Jeux

Bruno Bouzy[*] Tristan Cazenave[**] (coordinateur)
Vincent Corruble[***]
Olivier Teytaud[****]
[*] LIPADE, Université Paris Descartes
[**] LAMSADE, Université Paris-Dauphine
[***] LIP6, Université Pierre et Marie Curie
[****] TAO, Inria Saclay, Lri, UMR Cnrs 8623, Université Paris-Sud

22 janvier 2010

Les jeux sont étudiés en Intelligence Artificielle depuis ses origines. Historiquement le jeu d'Echecs et l'Alpha-Béta ont été les plus étudiés. Des algorithmes et des structures de données utilisées initialement pour les jeux comme l'approfondissement itératif et les tables de transpositions ont été ensuite réutilisés pour de nombreux autres problèmes.

Nous présentons dans ce chapitre différents algorithmes utilisés pour les jeux. Nous commençons par deux sections sur les jeux à deux joueurs, la première traite de l'Alpha-Beta et de certaines de ses optimisations, la deuxième porte sur les algorithmes de Monte-Carlo qui ont donné récemment de très bons résultats sur certains jeux et qui ont une portée très générale. On peut noter au passage les contributions essentiellement françaises aux algorithmes de Monte-Carlo, on peut même parler d'une école française du Monte-Carlo dans les jeux. Nous abordons ensuite les puzzles et l'analyse rétrograde. Nous concluons avec une section sur les jeux vidéo.

1 Minimax, Alpha-Beta et améliorations

Cette section présente Minimax, Alpha-Beta et ses optimisations. Alpha-Beta est un algorithme de recherche arborescente utilisé dans les jeux à deux joueurs [86], [2]. Il a été développé en même temps que la programmation des Échecs, domaine largement rempli et influencé par ce développement depuis 1950 [88] jusqu'en 1997 lorsque Deep Blue a battu le champion du monde humain Gary Kasparov [5], [15]. L'origine de Alpha-Beta est difficile à déterminer exactement, car il n'existe pas un article fondateur. En revanche, il existe de nombreux articles le décrivant [16], [55], [64]. Alpha-beta est une amélioration de Minimax [96] présenté dans la partie 1.1. La particularité de Alpha-beta par

rapport à Minimax d'élaguer certaines branches de l'arbre de recherche est expliquée dans la partie 1.2. En pratique, l'efficacité de Alpha-Beta dépend de nombreuses autres améliorations expliquées par les parties suivantes : les tables de transpositions (partie 1.3), Iterative Deepening (ID) (partie 1.4), la fenêtre de largeur minimale et MTD(f) (partie 1.5), ou bien d'autres encore (partie 1.6). Toutes les améliorations précédentes se placent dans le cadre d'une recherche à profondeur fixée à l'avance. D'autres algorithmes descendant de Minimax sont également très intéressants (partie 1.7). Enfin la détection de menaces permet de résoudre beaucoup plus rapidement certains jeux (partie 1.8).

1.1 Minimax

Minimax est un algorithme de recherche arborescente utilisé dans les jeux à deux joueurs à somme nulle avec alternance de coups amis et ennemis [96]. Il suppose l'existence d'une fonction d'évaluation appelée à une profondeur donnée et commune aux deux joueurs. Le joueur ami cherche à maximiser l'évaluation et le joueur ennemi cherche à la minimiser. A un noeud ami (respectivement ennemi), la valeur minimax est le maximum (respectivement minimum) des valeurs minimax des noeuds fils. Pour connaître la valeur minimax de la racine d'un arbre de profondeur d et de facteur de branchement b, l'algorithme minimax explore un nombre de noeuds approximativement égal à b^d pour b et d suffisamment grands.

