Apprendre à jouer aux jeux à deux joueurs à information parfaite sans connaissance

Quentin Cohen-Solal

Apprendre à jouer aux jeux à deux joueurs à information parfaite sans connaissance

Quentin Cohen-Solal
CRIL, Univ. Artois et CNRS
62300 Lens, France
cohen-solal@cril.fr

Résumé

Dans cet article, plusieurs techniques pour l'apprentissage par renforcement de fonctions d'évaluation d'états de jeu sont proposées. La première consiste à apprendre les valeurs de l'arbre de jeu au lieu de se restreindre à la valeur de la racine. La seconde consiste à remplacer le gain classique d'un jeu ($+1$ / -1) par une heuristique favorisant les victoires rapides et les défaites lentes. La troisième permet de corriger certaines fonctions d'évaluation en tenant compte de la résolution des états. La quatrième est une nouvelle distribution de sélection d'actions. Enfin, la cinquième est une modification du minimax à profondeur non bornée étendant les meilleures séquences d'actions jusqu'aux états terminaux. En outre, nous proposons une autre variante du minimax non borné, qui joue l'action la plus sure au lieu de jouer la meilleure action. Les expériences menées suggèrent que cela améliore le niveau de jeux lors des confrontations. Enfin, nous appliquons ces différentes techniques pour concevoir un programme-joueur au jeu de Hex (taille 11*) atteignant le niveau de Mohex 2.0 à la suite d'un apprentissage par renforcement contre soi-même sans utilisation de connaissance.*

Mots Clef

Décision séquentielle, Jeux, Planification, Apprentissage, Renforcement, Minimax non borné.

Abstract

In this paper, several techniques for learning game states evaluation functions by reinforcement are proposed. The first is to learn the values of the game tree instead of restricting oneself to the value of the root. The second is to replace the classic gain of a game ($+1$ / -1) with a heuristic favoring quick wins and slow defeats. The third corrects some evaluation functions taking into account the resolution of states. The fourth is a new action selection distribution. Finally, the fifth is a modification of the minimax with unbounded depth extending the best sequences of actions to the terminal states. In addition, we propose another variant of the unbounded minimax, which plays the safest action instead of playing the best action. The experiments conducted suggest that this improves the level of play during confrontations. Finally, we apply these different techniques to design a program-player to the Hex game (size 11) reaching the level of Mohex 2.0 with reinforcement learning from self-play without knowledge.

Keywords

Sequential Decision, Games, Planning, Learning, Reinforcement, Unbound Minimax.

1 Introduction

Une des tâches les plus difficiles en intelligence artificielle est la *prise de décision séquentielle* [18], dont les applications inclues la robotique et les jeux. Concernant les jeux, les succès sont nombreux. La machine dépasse l'homme pour plusieurs jeux, tels que le backgammon, les dames, les échecs et le go [31]. Une classe importante de jeux constitue les jeux à deux joueurs à *information parfaite*, c'est-à-dire les jeux où les joueurs jouent à tour de rôle, sans hasard ni information cachée. Il reste encore de nombreux défis pour ces jeux. Par exemple, pour le *jeu de Hex*, l'homme reste toujours supérieur à la machine. C'est également le cas du *general game playing* [31] (même restreint aux jeux à information parfaite) : l'homme est toujours supérieur à la machine sur un jeu inconnu (lorsque l'homme et la machine ont un temps d'apprentissage relativement court pour maîtriser les règles du jeu). Dans cet article, nous nous concentrons sur les jeux à deux joueurs à information parfaite et à somme nulle, bien que la plupart des contributions de cet article devraient pouvoir s'appliquer ou s'adapter aisément à un cadre plus général.
Les premières approches utilisées pour concevoir un programme capable de jouer à un jeu se basent sur un algorithme de recherche dans les arbres de jeu, tel que le *minimax*, combiné à une fonction d'évaluation des états de jeu, conçue à partir de connaissances expertes. L'avénement de cette technique est le programme Deep Blue [31] au jeu d'échecs. Cependant, le succès de Deep Blue tient en grande partie à la puissance brute de sa machine, lui permettant d'analyser deux cents millions états de jeu par seconde. De plus, cette approche est limitée par le fait de devoir concevoir une fonction d'évaluation manuellement (au moins partiellement). Cette conception est une tâche très complexe, qui doit, en plus, être réalisée pour chaque

jeu. Plusieurs travaux se sont donc focalisés sur l'apprentissage automatique de fonctions d'évaluation [19]. L'un des premiers succès de l'apprentissage de fonctions d'évaluation est sur le jeu Backgammon [31]. Cependant, pour de nombreux jeux, comme le Hex ou le Go, les approches basées sur le minimax, avec ou sans apprentissage automatique, n'ont pas permis de dépasser l'homme. Deux causes ont été identifiées [2]. Premièrement, le très grand nombre d'actions possibles à chaque état de jeu empêche d'effectuer une recherche exhaustive à une profondeur significative (le cours de la partie ne peut être anticipé qu'à un petit nombre de tours à l'avance). Deuxièmement, pour ces jeux, aucune fonction d'évaluation suffisamment performante n'a pu être identifiée. Une approche alternative permettant de résoudre ces deux problèmes a été proposée, donnant notamment de bons résultats au Hex et au Go, nommée recherche arborescente Monte Carlo et notée MCTS (pour Monte Carlo Tree Search) [5]. Cet algorithme explore l'arbre de jeu non uniformément, ce qui constitue une solution au problème du très grand nombre d'actions. De plus, il évalue les états de jeu à partir de statistiques de victoire provenant d'un grand nombre de simulations aléatoires de fin de partie. Il n'a ainsi pas besoin de fonction d'évaluation. Cela n'était cependant pas suffisant pour dépasser le niveau des joueurs humains. Plusieurs variantes de la recherche arborescente Monte Carlo ont alors été proposées, utilisant notamment des connaissances pour orienter l'exploration de l'arbre de jeu et/ou les simulations aléatoires de fin de partie [5]. Les récentes améliorations de la recherche arborescente Monte Carlo ont porté sur l'apprentissage automatique des connaissances du MCTS et leurs utilisations. Ces connaissances ont d'abord été générées par *apprentissage supervisé* [7, 8, 9, 6] puis par apprentissage supervisé suivi d'un *apprentissage par renforcement* [25], et enfin par apprentissage par renforcement seulement [27, 26, 1]. Cela a permis d'atteindre et de dépasser le niveau de champion du monde au jeu de Go avec les dernières versions du programme Alphago [25, 27]. En particulier, Alphago zero [27], qui n'utilise que l'apprentissage par renforcement, n'a donc eut besoin d'aucune connaissance initiale, humaine, pour atteindre son niveau de jeu. Ce dernier succès a toutefois nécessité 29 millions de parties et plus de 6 milliards évaluations d'états de jeu, ce qui est très largement supérieur à ce qu'il a fallu au champion humain du Go pour atteindre son niveau de jeu. Cette approche a également été appliquée aux échecs [26]. Le programme résultant a battu le meilleur programme des échecs, qui était toujours basé sur le minimax.

