenons l'exemple du Tic-tac-toe. La racine est la position de départ, la grille vide. Le premier joueur peut alors jouer 9 coups différents, puis il reste 8 coups possibles au deuxième joueur, puis à nouveau 7 coups pour le premier joueur... La figure 3 présente le début de cet arbre, s'arrêtant après le premier coup pour des raisons évidentes de place.

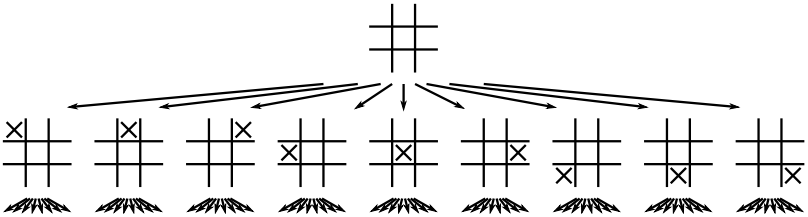

FIGURE 3. Arbre de jeu du Tic-tac-toe.

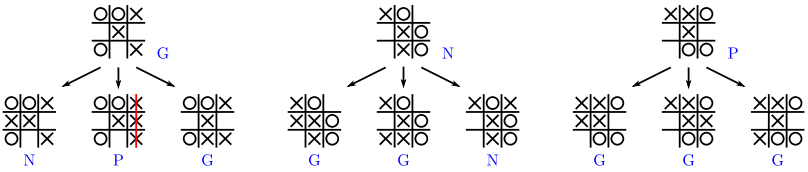

FIGURE 4. Un nœud gagnant, un nul, et un perdant.

Car si on développait cet arbre en entier, il y aurait *a priori* $9 \times 8 \times 7 \times ... \times 1 = 9! = 362\,880$ *feuilles* : les feuilles de l'arbre représentent les positions terminales, lorsque plus aucun coup n'est possible. En réalité, il y en aurait moins, car le jeu s'arrête dès lors qu'un alignement de trois symboles identiques est réalisé. En tenant compte de cette correction, on dénombre 255 168 feuilles [3].

Le *résultat* d'une position est le résultat, pour le joueur dont c'est le tour, d'une partie jouée à partir de cette position, en supposant qu'aucun joueur ne commette d'erreur. Dans le Tic-tac-toe comme dans les échecs, il peut être *gagnant*, *nul* ou *perdant*. Il est facile de déterminer le résultat d'une feuille : c'est *nul* s'il n'y a aucun alignement après 9 coups, et *perdant* si un alignement vient de se produire (car s'il vient d'être tracé par un joueur, c'est à l'autre joueur de jouer... et il a perdu). Les trois règles suivantes permettent ensuite de calculer le résultat d'un nœud à partir des résultats de ses fils :

— Si un nœud a au moins un fils perdant, alors il est gagnant.
— Si un nœud n'a aucun fils perdant, mais au moins un fils nul, alors il est nul.
— Si un nœud n'a que des fils gagnants, alors il est perdant.

La figure 4 illustre chacune des trois possibilités :

— Le premier nœud étudié est gagnant, car il a un fils perdant, celui du milieu, obtenu en faisant une ligne (l'adversaire a alors perdu). Évidemment, le joueur à la croix va préférer ce coup aux deux autres, car c'est le seul qui assure sa victoire.

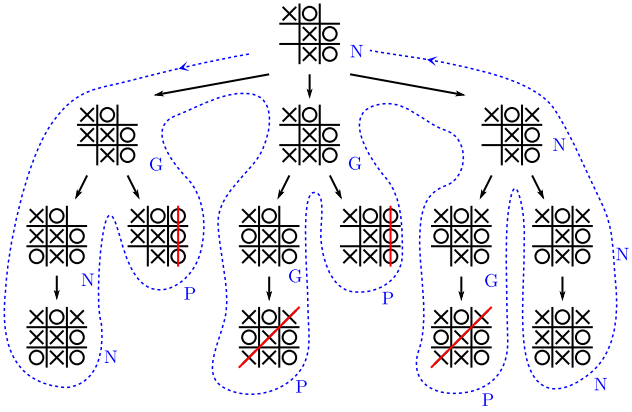

FIGURE 5. Déroulement de l'algorithme Minimax sur l'arbre de jeu d'une position de Tic-tac-toe.

— Le deuxième nœud étudié a deux fils gagnants (pour l'adversaire), et un nul. Le joueur à la croix joue donc le seul coup qui lui permet le nul, faute de mieux. Le nœud est donc nul.
— Le troisième nœud n'a que des fils gagnants : quel que soit le coup joué, le joueur à la croix va permettre à l'autre de faire une ligne (on reconnaît une situation de fourchette), il a donc perdu.

L'algorithme standard chargé d'utiliser ces règles pour calculer les résultats des nœuds autres que les feuilles, en remontant l'arbre, est appelé Minimax [5]. Concrètement, lorsqu'il est appelé sur un nœud, l'algorithme Minimax calcule le résultat de chacun de ses fils, puis utilise ces résultats pour calculer le résultat du nœud. La figure 5 permet de visualiser le déroulement du calcul sur une position de Tic-tac-toe.

L'algorithme explore les nœuds de l'arbre de jeu dans l'ordre indiqué par la ligne en pointillés, et le résultat de chaque nœud est indiqué à côté de la ligne dès lors que l'algorithme l'a calculé. Une fois l'arbre exploré en entier, on obtient le résultat de la racine. Ainsi, la connaissance de l'arbre de jeu dans sa totalité permet de déduire le résultat que peuvent escompter chacun des joueurs en début de partie.

Remarquons un avantage de cet algorithme : il n'utilise que très peu de mémoire. Par exemple, sur la figure 5, lorsqu'il est en train de calculer la branche du milieu, il n'a pas besoin de se rappeler tout le calcul de la branche de gauche ; il lui suffit de se rappeler que le premier fils est gagnant. En général, à un moment donné du

5. Les théoriciens remarqueront à la lecture des trois règles, d'une part qu'il s'agit d'une version épurée de l'algorithme Minimax, adaptée à la résolution de jeux plutôt qu'à la recherche d'un bon coup en temps limité ; d'autre part, qu'il s'agit plutôt de l'algorithme Negamax, version de l'algorithme adaptée aux jeux impartiaux.

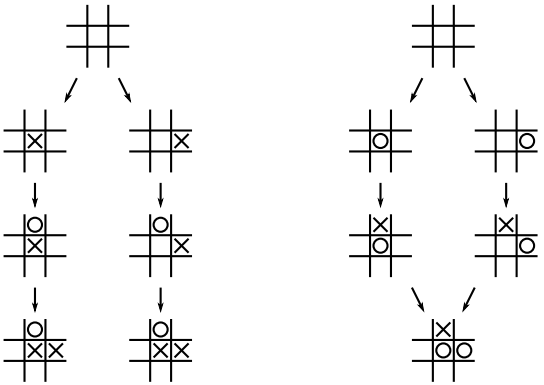

FIGURE 6. Une transposition, puis l'arbre de jeu représenté sous forme compacte..

calcul, il n'a besoin de se rappeler que d'une fratrie par étage. Même pour un jeu très complexe comme le jeu d'échecs, les parties ne peuvent dépasser quelques centaines de coups, et le stockage de quelques centaines ou même milliers de positions ne pose aucun problème. Le problème, car problème il y a, réside dans le temps de calcul.