1.2 Alpha-Beta

Alpha-Beta (AB) calcule la valeur minimax du noeud racine en parcourant moins de noeuds que ne le fait Minimax. Pour cela, AB utilise deux valeurs α et β, bornes d'un intervalle dans lequel est située la valeur AB du noeud considéré. Dans la suite, on appelle valeur AB, ou v, la valeur calculée par AB à un noeud de l'arbre. Le joueur ami (respectivement ennemi) cherche à obtenir une valeur v aussi grande (respectivement petite) que possible. Par définition, α est la valeur minimale v que le joueur ami est certain d'obtenir et que le joueur ennemi ne peut contester. β est la valeur maximale que le joueur ennemi est certain d'obtenir, et incontestable par le joueur ami. On a toujours $\alpha \leq v \leq \beta$ et $\alpha < \beta$. Au début de l'exécution de l'algorithme, le noeud courant est le noeud racine et on a $\alpha = -\infty$ et $\beta = +\infty$. La racine est supposée être un noeud ami. A un noeud ami, AB appelle AB récursivement en passant en paramètres α et β sur les noeuds fils, ceci dans un ordre donné généralement par les connaissances du domaine. Soit r la valeur retournée par AB sur un fils. Si $r \geq \beta$, AB s'arrête (par définition de β, il est impossible que r soit supérieur à β) et AB retourne β. Ce cas s'appelle une coupe β car les noeuds fils suivants ne sont pas explorés. Si $\beta > r > \alpha$, AB a amélioré ce qu'il pouvait atteindre donc AB met a jour α avec r. Si $r \leq \alpha$, AB continue en appelant AB sur les fils suivants. Quand tous les noeuds fils ont été explorés, AB retourne α. A un noeud ennemi, AB fait un traitement analogue : si $r \leq \alpha$, AB s'arrête et retourne α. Il s'agit d'une coupe α. Si $\alpha < r < \beta$, l'ennemi a amélioré ce qu'il pouvait atteindre jusque là, donc

AB met à jour β avec r. Si $r \geq \beta$, AB continue. Quand tous les noeuds fils ont été explorés, AB retourne β.

Si Alpha-Beta est lancé avec des valeurs initiales α et β, et si AB retourne une valeur r, alors AB garantit les résultats suivants : si $\alpha < r < \beta$ alors r est égale à la valeur minimax, si $\alpha = r$ alors la valeur minimax est inférieure ou égale à α, si $r = \beta$ alors la valeur minimax est supérieure ou égale à β.

Alpha-Beta utilise une mémoire linéaire en fonction de la profondeur du noeud courant.

Alpha-Beta est très sensible à l'ordre dans lequel il explore les noeuds. Cet ordre est en général donné par des connaissances du domaine ou par des heuristiques de recherche. Le fait d'explorer le meilleur noeud en premier permet d'augmenter α, ce qui produit des coupes et diminue le nombre de noeuds explorés. [55] montre que pour connaître la valeur minimax de la racine d'un arbre de profondeur d et de facteur de branchement b, AB explore un nombre de noeuds au moins égal à approximativement $2b^{d/2}$ et que cette borne inférieure peut être atteinte si l'ordre d'exploration des coups est bon. Cela signifie grossièrement que dans les bons cas AB explore approximativement $2\sqrt{T}$ noeuds où T est le nombre de noeuds explorés par minimax.

En pratique, Alpha-Beta est utilisée dans sa version NegaMax. Contrairement à AB qui différencie le cas des noeuds amis et des noeuds ennemis, NegaMax gère un seul cas et ne voit qu'un seul type de noeud. Pour cela, pour chaque noeud fils, NegaMax appelle récursivement -NegaMax avec les paramètres $-\beta$ et $-\alpha$.

1.3 Tables de transposition (TT)

En pratique, on fait tourner AB avec une table de transpositions (TT). La TT est une table de toutes les positions déjà rencontréee jusque là. Elle s'appelle table de transpositions car elle n'a d'intérêt que lorsque deux séquences de coups de la recherche aboutissent à la même position, ce qui s'appelle une transposition. Une transposition peut arriver dés qu'une séquence contient deux coups du même joueur. Par exemple, au Go, les séquences de coups A, B, C et C, B, A aboutissent à la même position si aucune capture n'arrive, ce qui est le cas général. En première approximation, quand AB est appelé sur un noeud, il regarde d'abord si la position courante est stockée dans la table. Si oui il utilise les informations disponibles et évite de refaire la recherche située sous cette position. Sinon, il fait la recherche. Quand AB a fini l'exploration d'un noeud, il écrit les résultats de la recherche issue de cette position dans la table. Cette amélioration est la première à apporter à AB.