On peut dès lors se demander si les approches de type minimax sont totalement dépassées ou si les succès spectaculaires des récents programmes tiennent plus de l'apprentissage par renforcement que de la recherche arborescente Monte Carlo. En particulier, on peut se demander si l'apprentissage par renforcement permettrait à une approche de type minimax de rivaliser avec la recherche arborescente Monte Carlo sur les jeux où celle-ci domine le minimax jusqu'à présent, tels que le Go ou le Hex.

Dans cet article, nous nous focalisons par conséquent sur l'apprentissage par renforcement dans le cadre du minimax. Nous proposons de nouvelles techniques pour l'apprentissage par renforcement de fonctions d'évaluation. Nous les appliquons pour concevoir un nouveau programme-joueur au jeu de Hex (sans utiliser d'autres connaissances que les règles du jeu). Nous comparons en particulier ce programme-joueur à Mohex 2.0 [13], le champion au Hex (taille 11) lors des Olympiades informatiques de 2013 à 2017 [11], qui est également le programme-joueur le plus fort publiquement disponible.

Dans la section 2, nous présentons succinctement les algorithmes de jeux et en particulier le minimax à profondeur non bornée sur lequel nous basons plusieurs de nos expériences. Nous présentons également l'apprentissage par renforcement dans les jeux, le jeu de Hex et l'état de l'art des programmes-joueurs sur ce jeu. Dans la section suivante, nous proposons différentes techniques ayant pour objectif d'améliorer l'apprentissage. Ensuite, nous exposons dans la section 4 les expériences réalisées utilisant ces techniques. Enfin, dans la dernière section, nous concluons et exposons les différentes perspectives de recherche.

2 Contexte et travaux connexes

Dans cette section, nous présentons succinctement les algorithmes de recherche dans les arbres de jeux, l'apprentissage par renforcement dans le contexte des jeux ainsi que leurs applications dans le cadre du jeu de Hex (pour plus de détails sur les algorithmes de jeux, voir [31]).

Les jeux peuvent être représentés de manière arborescente par leur *arbre de jeu* (un nœud correspond à un état de jeu et les fils d'un nœud sont les états atteignables par une action). À partir de cette représentation, on peut déterminer l'action à jouer en utilisant un algorithme de recherche dans l'arbre de jeu. Afin de gagner, chaque joueur cherche à maximiser son score (i.e. la valeur de l'état du jeu pour ce joueur à la fin de la partie). Comme nous nous plaçons dans le cadre des jeux à deux joueurs à somme nulle, maximiser le score d'un joueur revient à minimiser le score de son adversaire (le score d'un joueur est la négation du score de son adversaire).

2.1 Recherche dans les arbres de jeu

L'algorithme central est le *minimax* qui détermine récursivement la valeur d'un nœud à partir de la valeur de ses fils et des fonctions $\min$ et $\max$, jusqu'à une profondeur limite de récursion. Avec cet algorithme, l'arbre de jeu est exploré uniformément. Une implémentation plus performante du minimax utilise l'*élagage alpha-bêta* [31] qui permet de ne pas explorer les sections de l'arbre de jeux qui sont moins intéressantes étant données les valeurs des nœuds déjà rencontrés et les propriétés du $\min$ et du $\max$. De nombreuses variantes et améliorations du minimax ont été proposées [21]. Certaines de ces variantes effectuent une recherche à profondeur non bornée (c'est-à-dire que la profondeur de

leur recherche n'est pas fixée) [30]. Contrairement au minimax avec ou sans élagage alpha-bêta, l'exploration de ces algorithmes est non uniforme. L'un de ces algorithmes est la *recherche minimax en meilleur d'abord* [16]. Pour éviter toute confusion avec certaines approches en meilleur d'abord à profondeur fixée, nous appelons cet algorithme le *minimax en meilleur d'abord non borné*, ou plus succinctement UBFM (Unbound Best-First Minimax). UBFM étend itérativement l'arbre de jeu en ajoutant les fils de l'une des feuilles de l'arbre de jeu ayant la même valeur que celle de la racine (valeur minimax). Ces feuilles sont les états obtenus après avoir joué une des meilleures séquences d'actions possibles étant donnée la connaissance actuelle partielle de l'arbre de jeu. Ainsi, cet algorithme étend itérativement les meilleures séquences d'actions *a priori*. Ces meilleures séquences changent généralement à chaque extension. Cela permet d'explorer non uniformément l'arbre de jeu en se focalisant sur les actions les plus intéressantes *a priori* sans pour autant n'explorer qu'une seule séquence d'actions. Dans cet article, nous utilisons la version anytime d'UBFM [16], c'est-à-dire que nous laissons un temps de réflexion fixé à UBFM pour déterminer l'action à jouer. Nous utilisons en plus les tables de transposition [21] avec UBFM, ce qui permet de ne pas avoir à construire explicitement l'arbre de jeu et de fusionner les nœuds correspondant au même état.