En théorie des jeux combinatoires, la *game-tree complexity* est une mesure de la complexité d'un jeu : elle correspond au nombre de feuilles de l'arbre de jeu. Chaque feuille correspondant à une partie différente, il s'agit de manière équivalente du nombre de parties différentes de ce jeu. La game-tree complexity du jeu d'échecs est d'au moins 10^{120}, ce qui justifie l'impossibilité de résoudre ce jeu par le simple algorithme Minimax : même en disposant d'un ordinateur évaluant 3 000 milliards de feuilles à la seconde, une année ne permettrait d'évaluer qu'à peine 10^{20} feuilles... on est très loin du compte !

Mais on ne peut pas pour autant en déduire l'impossibilité de résoudre le jeu d'échecs : la game-tree complexity du Tic-tac-toe a beau être de 255 168, nous allons pouvoir le résoudre par des moyens humains avant la fin de cet article, ce qui apparaît *a priori* comme une performance, vu la piètre puissance de calcul de notre cerveau. En effet, la game-tree complexity ne tient pas compte d'un certain nombre de techniques de simplification du calcul que nous allons maintenant décrire.

5. Transpositions

Il arrive que deux parties aux déroulements différents mènent à la même position. C'est le cas par exemple à gauche de la figure 6, où l'on a représenté une portion de l'arbre de jeu du Tic-tac-toe. À partir de la position initiale, deux parties différentes mènent en trois coups à la même position T : on dit qu'il s'agit d'une *transposition*.

Dans le cas du jeu d'échecs, on rencontre de nombreuses transpositions. Dans le cas le plus simple, elles se produisent après seulement deux coups joués par l'un des joueurs. Par exemple, les débuts de partie correspondant aux séquences de coups :

```
1.Cf3 e5 2.Cc3
```

et

```
1.Cc3 e5 2.Cf3
```

conduisent à la même position ; mais il est possible d'obtenir des transpositions après de plus nombreux coups [6].

Du fait des transpositions, il est possible de représenter les arbres de jeu de manière plus compacte, en ne conservant qu'un seul nœud pour chaque transposition. Pour l'exemple de la figure 6, on obtient la portion d'arbre [7] de jeu représentée à droite sur la figure.

Il est possible d'exploiter ce phénomène pour accélérer le calcul du résultat de la position initiale : une fois le résultat de la position T calculé, on le stocke. Lorsque plus tard dans le calcul, on rencontre à nouveau la position T, on peut utiliser le résultat stocké plutôt que de le calculer à nouveau. Ce faisant, on gagne du temps, mais on perd de la mémoire : là où l'algorithme de la section 4 ne nécessitait à chaque instant que le stockage de quelques positions, il est désormais nécessaire de stocker le résultat de toutes les positions rencontrées, dans ce que l'on nomme une *table de transpositions*.

On vient d'observer une des multiples manifestations d'une règle bien connue des programmeurs : le temps de calcul peut se transformer en mémoire, et vice-versa. Toute la difficulté consiste à effectuer ces transformations de façon à ce que le calcul effectué devienne possible par la machine dont on dispose.

Refermons cette parenthèse, et déterminons la taille maximale prise par la table de transpositions. À la fin du calcul, elle contient toutes les positions possibles du jeu étudié. Ce nombre de positions possibles se nomme *state-space complexity*.

La state-space complexity est une autre mesure de la complexité d'un jeu. Elle est généralement nettement moins grande que la game-tree complexity, et plus pertinente, en tenant compte de la possibilité d'utiliser une table de transpositions pour résoudre le jeu. Pour le Tic-tac-toe, la réduction est notable, car on passe de 255 168 parties à 5 478 positions – encore trop pour résoudre le jeu manuellement. Et de

6. Pour vérifier que deux positions sont bien équivalentes, il faut vérifier qu'elles le sont du point de vue des règles du roque, de la prise en passant, ou pire, de la règle des 50 coups sans prise, et de celle de la répétition trois fois d'une même position. Cette dernière règle en particulier exclut la quasi-totalité des transpositions, dont celle de l'exemple. Il est cependant raisonnable de ne pas trop se soucier de cette règle, comme expliqué dans [4] au sujet du *graph-history interaction problem*.

7. Il ne s'agit plus d'un *arbre* au sens de la théorie des graphes, mais nous conservons tout de même le terme d'*arbre de jeu*.

même, dans le cas des échecs, la réduction n'est pas suffisante pour laisser espérer résoudre le jeu un jour.

En effet, dans le même article de 1950 [10], Shannon estime la state-space complexity du jeu d'échecs à 10^{43}. Ce n'est qu'une estimation, basée sur un calcul simple, or le nombre exact est très complexe à déterminer. Car Shannon à la fois sous-estime (il oublie captures de pièces, promotions de pions) et sur-estime ce nombre (en conservant des positions illégales, comme deux fous blancs pour un même joueur sans promotion). Des évaluations récentes plus précises tendent à montrer qu'il avait tout de même visé assez juste, probablement en surestimant le résultat plutôt que le contraire (il est certain que le nombre correct est en-dessous de 10^{46} [12]).

Quoi qu'il en soit, 10^{43} est une valeur encore bien trop grande. L'espace de stockage de tous les ordinateurs existants à l'heure actuelle est de l'ordre de 10^{20} octets, aussi nous ne sommes pas près de voir apparaître un disque dur assez grand pour stocker 10^{43} positions. Encore une fois, nous semblons être dans une impasse.

Mais même si la state-space complexity est une notion plus pertinente que la game-tree complexity pour évaluer la complexité d'un jeu (ce dont les spécialistes sont conscients, c'est souvent 10^{43} qui est cité plutôt que 10^{120} dans les articles abordant la résolution du jeu d'échecs), elle a aussi ses faiblesses.

Car il est facile d'imaginer un jeu qui mette cette mesure en échec : sur un échiquier vide, Blanc et Noir placent, chacun leur tour, un de leurs 8 pions sur un emplacement libre. C'est le seul coup possible : il n'y a pas de prise, pas de déplacement. Blanc commence, et le premier à avoir placé tous ses pions a gagné. En dépit de sa state-space complexity monstrueuse [8], il est aisé de voir que Blanc gagnera quoi qu'il arrive ce jeu idiot, plaçant son huitième pion juste après que Noir aura placé son septième...

L'algorithme Minimax ou la table de transpositions ne sont d'aucun secours ici. La résolution que l'on effectue naturellement dans ce cas est une résolution *par méthode*.

6. Résolution par méthode

La résolution par méthode consiste à trouver des simplifications diminuant le nombre de nœuds de l'arbre de jeu.

Dans le cas du jeu idiot, on sait que la partie se terminera en 15 coups quoi qu'il arrive. On peut fusionner tous les nœuds qui sont à la même distance de la racine, et l'arbre de jeu est réduit à une unique branche verticale de 16 nœuds. Cet exemple est exagérément simple ; en général, les jeux combinatoires ne se résolvent pas si facilement. Si certains se résolvent complètement par méthode (comme le jeu de

8. C'est un exercice plaisant de dénombrement que de montrer qu'elle vaut 1 216 708 506 251 460 977.

FIGURE 7. Position ne pouvant plus donner de ligne.

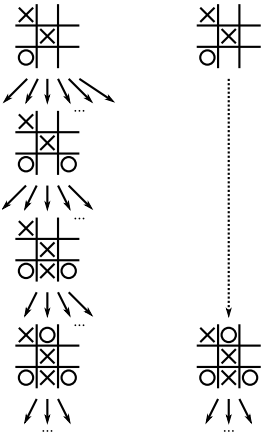

FIGURE 8. Coups forcés au Tic-tac-toe.

Nim), la plupart sont trop complexes pour cela, et les méthodes ne permettent que de simplifier l'arbre de jeu avant de l'explorer.