2.2 Apprentissage de fonctions d'évaluation

L'apprentissage par renforcement de fonctions d'évaluation peut être effectué par différentes techniques [19, 27, 1, 32]. L'idée générale de l'apprentissage par renforcement de fonctions d'évaluation d'états est d'utiliser un algorithme de recherche dans les arbres de jeu et une fonction d'évaluation adaptative f (par exemple un réseau de neurones) pour jouer une séquence de parties (par exemple contre soi-même, ce qui est le cas dans cet article). Chaque partie va générer des couples (e, v) où e est un état et v la valeur de e calculée par l'algorithme de recherche choisi utilisant la fonction d'évaluation f. L'apprentissage consiste alors à modifier f de façon à ce que pour tous les couples (e, v) générés $f(e)$ se rapproche de v suffisamment pour en constituer une bonne approximation. Au lieu de restreindre l'apprentissage aux données générées lors de la dernière partie, les données des dernières parties peuvent être gardées en mémoire et utilisées lors de l'apprentissage (cette technique est appelée *experience replay* [22]). Cependant, dans cet article, les données utilisées lors d'un apprentissage sont celles générées lors de la dernière partie.

Remarque 1. Les fonctions d'évaluation adaptatives ne servent qu'à évaluer les états non terminaux puisque l'on connait la vraie valeur des états terminaux.

2.3 Distribution de sélection d'actions

Un des problèmes lié à l'apprentissage par renforcement est le *dilemme exploration-exploitation* [19]. Celui-ci consiste à choisir entre explorer de nouveaux états pour apprendre de nouvelles connaissances et exploiter les connaissances acquises. De nombreuses techniques ont été proposées pour gérer ce dilemme [20]. Cependant la plupart de ces techniques ne passent pas à l'échelle car leur application nécessite de mémoriser tous les états rencontrés. Pour cette raison, dans le cadre des jeux à grand nombre d'états, les travaux se basent sur une exploration probabiliste [32, 27, 19]. Avec cette approche, exploiter, c'est jouer la meilleure action et explorer, c'est jouer au hasard uniformément. Une distribution de probabilité paramétrique est alors utilisée pour associer à chaque action sa probabilité d'être jouée. Le paramètre associé à la distribution correspond au taux d'exploration compris entre 0 et 1, que nous notons ϵ (le taux d'exploitation est donc $1 - \epsilon$, que nous notons ϵ'). Celui est souvent fixé expérimentalement. Un *recuit simulé* [15] peut cependant être appliqué afin de ne pas avoir besoin de choisir une valeur pour ce paramètre. Dans ce cas, au début de l'apprentissage par renforcement, le paramètre vaut 1 (on ne fait qu'explorer). Celui-ci diminue au fur et à mesure jusqu'à atteindre 0 à la fin de l'apprentissage. La distribution de sélection d'actions la plus simple est ϵ-*greedy* [32] (de paramètre ϵ). Avec cette distribution, l'action est choisie uniformément avec probabilité ϵ et la meilleure action est choisie avec probabilité $1 - \epsilon$. La distribution ϵ-greedy a le désavantage de ne pas différencier les actions (hormis la meilleure action) en termes de probabilités. Une autre distribution est souvent utilisée, corrigeant cette inconvénient. Il s'agit de la fonction *softmax*, également appelée *distribution de Boltzmann* [19].

2.4 Jeu de Hex

Le jeu de Hex est un jeu de stratégie combinatoire pour deux joueurs. Il se joue sur un plateau hexagonal $n \times n$ vide. On dira qu'un plateau $n \times n$ est de taille n. Le plateau peut être de taille quelconque, bien que les tailles classiques soient 11, 13 et 19. A son tour, chaque joueur pose une pièce de sa couleur sur une case vide (chaque pièce est identique). Le but du jeu est d'être le premier à relier les deux bords opposés du plateau correspondant à sa couleur. La figure 1 illustre une fin de partie. Bien que ces règles soient simplistes, les tactiques et stratégies au jeu de Hex sont complexes. Le nombre d'états et le nombre d'actions par état sont très importants, similaires au jeu de Go. Le nombre d'états est par exemple supérieur à celui des échecs, au moins dès le plateau de taille 11 (table 6 de [29]). Quelle que soit la taille du plateau, le premier joueur a une stratégie gagnante [3] qui est inconnue, sauf pour les plateaux de taille inférieure ou égale à 9 [23]. En fait, résoudre un état particulier est PSPACE-complet [24, 4].

2.5 Programmes-joueurs au Hex

De nombreux programmes-joueurs ont été développés au Hex. Par exemple, Mohex 1.0 [13] est un programme basé sur la recherche arborescente Monte Carlo. Il utilise en plus de nombreuses techniques dédiées au Hex, basées sur des résultats théoriques spécifiques au jeu de Hex. Il est en particulier capable de déterminer une stratégie gagnante pour

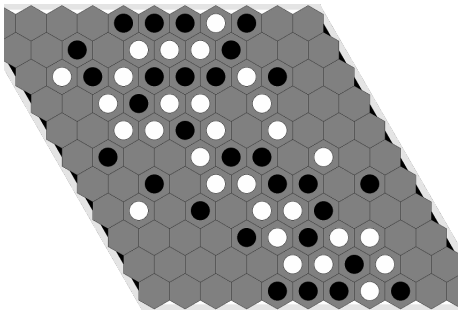

FIGURE 1 – Fin de partie au Hex, taille 11 (blanc gagne)

certains états et d'élaguer à chaque état de nombreuses actions qu'il sait être *inférieures*. Il utilise également des connaissances ad-hoc pour biaiser les simulations de la recherche arborescente Monte Carlo.

Mohex 2.0 [13] est une amélioration de Mohex 1.0 qui utilise des connaissances apprises par apprentissage supervisé pour orienter à la fois l'exploration de l'arbre et les simulations de la recherche arborescente Monte Carlo. Cet apprentissage supervisé a permis d'apprendre les corrélations de victoire de motifs du plateau.

D'autres travaux se sont ensuite focalisés sur la prédiction des meilleures actions, par apprentissage supervisé d'une base de données de parties, à l'aide d'un réseau de neurones. Le réseau de neurones est utilisé pour apprendre une *politique*, c'est-à-dire une distribution de probabilité a priori sur les actions à jouer. Ces probabilités a priori sont utilisées pour guider l'exploration de la recherche arborescente Monte Carlo. Il y a d'abord Mohex-CNN [8] qui est une amélioration de Mohex 2.0 utilisant un réseau de neurones convolutionnel [17]. Une nouvelle version de Mohex a ensuite été proposée : Mohex-3HNN [9]. Contrairement à Mohex-CNN, il est basé sur un réseau de neurones résiduel [12]. Il calcule, en plus de la politique, une valeur pour les états *et* les actions. La valeur des états remplace l'évaluation des états à base de simulations de la recherche arborescente Monte Carlo. L'ajout d'une valeur pour les actions permet à Mohex-HNN de diminuer le nombre d'appel du réseau de neurones, améliorant ainsi les performances.

Des programmes apprenant la fonction d'évaluation par renforcement ont également été conçus. Ces programmes sont NeuroHex [32], EZO-CNN [28] et ExIt [1]. Ils apprennent en jouant contre eux-même. Contrairement aux deux autres, NeuroHex effectue au préalable un apprentissage supervisé afin de diminuer le temps d'apprentissage par renforcement (en apprenant une heuristique commune du jeu de hex). NeuroHex démarre en plus ses parties d'un état provenant d'une base de données de parties. EZO-CNN, lui, utilise durant l'apprentissage des connaissances lui permettant de jouer (et donc d'apprendre) une stratégie gagnante dans certains états. ExIt, quand à lui, apprend une politique en plus de la valeur des états de jeu. Il est le seul programme a avoir appris à jouer au Hex sans utiliser de connaissances. Ce résultat est cependant limité aux plateaux de taille 9. Un comparatif des caractéristiques principales de ces différents programmes est présenté dans la table 1.

3 Techniques proposées

Nous proposons maintenant plusieurs techniques visant à améliorer et/ou accélérer l'apprentissage de fonctions d'évaluation d'états de jeu. Nous terminons cette section en proposant une légère modification d'UBFM que nous appelons UBFM_s afin d'améliorer le niveau de jeu.

3.1 Apprentissage de l'arbre de jeu

Avec les différents travaux de la littérature sur l'apprentissage par renforcement de fonctions d'évaluation, les états e des couples (e, v) utilisés lors de l'apprentissage sont les différents états du jeu obtenus au cours de la partie (il ne s'agit pas des états générés lors de la recherche dans l'arbre de jeu). Chaque valeur v est déterminée par l'algorithme de recherche utilisé. Nous appelons cette technique le *root learning*. Cette approche perd ainsi une partie des informations contenues dans l'arbre de jeu générées lors de la recherche de l'action à jouer. Plus précisément, les valeurs des états de l'arbre de jeu sont perdues puisqu'elles ne sont pas utilisées lors de l'apprentissage de la fonction d'évaluation. Nous proposons donc d'apprendre l'intégralité de l'arbre partiel de jeu (à l'exception des feuilles non terminales puisque l'on a déjà $f(e) = v$) au lieu de se restreindre à apprendre la valeur de la racine. Nous appelons cette technique le *tree learning*.

Le tree learning peut sembler moins intéressant que le root learning puisqu'il ajoute à l'apprentissage des valeurs de précision inférieure à celle de la racine (puisque moins d'états sont générés pour les déterminer). Il est de plus possible qu'un certain nombre d'états de l'arbre de jeu n'aient jamais été explorés depuis le début de l'apprentissage par renforcement. Des valeurs abberantes seraient alors apprises continuellement. Ces valeurs supplémentaires ont un gros risque de biaiser significativement l'apprentissage par renforcement. Il y a donc un dilemme entre le gain en information qui peut être apporté par l'apprentissage de ces valeurs supplémentaires et le risque de biaiser l'apprentissage dû à l'imprécision de ces valeurs.

Afin de réduire le biais dû à l'apprentissage de ces valeurs, nous proposons une technique supplémentaire que nous appelons *coefficiented tree learning*. Cette technique affecte à chaque état un coefficient c indiquant à quel point sa valeur est précise. Les valeurs avec un coefficient plus grand doivent alors être préférentiellement mieux apprises que celles avec un faible coefficient. Pour implémenter cette idée, lors des expériences de cet article, le coefficient est calculé par la formule suivante : $c = 1 + \lfloor \log_2(n+1) \rfloor$ avec n le nombre de fois que la valeur de cet état a été mise à jour au cours de la recherche (n est donc le nombre de nœuds descendants du nœud de cet état). Pour tenir compte des coefficients lors de l'apprentissage, nous dupliquons c fois chaque couple (e, v) où c est le coefficient de ce

Programmes	Tailles	Recherche	Apprentissage	Réseau	Utilisation
Mohex-CNN	13	MCTS	supervisé	convolutionnel	politique
Mohex-3HNN	13	MCTS	supervisé	résiduel	politique, état, action
NeuroHex	13	aucun	supervisé et renforcement	convolutionnel	état
EZO-CNN	7, 9, 11	Minimax	renforcement	convolutionnel	état
ExIt	9	MCTS	renforcement	convolutionnel	politique et état

TABLE 1 – Comparatif des caractéristiques principales des derniers programmes au Hex. Ces caractéristiques sont respectivement les tailles de plateau sur lesquelles ils peuvent jouer, l'algorithme de recherche dans les arbres utilisé, le type d'apprentissage effectué, le type de réseau de neurones utilisé et l'utilisation du réseau de neurones qui en est fait, c'est-à-dire s'il est utilisé pour approximer les valeurs des états, des actions et/ou de la politique.

couple.

Exemple 2. Suite à une partie, nous obtenons les couples suivant $(e_1, 1)$, $(e_2, -2)$ et $(e_3, 5)$. Les états e_1, e_2 et e_3 ont respectivement 0, 1 et 3 états descendants dans l'arbre de jeu. Le coefficient de e_1 (resp. e_2 ; resp. e_3) est donc 1 (resp. 2 ; resp. 3). Les données utilisées lors de l'apprentissage de la fonction d'évaluation adaptative f sont les couples suivants $(e_1, 1)$, $(e_2, -2)$, $(e_2, -2)$, $(e_3, 5)$, $(e_3, 5)$, $(e_3, 5)$.