Ce sont souvent des considérations géométriques, comme la symétrie, qui permettent de fusionner des nœuds de l'arbre de jeu. Dans le cas du Tic-tac-toe, si l'on tient compte de la symétrie, on peut fusionner jusqu'à 8 nœuds à la fois, et la state-space complexity passe de 5 478 à 765 positions. On peut aller plus loin [9], et nous utiliserons deux méthodes supplémentaires dans la suite de l'article :

— Une position qui ne peut plus aboutir à la création d'une ligne, quels que soient les coups joués, peut être considérée directement comme nulle, comme sur la figure 7.

— Si l'adversaire menace de faire une ligne au prochain coup, il faut parer la menace, avec un *coup forcé*. Parfois, les menaces s'enchaînent, et ce sont plusieurs coups forcés qui sont joués à la suite. On peut alors supprimer les nœuds intermédiaires, comme illustré sur la figure 8.

9. En fait, le Tic-tac-toe pourrait se résoudre uniquement par méthode [6]. Nous ne le ferons pas dans le cadre de cet article car le jeu n'illustrerait plus les concepts développés dans les sections 7 et 8.

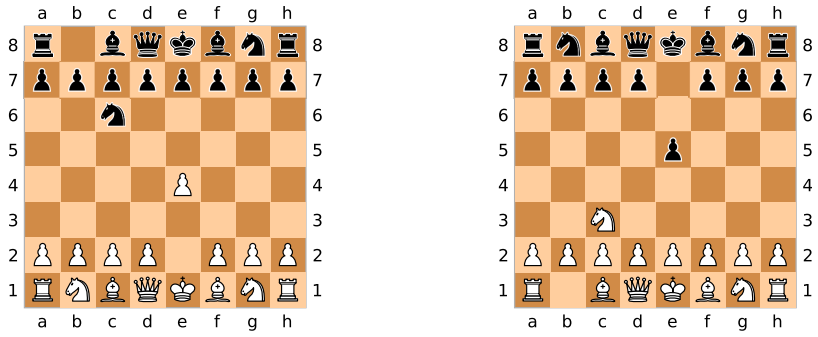

FIGURE 9. Une transposition pas si évidente.

La première méthode permet de déterminer directement le résultat de certains nœuds, ce qui aboutit à supprimer tous les nœuds situés en-dessous. La deuxième permet de supprimer des nœuds internes à l'arbre, car leur étude est inutile.

Des méthodes similaires peuvent être utilisées aux échecs. En début de partie, une seule symétrie est disponible : la symétrie d'axe horizontal, avec inversion des couleurs des joueurs. Donnons un exemple. Les deux positions obtenues après les séquences de coups :

```
1.e3 Cc6 2.e4
```

et

```
1.Cc3 e5
```

sont similaires (voir figure 9), si par exemple c'est Blanc qui dispose d'une stratégie gagnante à partir de la première position, alors c'est Noir qui dispose de cette même stratégie à partir de la deuxième position. Il s'agit en fait d'une transposition [10].

Lorsque les possibilités de roque disparaissent, on peut utiliser la symétrie d'axe vertical ; puis lorsque les pions disparaissent, les symétries d'axes diagonaux et celle d'axe horizontal sans changement de couleur se rajoutent. Au maximum, ce sont donc 16 nœuds que l'on peut fusionner grâce à des considérations de symétrie, comme dans la figure 10.

Les deux autres méthodes vues à propos du Tic-tac-toe ont leur pendant aux échecs. Les coups forcés existent, et donnent, entre autres, quelques problèmes d'autant plus faciles à résoudre qu'il n'y a qu'une variante à explorer. Par exemple, sur la figure 11, les blancs jouent et matent en trois coups, et les deux réponses du joueur

10. En utilisant ces transpositions, on perd une caractéristique des arbres de jeu des échecs : l'alternance entre étages, Blanc ayant le trait pour les positions des étages pairs, et Noir ayant le trait pour les positions des étages impairs.

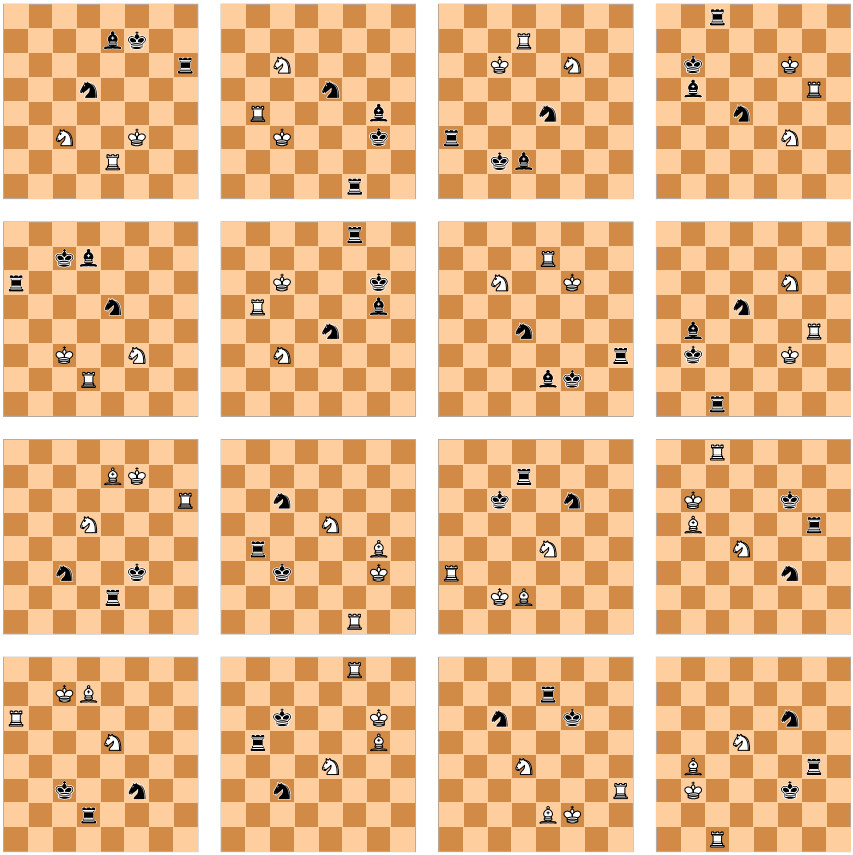

FIGURE 10. 16 positions équivalentes. Les 8 du haut sont avec trait aux blancs, les 8 du bas avec trait aux noirs.

noir sont des coups forcés, car toute autre réponse est sanctionnée d'un mat immédiat [11].

Quant à la résolution directe de nœuds, c'est ce que font déjà les joueurs d'échecs lorsqu'ils apprennent le résultat des finales. Ils savent ainsi que toute position du type *Roi-Tour contre Roi* est gagnante pour le joueur qui a conservé sa tour, hormis les

11. Solution : 1.Dxf6 gxf6 (coup forcé, sinon mat au coup suivant avec la dame en g7) 2.Tg1+ Rh8 (coup forcé, car seul coup légal) 3.Fxf6#

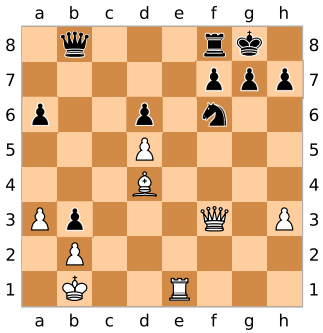

FIGURE 11. Les blancs jouent et matent en trois coups.

cas de pat [12]. Du point de vue du programmeur, il est bien plus efficace de programmer une fonction traduisant le résultat de cette finale, que de stocker le résultat des dizaines de milliers de positions du type *Roi-Tour contre Roi*.

En poussant ce raisonnement, on peut chercher à déterminer le résultat de positions plus complexes que celles des finales classiques. C'est ce que font les *tables de finales*, cependant, il devient difficile de trouver des règles simples décrivant quelles sont les positions gagnantes ou perdantes lorsque la complexité des positions étudiées augmente. On se base alors sur des techniques de compression de données plus classiques, et à l'heure actuelle, les tables de finales de l'ensemble des positions à six pièces ou moins sont calculées [13].