Notons qu'il y a une autre approche dans la littérature, utilisée par alphago zéro [27]. Avec celle-ci, les états des couples (e, v) sont toujours les états obtenus au cours de la partie, mais leur valeur est le résultat de la partie (la valeur de l'état terminal de la partie). Nous appelons cette technique le *terminal learning*. Elle a l'avantage de ne pas apprendre des valeurs approximées et donc de ne pas approximer des approximations. Cependant, il y a toujours un biais, puisque la valeur terminale n'est en général pas la vrai valeur de chaque état de la partie.

3.2 Heuristique de la profondeur

Dans cette section, nous proposons une fonction d'évaluation des états terminaux alternative à la fonction d'évaluation classique des états terminaux, qui retourne 1 si le joueur gagne et -1 si son adversaire gagne [32, 27, 9]. Nous l'appelons *heuristique de la profondeur* et nous la notons $\mathrm{p_t}$. Elle donne une meilleure valeur aux états gagnants proches du début de partie qu'aux états gagnants loin du début de partie. Elle donne également une meilleur valeur aux états perdants loin du début de partie qu'aux états perdants proches du début de partie. Avec cette fonction d'évaluation, on cherche à gagner le plus tôt possible et perdre le plus tard possible. En apprenant par renforcement avec cette fonction d'évaluation terminale, on préférera ainsi un état que l'on pense gagnant et proche de la fin de partie à un autre état que l'on pense gagnant et loin de la fin de partie. On peut espérer que cela améliore le niveau de jeu. Un état que l'on pense gagnant et proche de la fin de partie a moins de chances d'être un état perdant qu'un état que l'on pense gagnant et loin de la fin de partie. En effet, plus un état est proche de la fin de partie, plus il y a de chances que sa valeur soit précise. De plus, avec une partie longue, un joueur en difficulté aura plus d'occasions de reprendre le dessus. Inversement, un état que l'on pense perdant et loin de la fin de partie sera préféré à un état que l'on pense perdant et proche de la fin de la partie, puisque le premier état a hypothétiquement une valeur moins précise et a donc *a priori* plus de chances d'être en réalité un état gagnant.
Nous proposons comme réalisation de l'heuristique de la profondeur la fonction $\mathrm{p_t}$ suivante. La fonction d'évaluation $\mathrm{p_t}$ retourne la valeur l si le joueur gagne et la valeur $-l$ si son adversaire gagne, avec l le nombre maximum d'actions réalisables au cours d'une partie plus 1 moins le nombre d'actions jouées depuis le début de la partie. Pour le Hex, l est le nombre de cases vides du plateau plus 1.

3.3 Complétion de fonctions d'évaluation

Nous introduisons dans cette section la complétion C d'une fonction d'évaluation d'états de jeu f. La complétion a pour objectif d'améliorer la fonction d'évaluation en tenant compte de la résolution des états. Un état est dit *résolu* si l'algorithme de recherche a permis de déterminer la vraie valeur de cet état. Sans la complétion, on peut avoir un état résolu et gagnant de valeur plus faible qu'un état non résolu que l'on pense gagnant mais qui est peut-être perdant. L'objectif de la complétion est donc de toujours choisir une action menant à un état résolu gagnant et de ne jamais choisir une action menant à un état résolu perdant lorsque l'on a le choix (lors de l'exploration de l'arbre ou de la décision de l'action à jouer). La fonction de complétion peut se définir par $C(f(e)) = (r(e), f(e))$ avec e un état et r la fonction suivante qui retourne 1 si l'état est résolu et si le joueur gagne, -1 si l'état est résolu et si son adversaire gagne et 0 si l'état n'est pas résolu. On utilisera alors l'ordre lexicographique pour comparer les états.

Remarque 3. Bien que la complétion remplace la fonction d'évaluation lors de l'exploration et lors du choix de l'action à jouer, les couples utilisés lors de l'apprentissage restent les couples de la forme $(e, f(e'))$.

Il n'y a pas besoin de compléter une fonction d'évaluation d'états non terminaux si elle est à valeurs dans $]a, b[$ (avec $a < b$) et si les états terminaux sont évalués par a s'ils sont perdants et par b s'ils sont gagnants.

3.4 Descente : générer de meilleures données

Nous introduisons dans cette section une modification d'UBFM, que nous nommons *descente*, visant à être uti-

lisée lors de l'apprentissage par renforcement afin de générer des couples (e, v) différents mais supposément de meilleure qualité. L'idée de *descente* est de combiner UBFM avec des simulations déterministes de fin de partie apportant des valeurs intéressantes du point de vue de l'apprentissage. L'algorithme *descente* (algorithme 1) sélectionne récursivement le meilleur fils du nœud actuel, qui devient le nouveau nœud actuel. Il ajoute les fils du nœud actuel s'ils ne sont pas dans l'arbre. Il effectue cette récursion partant de la racine (l'état actuel du jeu) jusqu'à atteindre un nœud terminal (une fin de partie). Il met alors à jour la valeur des nœuds sélectionnés (valeur minimax). L'algorithme *descente* recommence cette opération récursive repartant de la racine tant qu'il reste du temps de recherche. *Descente* est quasiment identique à UBFM. La seule différence est que *descente* effectue une iteration jusqu'à atteindre un état terminal alors qu'UBFM effectue cette itération jusqu'à atteindre une feuille de l'arbre (UBFM arrête donc l'itération beaucoup plus tôt). Autrement dit, lors d'une itération, UBFM étend juste une des feuilles de l'arbre de jeu alors que *descente* étend récursivement le meilleur des fils en partant de cette feuille jusqu'à atteindre une fin de partie. L'algorithme *descente* a l'avantage d'UBFM, c'est-à-dire d'effectuer une recherche plus longue permettant de déterminer une meilleure action à jouer. Grâce au tree learning, il a également l'avantage d'une recherche minimax à profondeur 1, c'est-à-dire de faire remonter les valeurs des nœuds terminaux vers les autres nœuds plus rapidement. En outre, les états ainsi générés sont plus proches des états terminaux. Leurs valeurs sont donc de meilleures approximations.