Toutes ces méthodes ne sont pas à négliger dans l'optique de la résolution des jeux d'échecs, mais leur effet ne sera probablement pas décisif. Le travail sur les finales ne concerne que les fins de partie ; or l'écrasante majorité des positions à étudier sont des positions de début ou de milieu de partie, car lorsqu'il y a plus de pièces, il y a plus de façons de les placer. L'effet des coups forcés est anecdotique : en partie réelle, les coups forcés sont assez rares, surtout en début de partie, et de plus ceux que l'on peut imposer à l'adversaire ne se soldent que rarement par un gain.

La méthode la plus efficace est probablement la symétrie. La quasi-totalité des positions légales ne permet pas de roquer, mais dispose aussi d'au moins un pion. La symétrie permet donc de réduire le nombre de positions à étudier d'un facteur 4 environ. Ce n'est pas négligeable, mais par rapport aux 10^{43} positions légales, cela ne

12. Roi A dans un coin (e.g. a1), tour B juste en diagonale (b2) et roi B protégeant la tour B, ou roi A dans un coin (e.g. a1), roi B en c1 ou a3, et tour B sur la ligne passant entre les deux rois, sans mettre le roi A en échec.

13. Pour un espace mémoire d'un peu plus de 1 To. Les tables de l'ensemble des positions à 7 pièces devraient bientôt être calculées, pour un espace mémoire de l'ordre de 100 To. Inutile donc d'espérer résoudre le jeu d'échecs par ce moyen, qui sature trop vite la mémoire.

fait que 1,4 % du chemin [14]. Il va donc falloir compter sur autre chose pour résoudre le jeu d'échecs.

7. Élagage alpha-bêta

La raison principale pour laquelle le jeu d'échecs pourrait être résolu un jour réside dans l'existence de l'*élagage alpha-bêta*. C'est un algorithme simple, qui apparaît naturellement dans l'étude des jeux combinatoires, et dont les premières formulations remontent aux années 1950. Il repose sur l'asymétrie entre les nœuds perdants et gagnants [15] :

— Un nœud est perdant si tous ses fils sont gagnants.
— Un nœud est gagnant si au moins un de ses fils est perdant.

Ainsi, pour prouver qu'un nœud est perdant, il faudra de toute façon prouver que tous ses fils sont gagnants. Mais pour prouver qu'un nœud est gagnant, il suffit de prouver qu'*un* de ses fils est perdant. Une fois cette tâche effectuée, le calcul des autres fils devient inutile.

Reprenons le mat en trois coups de la figure 1. Pour prouver que cette position est gagnante pour le joueur blanc, il suffit de trouver un coup qui place le joueur noir dans une position perdante : le coup c5 fait l'affaire. Mais pour prouver que cette nouvelle position est bien perdante, il faut montrer que tous les coups jouables maintenant par Noir placent Blanc dans une position gagnante. On étudie donc les 5 coups Cb7, Ce6, Cf7, bxc5 et b5.

Étage après étage, on alterne entre positions gagnantes et perdantes, choix d'un coup et étude de tous les coups. La figure 12 résume la solution du problème, et l'arbre obtenu est un *arbre solution* : c'est une partie de l'arbre de jeu suffisante pour calculer le résultat de la position étudiée.

Sur cet arbre, il apparaît que lorsque Blanc a le trait, l'étude d'un coup suffit, et lorsque Noir a le trait, on étudie tous les coups. On peut remarquer que sur les quatre branches de gauche, le deuxième coup de Noir est forcé (représenté en pointillés), car il n'a alors qu'un seul coup légal.

Maintenant, il est possible de décrire le comportement de l'algorithme alpha-bêta. Il fonctionne de manière similaire à l'algorithme Minimax, hormis sur un point : lorsque l'on calcule les fils d'un nœud, dès qu'on a prouvé qu'un de ces fils est perdant, on en déduit immédiatement que le nœud père est gagnant, et on n'effectue pas le calcul des frères du fils perdant. Ce faisant, on tire parti du fait qu'il n'est pas nécessaire d'explorer tout l'arbre de jeu pour obtenir le résultat de la racine.

14. Car $\frac{\log(4)}{\log(10^{43})} \simeq 0,014$: les techniques employées pour réduire le nombre de nœuds étudiés tendent à les diviser par un certain facteur, ce qui justifie le recours à une échelle logarithmique.

15. Dans cette section, nous nous contenterons d'aborder le cas des nœuds gagnants et perdants pour simplifier le propos. Le cas du nul sera traité dans la section 8.

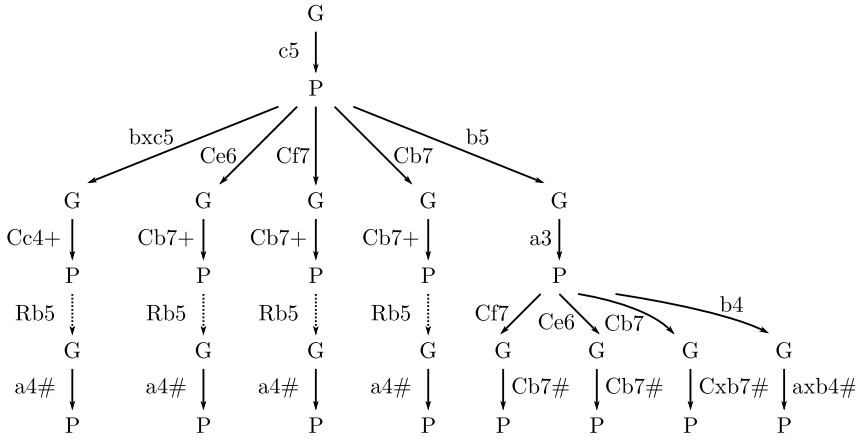

FIGURE 12. Arbre solution du problème de la figure 1.

La figure 13 présente un exemple d'utilisation de l'algorithme alpha-bêta appliqué à une position de Tic-tac-toe. Les flèches barrées représentent tous les nœuds qu'il n'est pas utile d'explorer, dès lors qu'un fils perdant a été trouvé dans leur fratrie. On voit qu'une réduction importante (en mémoire comme en temps de calcul) est opérée par rapport à l'algorithme Minimax : dans le cas de cette position, seules 18 positions sont étudiées, là où l'algorithme Minimax en aurait nécessité 66.

18 n'est cependant pas le nombre optimal. En effet, la racine de cet arbre est gagnante, or l'algorithme a d'abord calculé un premier fils, gagnant, puis un deuxième, perdant : le calcul du premier fils était donc inutile pour démontrer que la racine était gagnante, et ainsi toute la moitié gauche de l'arbre est inutile. En la supprimant, on constate qu'un arbre solution pour cette position contiendrait seulement 10 nœuds.

Ce cas n'est pas une anomalie. En général, l'algorithme alpha-bêta ne tombe pas miraculeusement sur l'arbre solution idéal ; si on explore l'arbre de jeu dans n'importe quel ordre, l'algorithme risque d'effectuer un certain nombre de calculs inutiles, si bien que la façon dont l'algorithme alpha-bêta est programmé influe fortement sur sa performance. Voici donc deux grands principes concernant l'ordre dans lequel on étudie les nœuds, et qui permettent de réduire le nombre de nœuds rencontrés :

— Il faut identifier les bons coups pour éviter les calculs inutiles : si un nœud est gagnant, il faut éviter de calculer ses fils gagnants, ce qui est du temps perdu, et ne calculer qu'un fils perdant. On ordonne donc les nœuds fils de façon à étudier en premier les nœuds qu'on suppose perdants (pour l'adversaire).

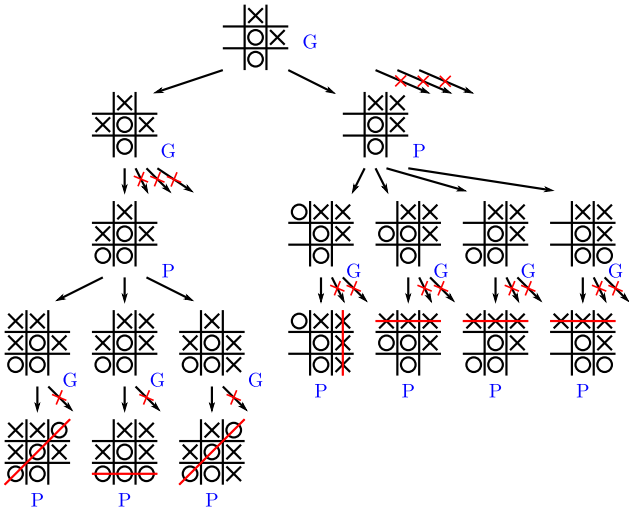

FIGURE 13. Algorithme alpha-bêta appliqué à une position de Tic-tac-toe.

— Il faut préférer un très bon coup à un bon coup : si un nœud gagnant a plusieurs fils perdants, il suffit d'en calculer un. Autant choisir celui dont l'étude sera la plus simple. On ordonne donc les nœuds fils de façon à étudier en premier les nœuds dont l'étude sera la plus rapide.

Si le premier principe est parfaitement appliqué, alors la table de transpositions ne contient qu'un arbre solution. Dans le cas contraire, en plus de cet arbre solution, on trouve certaines branches inutiles comme la branche de gauche de la figure 13.

Pour l'implémentation de ce principe, il serait utile de disposer d'un oracle qui indique quelles sont les positions gagnantes, et quelles autres sont perdantes. Ainsi, il devient utile d'être un bon joueur, ou tout du moins de connaître les stratégies des bons joueurs pour programmer l'alpha-bêta, car il faut que le programme étudie en priorité les bons coups [16].

Le deuxième principe commande de rechercher un arbre solution contenant le moins de nœuds possible. Car une même position possède, en général, de nombreux arbres solution très différents – de la même façon qu'à partir d'une position gagnante, des joueurs de niveaux différents peuvent s'assurer du mat de manière plus

16. Un programmeur qui serait un mauvais joueur d'échecs ne serait pas pour autant impuissant face à ce problème : des algorithmes de parcours de l'arbre de jeu, comme le PN-search [2], utilisent les informations obtenues durant le développement de l'arbre de jeu pour prédire le résultat des positions – et ce, même en l'absence de fonction d'évaluation cherchant à traduire les stratégies humaines.

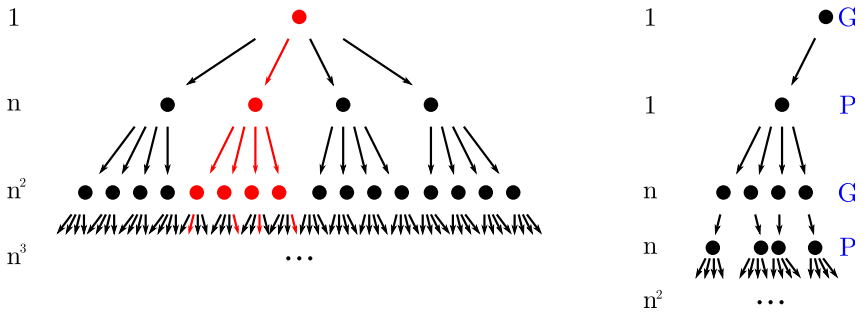

FIGURE 14. Arbre de jeu équilibré, et arbre solution extrait de cet arbre de jeu.

ou moins rapide. Plus la stratégie utilisée pour mater est rapide, plus l'arbre solution correspondant est simple.

Pour l'implémentation de ce principe, il faut tirer parti du fait que l'arbre de jeu des échecs est déséquilibré. Ainsi, quand on a le choix entre plusieurs bons coups, on doit sélectionner celui dont l'étude sera la plus courte. C'est ce que font instinctivement les joueurs qui, disposant d'un avantage matériel sur leur opposant, cherchent à forcer l'échange de pièces pour accélérer la fin de partie.

Évaluons maintenant l'intérêt de l'alpha-bêta dans la résolution du jeu d'échecs. Si on étudie un arbre parfaitement équilibré (chaque nœud interne comportant n fils), sans transposition, et de profondeur p (dont les parties se terminent en p coups), alors le nombre de nœuds de cet arbre est :

$$1+n+n^2+...+n^p = \frac{n^{p+1}-1}{n-1} \sim n^p \quad (1)$$

Si on suppose de plus que p est impair, et que la racine de l'arbre est gagnante, alors le nombre de nœuds d'un arbre solution est :

$$1+1+n+n+n^2+n^2+...+n^{\frac{p-1}{2}} = 2.\frac{n^{\frac{p+1}{2}}-1}{n-1} \sim 2.n^{\frac{p-1}{2}} \simeq \sqrt{n^p} \quad (2)$$

La figure 14 illustre ce calcul avec les premiers étages d'un arbre de jeu, et d'un arbre solution correspondant, pour $n = 4$.

Le nombre de nœuds d'un arbre solution est donc, à peu de choses près, de l'ordre de la racine carrée du nombre de nœuds de l'arbre de jeu. Les autres cas (avec p pair ou une racine perdante) donnent la même estimation. Le gain est énorme ! Pour le jeu d'échecs, comportant de l'ordre de 10^{40} nœuds, cela représenterait seulement 10^{20} nœuds pour un arbre solution – et donc pour la table de transpositions. Ce n'est

déjà plus la même difficulté... L'algorithme alpha-bêta est ainsi la raison principale pour laquelle le recours au nombre de Shannon n'est pas un argument correct pour affirmer que le jeu d'échecs ne peut être résolu.

10^{20} n'est cependant toujours pas une estimation fiable de la taille d'une table de transpositions à même de résoudre le jeu d'échecs. Voici plusieurs phénomènes qui peuvent influer sur la taille de cette table :

— L'algorithme alpha-bêta réalise des calculs inutiles, ce qui conduit à réviser cette estimation à la hausse (le premier principe ne pouvant généralement pas être parfaitement respecté).

— L'effet des transpositions sur cette estimation est ambigu. Par rapport à un arbre sans transpositions, les transpositions réduisent à la fois le nombre nœuds de l'arbre de jeu, et celui de l'arbre solution. Comme on s'intéresse au rapport entre ces deux nombres, les deux effets sont antagonistes. On verra ce qu'il en est au final dans la section 9.

— En revanche, le calcul précédent repose sur un arbre parfaitement équilibré, alors que l'arbre de jeu des échecs est fortement déséquilibré. Concentrer l'étude sur les parties de l'arbre les plus simples (deuxième principe) permet ainsi de réduire fortement le nombre de nœuds étudiés.

On peut prendre conscience de l'intérêt de cette dernière remarque en revenant au problème de la figure 1 : l'arbre solution de la figure 12 ne contient que 28 nœuds, alors que le recours à l'algorithme Minimax nécessiterait de développer tout l'arbre de jeu, arbre qui contient des milliers de positions. Ce sont toutes les positions que l'on peut obtenir à partir de la position de départ de ce problème, avec tous les déplacements des rois, des cavaliers, les promotions de pions, y compris les plus improbables, en fou ou en tour... Ce phénomène est lié au fait que les échecs est un jeu avec une propriété de *mort subite* : le mat. Ainsi, certaines parties se terminent en quelques coups [17], quand d'autres en nécessitent des centaines.