3.5 Distribution ordinale d'actions

Dans cette section, nous proposons une distribution de probabilité alternative à la fonction softmax et à ϵ-greedy (voir la section 2.3). Cette distribution ne dépend pas de la valeur des états. Elle dépend par contre de l'ordre de leurs valeurs. Sa formule est la suivante :

$$P(f_i) = \left(\epsilon' + \frac{1-\epsilon'}{n-i}\right) \cdot \left(1 - \sum_{j=0}^{j<i} P(f_j)\right)$$

avec n le nombre de fils de la racine, $i \in \{0, \ldots, n-1\}$, f_i le $i^{ème}$ meilleur fils de la racine, $P(f_i)$ la probabilité de jouer l'action menant au fils f_i et ϵ' le paramètre d'exploitation ($\epsilon' = 1 - \epsilon$).

Nous proposons en plus d'utiliser la règle suivante lors de la sélection de l'action à jouer, pour réduire la durée des parties et donc *a priori* la durée de l'apprentissage : toujours jouer une action menant à un état résolu gagnant s'il existe et ne jamais jouer une action menant à un état résolu perdant si cela est possible. Ainsi, si parmi les actions disponibles on sait qu'une des actions est gagnante, on la joue. S'il n'y en a pas, on joue aléatoirement suivant la loi choisie parmi les actions ne menant pas à un état résolu perdant (si les états ne sont pas tous perdants).

3.6 UBFM_s : jouer la sécurité

Nous proposons maintenant une légère modification d'UBFM, notée UBFM_s, qui vise à fournir un jeu plus sûr. L'action qu'UBFM choisit de jouer est celle qui mène à l'état de meilleure valeur. Dans certains cas, la meilleure action (*a priori*) peut mener à un état qui n'a pas été *suffisamment* visité (comme une feuille non terminale). Choisir cette action est donc une décision risquée. Nous proposons, pour éviter ce problème, une décision différente qui vise à jouer l'action la plus sure, à l'instar du MCTS (max child selection [5]). Si aucune action ne mène à un état résolu gagnant, l'action choisie par UBFM_s est celle

Fonction `descente(`e`)`
- **if** `terminal(`e`)` **then**
  - retourner $f(e)$
- **else**
  - **if** $e \notin E$ **then**
    - $E \leftarrow E \cup \{e\}$
    - **foreach** $a \in$ `actions(`e`)` **do**
      - $v(e, a) \leftarrow f(a(e))$
  - $a_b \leftarrow$ `meilleure_action(`e`)`
  - $v(e, a_b) \leftarrow$ `descente(`$a_b(e)$`)`
  - $a_b \leftarrow$ `meilleure_action(`e`)`
  - retourner $v(e, a_b)$

Fonction `meilleure_action(`e`)`
- **if** `premier_joueur(`e`)` **then**
  - retourner $\underset{a \in \texttt{actions}(e)}{\arg\max} \; v(e, a)$
- **else**
  - retourner $\underset{a \in \texttt{actions}(e)}{\arg\min} \; v(e, a)$

Fonction `apprentissage_descente()`
- $E \leftarrow \{\}$
- $e \leftarrow$ état initial
- **while** $\neg$`terminal(`e`)` **do**
  - $t =$ `time()`
  - **while** `time()` $- t < \tau$ **do** `descente(`e`)`
  - $a_b \leftarrow$ `selection_action(`e`)`
  - $e \leftarrow a_b(e)$
- $D \leftarrow \{\}$
- **foreach** $e \in E$ **do**
  - $a_b \leftarrow$ `meilleure_action(`e`)`
  - $D \leftarrow D \cup \{(e, v(e, a_b))\}$
- adapter f par rapport à D

Algorithme 1 : Apprentissage d'une partie avec *descente* et *tree learning* ($a(e)$: état obtenu après avoir joué l'action a dans l'état e ; $v(e, a)$: valeur obtenue après avoir joué a dans e ; f est du point de vue du premier joueur ; E : clefs de la table de transposition ; τ : temps de réflexion par action ; selection_action : meilleure_action() ou une autre distribution d'actions).

qui a été le plus de fois sélectionnée (depuis l'état actuel du jeu) au cours de l'exploration de l'arbre. En cas d'égalité, UBFM_s départage en choisissant celle qui mène à l'état de meilleure valeur. Cette décision est plus sure du fait que le nombre de fois qu'une action est sélectionnée correspond au nombre de fois que cette action est plus intéressante que les autres (la recherche est effectuée en meilleur d'abord).

Exemple 4. Le joueur en cours a le choix entre deux actions a_1 et a_2. L'action a_1 mène à un état de valeur 5 et a été sélectionnée 7 fois (depuis l'état courant et depuis le début de la partie). L'action a_2 mène à un état de valeur 2 et a été sélectionnée 30 fois. UBFM choisit l'action a_1 alors que UBFM_s choisit l'action a_2.

4 Expériences

Nous décrivons maintenant les différentes expériences réalisées. Elles se basent sur les techniques présentées dans la section précédente. Nous commençons par décrire les détails de l'apprentissage. Ensuite, nous évaluons le résultat d'un apprentissage en temps long utilisant nos techniques contre la version 1.0 et la version 2.0 de Mohex. Enfin, nous comparons expérimentalement différents algorithmes pour jouer, tels que UBFM_s et la complétion.