8. Prise en compte du Nul

L'algorithme alpha-bêta expliqué à la section 7 convient à un jeu comportant des positions gagnantes et perdantes, mais ne prend pas en compte les positions nulles. Or, il est facile de l'adapter pour un jeu qui, comme le Tic-tac-toe ou les échecs, contient de telles positions.

Il suffit pour cela de faire deux calculs séparés [18]. Dans un premier calcul, on considère que le premier joueur veut gagner ou faire match nul, et que le deuxième

17. La partie d'échecs la plus courte ne nécessite que deux coups de chaque joueur (et une grande naïveté du joueur blanc) :
1.f3 e6 2.g4 Dh4#.

18. C'est tout du moins la méthode présentée dans cet article. D'autres méthodes existent, par exemple l'algorithme alpha-bêta peut être étendu à trois valeurs.

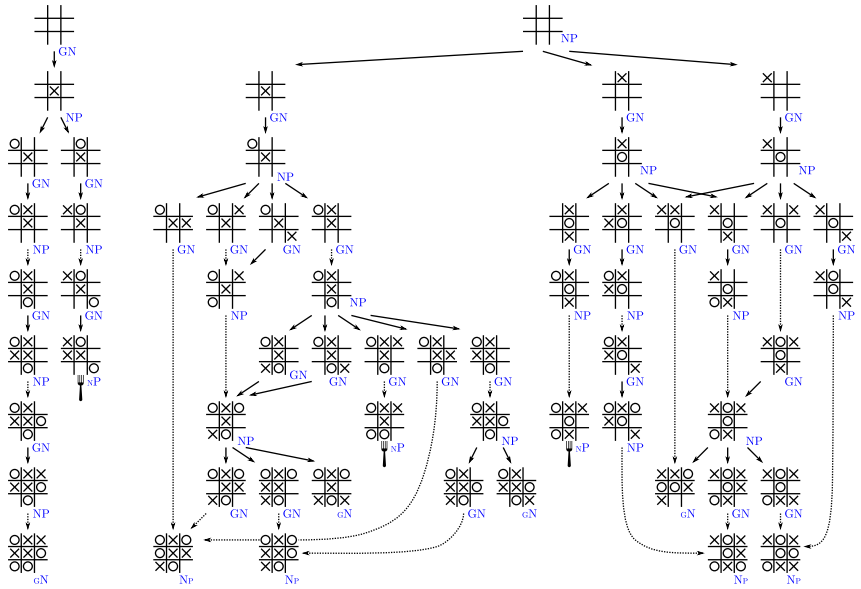

FIGURE 15. Résolution du Tic-tac-toe.

veut seulement gagner. Ainsi, un seul des deux joueurs sera satisfait à la fois, ce qui correspond à un calcul classique du type gagnant/perdant.

Si le résultat du calcul est « perdant », ce qui signifie que le deuxième joueur gagne, alors le travail est terminé. Sinon, on effectue un deuxième calcul, dans lequel on considère que le premier joueur ne veut que gagner, tandis que le deuxième veut gagner ou faire match nul. Une fois le résultat de ce calcul connu, on peut en déduire le résultat de la position étudiée.

En guise d'exemple, nous allons résoudre une bonne fois pour toutes le jeu du Tic-tac-toe, dans la figure 15. Il y a deux calculs séparés : le premier, l'arbre de gauche, montre que la position de départ est gagnante ou nulle (notée « GN »). Le second, l'arbre de droite, montre que la position de départ est nulle ou perdante (notée « NP »). On en déduit que la position de départ est un nul : joué parfaitement, le Tic-tac-toe se finit par un match nul.

On retrouve la dichotomie à l'œuvre dans les calculs du type Gagnant/Perdant : pour montrer qu'un nœud est GN, il suffit d'un fils NP, tandis que pour montrer qu'un nœud est NP, il faut montrer que tous les fils sont GN.

Explicitons les notations. Les pointillés traduisent les coups forcés. Les trois nœuds marqués par une fourchette indiquent des positions où interviennent des... fourchettes, elles sont donc perdantes (et notées « NP » : une position perdante est *a*

fortiori « nulle ou perdante »). Les autres positions terminales sont nulles, et donc notées « GN » ou « NP » suivant leur utilisation.

Les deux arbres ont respectivement 13 et 48 nœuds : la différence de taille traduit qu'il est plus difficile d'obtenir le nul pour le deuxième joueur, ce qui se conçoit car il a un temps de retard. Enfin, remarquons que 61 nœuds suffisent à résoudre le Tic-tac-toe. La comparaison avec la game-tree complexity de 255 168, ou la state-space complexity de 5 478 montre bien que ces deux mesures n'évaluent pas bien la difficulté d'un jeu.

9. Estimation des ressources requises pour la résolution

L'estimation donnée, à la fin de la section 7, de la taille d'une table de transpositions assez grande pour résoudre le jeu d'échecs, est assez grossière ; nous allons maintenant l'affiner, en évaluant l'effet de trois phénomènes différents sur cette taille.

Calculs inutiles

L'algorithme alpha-bêta a pour but de déterminer un arbre solution. Lorsqu'il cherche à démontrer qu'un nœud est gagnant, il choisit soigneusement l'ordre dans lequel les fils de ce nœud sont étudiés, en essayant d'explorer en priorité un fils perdant. Or, quels que soient les raffinements mis en œuvre (par exemple, tri en utilisant la valeur des pièces, les menaces potentielles...), l'ordre choisi se trouve mis en échec sur certaines positions, et l'algorithme se retrouve à explorer des branches inutiles de l'arbre de jeu. En effet, un programme qui trouverait un fils perdant systématiquement sans erreur jouerait parfaitement aux échecs, et le jeu serait alors résolu par méthode.

Aussi, en pratique, pour certains nœuds gagnants, l'algorithme rate son coup et étudie un ou plusieurs fils gagnants avant de trouver un fils perdant. Estimons l'impact de ces calculs inutiles sur la taille de l'arbre solution. On suppose toujours étudier un arbre de jeu équilibré dont chaque nœud interne a n fils, et dont les parties se terminent en p coups. Nous avons vu avec la relation (1) que le nombre de nœuds de cet arbre de jeu est de l'ordre de n^p.

On suppose alors qu'en moyenne, au lieu d'un unique fils, l'algorithme explore α fils avec $\alpha \in [1;n]$. On démontre en annexe que le nombre de nœuds étudiés est de l'ordre de $\alpha^{\frac{p}{2}}\sqrt{n^p}$: on retrouve $\sqrt{n^p}$, l'estimation (2) du nombre de nœuds de l'arbre solution, corrigée d'un facteur $\alpha^{\frac{p}{2}}$ qui traduit l'augmentation de la longueur du calcul. Ce facteur traduit directement la qualité de la programmation, il est possible de le faire diminuer en programmant des algorithmes qui jouent de meilleurs coups.

Transpositions

Étudions maintenant l'effet des transpositions. Nous avions vu que l'effet était difficile à prévoir *a priori*, c'est donc le calcul qui va nous indiquer dans quel sens penche la balance.

On suppose donc que l'arbre de jeu comporte des transpositions, uniformément réparties dans l'arbre ; si bien que lorsqu'on passe d'un étage au suivant, le nombre de nœuds de l'étage n'est pas multiplié par n, mais par tn, où $t \in]0;1]$ (t représente le taux moyen de nœuds restant après transpositions ; $t = 1$ lorsqu'il n'y a pas de transposition).