4.1 Détails techniques

Après chaque partie, afin d'améliorer la variabilité des données (exactes) utilisées par l'apprentissage, les états terminaux e des données récoltées $D = \{(e, v)\}$ sont retirés de D. À la place, pour un état sur deux e (différent) restant dans D, une simulation aléatoire de fin de partie est effectuée et l'état terminal obtenu e_t et sa valeur $\mathrm{p_t}(e_t)$ sont ajoutés dans D. Les données de D sont ensuite augmentées, le jeu présentant une symétrie par rapport au plateau (rotation de 180°). L'ajout du symétrique de chaque état double ainsi le nombre de données : $D = \{(e, v)\} \cup \{(s_{180°}(e), v)\}$. La dernière étape avant la phase d'apprentissage est la répartition aléatoire des données de D en sous-ensembles disjoints D_i (de taille 1024). La méthode d'optimisation utilisée pour l'apprentissage est *Adam* [14]. Nous l'utilisons pour minimiser itérativement la valeur $\sum_{(e,v)\in D_i} (f(e) - v)^2$ pour chaque D_i. Adam est appliqué une fois par D_i (une mise à jour par D_i). Les données sont ensuite effacées et une nouvelle partie débute.
La fonction d'évaluation adaptative utilisée est un réseau de neurones convolutif [17] ayant trois couches de convolution suivi d'une couche cachée entièrement connectée. Pour chaque couche convolutive, le noyau est de taille 3×3 et le nombre de calque est 150. Le nombre de neurones de la couche entièrement connectée est 81. La marge de chaque couche est à zéro. Après chaque couche sauf la dernière, la fonction d'activation ReLU [10] est utilisée. La couche de sortie contient un neurone. Il n'y a pas de fonction d'activation pour la sortie. L'entrée du réseau de neurones est un plateau de jeu étendu d'une ligne en haut, en bas, à droite et à gauche (à la manière de [32, 1]). Plus précisément, chacune de ces lignes est remplie entièrement des pièces du joueur du bord où elle se trouve. Cette extension revient simplement à représenter explicitement les bords du plateau et leur appartenance.
Le taux d'exploitation de la distribution de sélection d'actions évolue linéairement au cours du temps : $\epsilon' = \frac{t}{T}$ avec t le temps écoulé depuis le début de l'apprentissage par renforcement et T la durée totale de cet apprentissage.
Lors de l'apprentissage (en jouant contre soi-même), les deux joueurs utilisent la même fonction d'évaluation et la même table de transposition, qui est vidée après chaque partie (les états résolus sont cependant conservés).

4.2 Résultats contre Mohex 1.0 et Mohex 2.0

Un long apprentissage contre soi-même pour les plateaux de taille 11 a été effectué avec l'algorithme descente utilisant l'heuristique de la profondeur, la complétion, le coefficiented tree learning ainsi que la distribution et la règle associée de la section 3.5. La fonction d'évaluation a été pré-initialisée en apprenant les valeurs d'états terminaux aléatoires (au nombre de 8.836.000). L'apprentissage par renforcement de la fonction d'évaluation a duré 58.674 parties, à raison d'une seconde de réflexion par action.
UBFM_s doté de la fonction d'évaluation générée par cet apprentissage atteint le score de 60% de victoire (84% en premier joueur ; 37% en second joueur) contre Mohex 1.0 à la suite de 300 parties en premier joueur et 300 autres parties en second joueur, avec une seconde de réflexion par action. Son score contre Mohex 2.0 est de 53% victoire (81% en premier joueur ; 25% en second joueur) à la suite de 1000 parties en premier joueur et 1000 autres parties en second joueur, avec une seconde et demi par action.

4.3 Comparaison d'algorithmes de jeu

Nous comparons maintenant les taux de victoire de différents algorithmes pour jouer en les évaluant contre Mohex 2.0. Chacun d'eux se base sur la fonction d'évaluation des plateaux de taille 11 générée au cours de l'expérience précédente. Nous comparons UBFM_s sans complétion et les algorithmes suivants avec complétion : UBFM, UBFM_s, alpha-bêta à profondeur 1, alpha-bêta à profondeur 2 ainsi qu'une modification d'UBFM_s, notée UBFM_s + UCT. Cette variante effectue une exploration en meilleur d'abord tenant compte du nombre de visites des nœuds à la manière de la version la plus populaire du MCTS, UCT [5]. Pour cela, le fils sélectionné est le meilleur selon la somme de la valeur d'UBFM_s (normalisée pour être dans $[0, 1]$) et du terme d'exploration suivant : $c \cdot \sqrt{\frac{\log n}{N}}$ avec n le nombre de visites du fils considéré, N le nombre de visites du nœud père et c une constante d'exploration (en théorie $\sqrt{2}$).
Les taux de victoire contre Mohex 2.0 sont décrits dans la table 2. C'est UBFM_s qui obtient le meilleur taux de victoire. L'utilisation de la complétion apporte un très léger gain de l'ordre de 2%. Jouer l'action la plus sure a apporté un gain de 6%. UBFM_s est supérieur de 24% à l'alpha-bêta à profondeur 1. Il est supérieur de 18% à l'alpha-bêta à profondeur 2, bien que ce dernier ait un temps de réflexion

Algorithmes	1er	2ème	moyenne
UBFM_s	81%	25%	53%
UBFM_s sans complétion	77%	25%	51%
UBFM	61%	34%	47%
Alpha-bêta à profondeur 2	56%	13%	35%
Alpha-bêta à profondeur 1	37%	20%	29%
UBFM_s + UCT ($c = 2 \cdot \sqrt{2}$)	55%	12%	33%
UBFM_s + UCT ($c = \sqrt{2}$)	55%	15%	35%
UBFM_s + UCT ($c = \frac{\sqrt{2}}{2}$)	50%	12%	31%
UBFM_s + UCT ($c = \frac{\sqrt{2}}{10}$)	56%	15%	36%

TABLE 2 – Statistiques de victoire contre Mohex 2.0 de différents algorithmes de recherche pour 600 parties en premier joueur et 600 autres parties en second joueurs. UBFM, UBFM_s, chaque UBFM_s + UCT et Mohex 2.0 ont une seconde et demi par action. Alpha-bêta à profondeur 2 prend en moyenne environ deux secondes par action.

par action $1/4$ de temps supérieur en moyenne. Enfin, ajouter de l'exploration à UBFM_s a diminué le taux de victoire d'au moins 17% (pour chaque valeur c choisie).
Cette expérience suggère qu'une fois que l'apprentissage par renforcement est terminé, il est plus avantageux d'utiliser tout son temps à l'exploitation, en réfléchissant en meilleur d'abord. Cette stratégie amène à se focaliser sur des zones spécifiques et profondes de l'arbre de jeux (pour les parties de l'espace d'états suffisamment explorés lors de l'apprentissage). Au moment de l'évaluation, en général, il ne faut plus explorer. Il faut encore moins faire une recherche exhaustive en largeur. C'est à l'inverse plutôt le moment de valider que la meilleure action *a priori* est bien la meilleure action. Notons qu'UBFM_s (ou UBFM) est malgré tout capable de faire une recherche plus ou moins en largeur lorsqu'il rencontre des états ayant une grande variabilité de valeurs d'une action à l'autre. Cela arrive dans les parties de l'espace d'états qu'il n'a pas suffisamment explorés au cours de l'apprentissage. En résumé, jouer en meilleur d'abord permet d'effectuer un élagage très important, éventuellement temporaire, permettant ainsi d'obtenir un coup d'avance sur son adversaire. Il est toutefois possible que pour des temps de réflexion beaucoup plus long, ajouter de l'exploration donne un avantage.

5 Conclusion

Nous avons proposé plusieurs nouvelles techniques pour l'apprentissage par renforcement de fonctions d'évaluation. La première technique consiste à apprendre les valeurs de l'arbre de jeu au lieu de se restreindre à la valeur de la racine. Pour limiter le biais induit sur l'apprentissage par l'ajout de ces valeurs, nous avons proposé d'associer un coefficient à chaque état, dépendant du nombre de visites. Nous avons également proposé de remplacer le gain classique d'un jeu ($+1/-1$) par l'heuristique de la profondeur. Cela permet de tenir compte de la durée de la partie afin de favoriser les victoires rapides et les défaites lentes. Nous suggérons l'utilisation de la complétion pour améliorer le comportement de certaines fonctions d'évaluation, comme celles basées sur l'heuristique de la profondeur. Enfin, nous avons proposé l'algorithme descente qui explore à la manière du minimax à profondeur non bornée. Contrairement à ce dernier, descente explore itérativement les séquences de meilleures actions *jusqu'aux états terminaux*. Son objectif est d'améliorer la qualité des données utilisées lors de l'apprentissage, tout en gardant les avantages du minimax à profondeur non bornée.
En outre, nous avons proposé une autre variante du minimax à profondeur non bornée, qui joue l'action la plus sure au lieu de jouer la meilleure action. Nos expériences suggèrent que cela améliore le niveau de jeux lors des confrontations. Nos expériences suggèrent également qu'une fonction d'évaluation apprise par renforcement donne un meilleur niveau de jeux lorsqu'elle est utilisée par un algorithme de recherche en meilleur d'abord.
Enfin, l'utilisation des techniques proposées a permis de concevoir, pour la première fois, un programme-joueur au jeu de Hex (taille 11) atteignant le niveau de Mohex 2.0 par un apprentissage par renforcement contre soi-même sans utilisation de connaissance (ni de techniques Monte Carlo). Une étude comparant la vitesse d'apprentissage en fonction des techniques utilisées est en cours. Les résultats préliminaires suggèrent d'une part que l'utilisation de l'heuristique de la profondeur améliore l'apprentissage. Ils suggèrent d'autre part que l'apprentissage avec descente est meilleur qu'avec UBFM.
Les perspectives de recherche incluent l'application de nos contributions au jeu de Go et au General Game Playing avec information parfaite dans un premier temps. Elles incluent également, par conséquent, l'adaptation de nos contributions aux contextes des informations cachées, des jeux stochastiques et des jeux multi-joueurs.

Remerciements

Je remercie le GREYC de m'avoir donné accès à son serveur de calcul, ce qui m'a permis d'effectuer l'expérience de la section 4.2.

Références

[1] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[2] Hendrik Baier and Mark HM Winands. Mcts-minimax hybrids with state evaluations. *Journal of Artificial Intelligence Research*, 62 :193–231, 2018.

[3] Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning Ways for Your Mathematical Plays, Volume 3*. AK Peters/CRC Press, 2018.

[4] Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. On the complexity of connection games. *Theoretical Computer Science*, 644 :2–28, 2016.

[5] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Transactions on Computational Intelligence and AI in games*, 4(1) :1–43, 2012.

[6] Tristan Cazenave. Residual networks for computer go. *Transactions on Games*, 10(1) :107–110, 2018.

[7] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pages 1766–1774, 2015.

[8] Chao Gao, Ryan B Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *Transactions on Games*, 2017.

[9] Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for monte carlo tree search. In *International Joint Conference on Artificial Intelligence*, pages 3762–3768, 2018.

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *The fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[11] Ryan Hayward and Noah Weninger. Hex 2017 : Mohex wins the 11× 11 and 13× 13 tournaments. *ICGA Journal*, 39(3-4) :222–227, 2017.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0 : a pattern-based mcts hex player. In *International Conference on Computers and Games*, pages 60–71. Springer, 2013.

[14] Diederik P Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014.

[15] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598) :671–680, 1983.

[16] Richard E Korf and David Maxwell Chickering. Best-first minimax search. *Artificial intelligence*, 84(1-2) :299–337, 1996.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] Michael Lederman Littman. *Algorithms for sequential decision making*. PhD thesis, 1996.

[19] Jacek Mandziuk. *Knowledge-free and learning-based methods in intelligent game playing*, volume 276. Springer, 2010.

[20] Joseph Mellor. *Decision Making Using Thompson Sampling*. PhD thesis, 2014.

[21] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540) :529, 2015.

[23] Jakub Pawlewicz and Ryan B Hayward. Scalable parallel dfpn search. In *International Conference on Computers and Games*, pages 138–150. Springer, 2013.

[24] Stefan Reisch. Hex ist pspace-vollständig. *Acta Informatica*, 15(2) :167–191, 1981.

[25] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587) :484, 2016.

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv :1712.01815*, 2017.

[27] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676) :354, 2017.

[28] Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning for creating evaluation function using convolutional neural network in hex. In *2017 Conference on Technologies and Applications of Artificial Intelligence*, pages 196–201. IEEE, 2017.

[29] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved : Now and in the future. *Artificial Intelligence*, 134(1-2) :277–311, 2002.

[30] H Jaap Van Den Herik and Mark HM Winands. Proof-number search and its variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

[31] Georgios N Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018.

[32] Kenny Young, Gautham Vasan, and Ryan Hayward. Neurohex : A deep q-learning hex agent. In *Computer Games*, pages 3–18. Springer, 2016.