On démontre cette fois (les calculs sont en annexe) que le nombre de nœuds de l'arbre de jeu est de l'ordre de $t^p n^p$, tandis que le nombre de nœuds de l'arbre solution est de l'ordre de $t^{\frac{p}{2}}\sqrt{t^p n^p}$. On obtient à nouveau quelque chose de l'ordre de la racine carrée du nombre de nœuds de l'arbre de jeu ; la seule différence est le facteur $t^{\frac{p}{2}}$, plus petit que 1, qui laisse penser que l'existence de transpositions tend à faire diminuer la taille des arbres solution.

Mort subite

Pour estimer l'effet de la mort subite, étant donné un étage de l'arbre de jeu, on appelle $m \in [0;1]$ le taux de mort subite parmi les nœuds fils. Notons que m dépend de l'étage : en début de partie, il y a moins de positions de mat que plus bas dans l'arbre. Une fois m déterminé pour chaque étage (par exemple, par une formule), il est possible de calculer le nombre de nœuds de chaque étage de l'arbre de jeu, car si un étage père contient S nœuds, l'étage fils contient $S \times n$ nœuds fils, dont $S \times n \times m$ sont terminaux – et ainsi, l'étage encore en-dessous contient $S \times n^2 \times (1-m)$ nœuds fils.

On peut mener un calcul similaire pour l'arbre solution, dont les étages alternent entre nœuds perdants et gagnants (voir figure 14). Ainsi, si un étage père de l'arbre solution contient S nœuds perdants, il y a de même $S \times n$ nœuds fils gagnants, dont $S \times n \times m$ sont terminaux. Il y a cependant une différence : si l'étage père contient S nœuds gagnants, alors l'étage fils contient S nœuds perdants, dont $S \times m$ sont terminaux.

On peut même être plus efficace dans ce dernier cas, car lorsqu'un nœud gagnant a plusieurs fils perdants, on peut choisir en priorité un fils terminal, ce qui va permettre de réduire la taille de l'arbre solution. Pour traduire cela par le calcul, il faut commencer par estimer les proportions de nœuds gagnants et perdants dans l'arbre de jeu. Appelons g la probabilité qu'un nœud soit gagnant. Alors, la probabilité qu'un nœud soit perdant est $1-g$, mais aussi g^n car un nœud est perdant lorsque ses n fils sont gagnants. On obtient donc l'équation :

$$g^n = 1 - g \tag{3}$$

Cette équation a une seule solution entre 0 et 1, qui donne donc la probabilité qu'un nœud soit gagnant. Cette probabilité ne dépend que de n ; par exemple, pour $n = 18$ (valeur retenue dans notre estimation ci-après), on obtient 88,6 % de nœuds gagnants [19].

Un calcul explicité en annexe permet alors de déterminer que l'on peut choisir les S nœuds fils perdants de sorte que $S \times \frac{1}{g}(1-[1-m(1-g)]^n)$ de ces nœuds soient terminaux, ce qui est meilleur que la valeur initiale de $S \times m$.

Estimation

Nous allons maintenant utiliser ces trois modélisations pour obtenir l'estimation recherchée. Pour cela, il faut commencer par estimer les divers paramètres nécessaires à ces modélisations.

En premier lieu, des observations expérimentales permettent d'obtenir un ordre de grandeur du nombre moyen de coups jouables à partir d'une position. Par exemple, dans l'article [8], on parle de 35 ; dans le livre [5], il est dit être compris entre 30 et 40.

Cependant, François Labelle a calculé les 10 premiers étages de l'arbre de jeu des échecs [7], et le rapport du nombre de positions d'un étage sur l'autre semble être plutôt de l'ordre de 9 que de 35. Il y a deux raisons à cet écart : d'une part, les transpositions, et d'autre part, dans les 35 coups jouables à partir d'une position, certains ramènent à une position qu'il est possible d'obtenir en moins de coups. Ceci étant considéré, nous choisissons $n = 18$, et t =0,5 pour le taux de transpositions, ce qui donne bien 9 pour le rapport.

Quant à la mort subite, m évolue en fonction de la hauteur des étages : au début de l'arbre de jeu, le taux est nul, tandis qu'au dernier étage de l'arbre de jeu, il vaut 1. Entre les deux, l'évolution est difficile à estimer. La page [7] permet encore d'obtenir des valeurs sur les premiers étages, dont nous avons tiré la formule suivante :

$$m(h) = \frac{1+\tanh(h/8,5-5)}{2}$$

où h représente la distance de l'étage à la racine de l'arbre.

Reste à évaluer α, qui traduit les calculs inutiles effectués par l'algorithme alpha-bêta. Nous prendrons $\alpha = 2$, soit en moyenne, un nœud inutilement évalué avant que le nœud perdant soit trouvé, hypothèse nous semblant prudente.

L'utilisation conjointe de ces paramètres et des modélisations décrites précédemment donnent, après un calcul réalisable avec un tableur :

— $2,5 \times 10^{43}$ pour le nombre de positions de l'arbre de jeu.
— $3,1 \times 10^{12}$ pour le nombre de positions de l'arbre solution.

19. Il est logique que les nœuds gagnants soient plus nombreux : il suffit d'un fils perdant pour qu'un nœud soit gagnant, tandis qu'il faut que tous ses fils soient gagnants pour qu'il soit perdant, ce qui paraît plus difficile à obtenir.

— $7,1 \times 10^{18}$ pour le nombre de positions de la table de transpositions, en prenant $\alpha = 2$.

Cette estimation est bien sûr critiquable. Le nombre de positions de l'arbre solution paraît bien faible, la responsabilité en incombe aux transpositions dont l'effet paraît surestimé. Concernant les calculs inutiles effectués par l'algorithme alpha-bêta, la valeur $\alpha = 2$ est quelque peu artificielle, et là encore, son effet est probablement surestimé (ce qui compense l'influence des transpositions).

À l'inverse, l'effet de la mort subite est lui probablement sous-estimé : il ne compte que pour un facteur 5 dans la taille de l'arbre solution, et 11 dans la table de transpositions. En réalité, les nœuds terminaux ne sont pas répartis de manière homogène comme dans notre modélisation, si bien que concentrer l'étude sur les branches de l'arbre de jeu comprenant le plus de nœuds terminaux est plus efficace que ce que notre calcul laisse entendre.

Tout ceci étant considéré, et constatant que par rapport à l'estimation initiale de 10^{20} donnée à la section 7, seuls les calculs inutiles tirent ce nombre dans le mauvais sens, il nous paraît très vraisemblable qu'une table de transpositions capable de résoudre les échecs contienne entre 10^{15} et 10^{20} positions.

Voyons ce que cette estimation implique en terme de délai avant que le jeu d'échecs soit résolu. Le meilleur point de repère à notre disposition est la résolution du jeu de dames anglaises en 2007, pour lequel la quantité de calculs effectués était de l'ordre de 10^{14} [9]. Si l'on suppose que la loi de Moore, qui implique le gain d'un facteur 10 tous les 5 ans, continuera à être suivie, 10^{20} correspond à 2037 comme date limite pour la résolution du jeu d'échecs.

Conclusion

Cet article a permis de mettre à jour certaines idées reçues quant à la possibilité de résoudre le jeu d'échecs. Trop souvent, quand on imagine le fonctionnement des programmes spécialisés dans les jeux combinatoires, on voit le calcul comme un problème de force brute, et donc on surestime la puissance de calcul nécessaire ; et à l'inverse, on sous-estime l'intérêt et l'efficacité des techniques développées pour simplifier les calculs. Nous avons ainsi montré que des techniques pourtant très basiques dans l'étude des jeux combinatoires permettent de faire redescendre la complexité du jeu d'échecs de 10^{120} à un nombre aux alentours de 10^{20}.

Et de fait, les progrès qui ont permis aux ordinateurs de battre les joueurs humains aux échecs sont à créditer au moins autant à l'amélioration des méthodes de calcul qu'à l'augmentation de la puissance des ordinateurs. Notre estimation de la date de résolution du jeu d'échecs est imprécise, avec des faiblesses que nous avons déjà pointées, mais de plus, nous n'avons pas tenu compte d'éventuels progrès des méthodes de calcul : la date pourrait ainsi être encore plus proche qu'escomptée.

Pour en avoir le cœur net, il est nécessaire de commencer la recherche sur le sujet, se confronter au problème étant le meilleur moyen d'en prendre la mesure.

Références

[1] Louis Victor Allis. A knowledge based approach to connect-four. the game is solved. Master's thesis, University of Vrije, Amsterdam, 1988.

[2] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.

[3] Henry Bottomley. How many tic-tac-toe (noughts and crosses) games are possible ? `http://www.se16.info/hgb/tictactoe.htm`, 2001.

[4] Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Universiteit Maastricht, 1998.

[5] Tristan Cazenave, Mark H.M. Winands, and Hiroyuki Iida. Computer games. *Communications in Computer and Information Science*, 408, 2014.

[6] Kevin Crowley and Robert S. Siegler. Flexible strategy use in young children's tic-tac-toe. *Cognitive Science*, 17(4) :531–561, 1993.

[7] François Labelle. Statistics on chess positions. `http://wismuth.com/chess/statistics-positions.html`, 2012.

[8] Alan Levinovitz. The mystery of go, the ancient game that computers still can't win. `http://www.wired.com/2014/05/the-world-of-computer-go`, 2014.

[9] Jonathan Schaeffer and al. Checkers is solved. *Science*, 317 :1518–1522, 2007.

[10] Claude E. Shannon. Programming a computer for playing chess. *Philosophical magazine*, 41, 1950.

[11] Suhas Sreedhar. Checkers, solved ! `http://spectrum.ieee.org/computing/software/checkers-solved`, 2007.

[12] John Tromp. John's chess playground. `http://tromp.github.io/chess/chess.html`, 2010.

[13] Alan Turing. Digital computers applied to games. 1953.

[14] H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved : Now and in the future. *Artificial Intelligence*, 134(1-2) :277–311, January 2002.

Annexe : Détail des calculs pour les estimations

Calculs inutiles

Soit un arbre de jeu équilibré dont chaque nœud interne a n fils, et dont les parties se terminent en p coups. En moyenne, au lieu d'un unique fils, l'algorithme explore α fils avec $\alpha \in [1;n]$ (ce qui traduit les calculs inutiles). Alors le nombre de nœuds étudiés est :

$$\begin{aligned} & 1+\alpha+\alpha n+\alpha^2 n+\alpha^2 n^2+\alpha^3 n^2+\ldots+\alpha^{\frac{p+1}{2}} n^{\frac{p-1}{2}} \\ &= (1+\alpha)+(1+\alpha)n+(1+\alpha)n^2+\ldots+(1+\alpha)n^{\frac{p-1}{2}} \\ &= (1+\alpha)\frac{(\alpha n)^{\frac{p+1}{2}}-1}{\alpha n-1} \sim (1+\alpha)(\alpha n)^{\frac{p-1}{2}} \simeq \alpha^{\frac{p}{2}}\sqrt{n^p} \end{aligned} \tag{4}$$

Transpositions

Soit un arbre de jeu équilibré dont chaque nœud interne a n fils, dont les parties se terminent en p coups, et comportant des transpositions uniformément réparties : quand on passe d'un étage au suivant, le nombre de nœuds de l'étage est multiplié par tn avec $t \in]0;1]$. Le nombre de nœuds de l'arbre de jeu est alors :

$$1+tn+t^2n^2+\ldots+t^pn^p = \frac{(tn)^{p+1}-1}{tn-1} \sim t^pn^p \tag{5}$$

Quant au nombre de nœuds de l'arbre solution, il vaut :

$$\begin{aligned} & 1+t+t^2n+t^3n+t^4n^2+t^5n^2+\ldots+t^pn^{\frac{p-1}{2}} \\ &= (1+t)+(t^2n)(1+t)+(t^2n)^2(1+t)+\ldots+(t^2n)^{\frac{p-1}{2}}(1+t) \\ &= (1+t)\frac{(t^2n)^{\frac{p+1}{2}}-1}{(t^2n)-1} \sim (1+t)t^{p-1}n^{\frac{p-1}{2}} \simeq t^pn^{\frac{p}{2}} = t^{\frac{p}{2}}\sqrt{t^pn^p} \end{aligned} \tag{6}$$

Mort subite

Nous étudions ici l'effet du choix prioritaire d'un fils terminal parmi les fils perdants d'un nœud gagnant. Pour commencer, nous allons utiliser la relation suivante :

$$\sum_{i=0}^{n} \binom{n}{i} g^i(1-g)^{n-i} = 1 \tag{7}$$

Sa démonstration est triviale, il suffit d'appliquer la formule du binôme à $(g+(1-g))^n$. Remarquons que le terme numéro i de cette somme traduit la probabilité qu'un nœud ait i fils gagnants, et donc $n-i$ fils perdants. En particulier, le terme numéro n est la probabilité que le nœud soit perdant (car tous ses fils sont gagnants), et vaut g^n, soit $1-g$ par la relation (3). La somme des $n-1$ autres termes vaut donc g.

Maintenant, nous pouvons calculer la proportion de nœuds gagnants possédant au moins un fils perdant terminal :

$$\frac{1}{g}\sum_{i=0}^{n-1}\binom{n}{i}g^i(1-g)^{n-i}[1-(1-m)^{n-i}] \tag{8}$$

En effet, le nœud étant gagnant, on enlève le terme numéro n de la relation (7), et on divise par la somme des autres termes, d'où le facteur $\frac{1}{g}$. Pour justifier le crochet, on remarque que s'il y a $n-i$ fils perdants, la probabilité qu'aucun ne soit terminal est $(1-m)^{n-i}$, donc la probabilité qu'on puisse en choisir un terminal est $[1-(1-m)^{n-i}]$.

Remarquant que le crochet s'annule si $i=n$, la somme (8) devient :

$$\begin{aligned}
&\frac{1}{g}\sum_{i=0}^{n}\binom{n}{i}g^i(1-g)^{n-i}[1-(1-m)^{n-i}]\\
&=\frac{1}{g}\sum_{i=0}^{n}\binom{n}{i}g^i(1-g)^{n-i}-\frac{1}{g}\sum_{i=0}^{n}\binom{n}{i}g^i[(1-g)(1-m)]^{n-i}\\
&=\frac{1}{g}[g+(1-g)]^n-\frac{1}{g}[g+(1-g)(1-m)]^n\\
&=\frac{1}{g}\left(1-[1-m(1-g)]^n\right) \qquad (9)
\end{aligned}$$

Ainsi, comme annoncé, on peut choisir les S nœuds fils perdants de sorte que $S\times\frac{1}{g}\left(1-[1-m(1-g)]^n\right)$ de ces nœuds soient terminaux.

Remarquons que l'inégalité de Jensen permet logiquement de constater que l'expression (9) est supérieure à m, et donc que ce choix réalise une amélioration par rapport à la méthode initiale :

$$\begin{aligned}
(1-m).1^n+m.g^n &\geqslant [(1-m).1+m.g]^n\\
1-m+mg^n &\geqslant [1-m+mg]^n\\
1-m(1-g^n) &\geqslant [g+1-m-g+mg]^n\\
1-mg &\geqslant [1-m(1-g)]^n\\
m &\leqslant \frac{1}{g}\left(1-[1-m(1-g)]^n\right) \qquad (10)
\end{aligned}$$