Pour représenter une position d'un jeu de manière utilisable avec une table, une première idée est de l'associer à un nombre compris entre 0 et $|E|$, la taille de l'espace des états E du jeu. $|E|$ est potentiellement très grand, disons 2^G. (Au Go 9x9, on a approximativement $G = 133$, car $|E| \simeq 3^{81} \simeq 10^{40} \simeq 2^{133}$). Cette première idée n'est donc pas possible telle quelle sur les ordinateurs actuels et il faut représenter une position par un nombre significativement plus petit, au risque d'avoir des collisions (appelées collisions de type 1). Zobrist a inventé un

codage permettant de représenter une position avec un nombre entier utilisable avec un ordinateur actuel (un nombre sur 32 ou 64 bits), avec une probabilité de collision arbitrairement faible. Pour cela, à chaque valeur de propriété de la position est associée à un nombre aléatoire, fixé hors ligne. (Par exemple, au Go, pour la couleur blanche de l'intersection numéro 4 du damier correspond un nombre aléatoire fixé, pour la couleur noire de l'intersection numéro 4 correspond un autre nombre aléatoire fixé, et pour la couleur vide de cette intersection correspond le nombre 0, et ainsi de suite pour toutes les intersections du damier). Le nombre de Zobrist d'une position est alors le XOR des nombres aléatoires associés aux valeurs des propriétés de la position. (Par exemple, au Go 9x9, le nombre de zobrist d'une position est le XOR de tous les nombres de Zobrist correspondant aux 81 intersections du damier). Quand un coup est joué, le nombre de Zobrist de la nouvelle position est mis a jour incrémentalement en fonction des changements de valeurs des propriétés de la position (Au Go, si une pierre blanche est posée sur une intersection i, le nombre de Zobrist de la nouvelle position est le XOR de l'ancienne position avec le nombre de Zobrist correspondant à la couleur blanche de l'intersection i). Zobrist montre que la probabilité de collision de type 1 entre deux positions peut être rendue arbitrairement faible [97]. Ce mécanisme s'appelle le hâchage de Zobrist.

En pratique, la taille de la TT est limitée, disons égale à 2^L (avec $L = 20$ ou $L = 30$). On prend alors les L premiers bits du nombre de Zobrist pour obtenir l'index d'un enregistrement dans la table. Des collisions peuvent alors se produire si deux positions différentes ont des index identiques (collisions de type 2). Pour éviter une collision de type 2 et l'utilisation des informations sur un enregistrement ne correspondant pas à la position courante, la première chose faite par l'algorithme de recherche est de vérifier que le nombre de Zobrist de la position courante est égal à celui de l'enregistrement trouvé avec l'index. Si c'est le cas, l'algorithme peut utiliser l'information contenue dans l'enregistrement, sinon il fait comme si l'enregistrement n'existait pas et explore la position.

Les premiers programmes d'Échecs [47] utilisaient déjà les TT. Le hâchage de Zobrist est un mécanisme très général, utilisé dans beaucoup de jeux utilisant une recherche arborescente.

1.4 Iterative Deepening

AB est un algorithme en profondeur d'abord. Si la solution optimale est courte et située en dessous du second noeud de la racine, Alpha-beta explore d'abord tous les noeuds situés sous le premier noeud. Il peut alors dépenser un temps inutile à explorer les profondeurs sous ce premier noeud.

Iterative Deepening (ID) est un algorithme itératif appelant AB à profondeur 1, puis 2, etc, tant que du temps de réflexion existe [57]. Le premier avantage de ID est d'être "anytime" (tant que du temps reste, ID explore la profondeur suivante, si le temps est écoulé, ID retourne le coup calculé à l'itération précédente). Un deuxième avantage est de trouver la solution optimale la plus courte. Un troisième avantage est son couplage avec les TT. A une itération donnée et sur une position donnée, ID stocke le meilleur coup trouvé. Aux itérations suiv-

antes, ID explore le meilleur coup d'une position d'abord, ce qui produit des coupes et rend AB efficace. Ce mécanisme a été utilisé dans les premiers programmes d'Échecs [89].

1.5 MTD(f)

Au lieu de lancer AB avec $\alpha = -\infty$ et $\beta = +\infty$, on peut lancer AB avec des valeurs quelconques pourvu que $\alpha < \beta$. Soit v la valeur AB théorique de la racine. Si α est tel que $v \leq \alpha$, alors AB retournera α et réciproquement. De même, si β est tel que $v \geq \beta$, alors AB retourne β et réciproquement. Enfin, si α et β sont tels que $\alpha < v < \beta$, alors AB retourne v et réciproquement.

L'idée des fenêtres à largeur minimale est de poser $\beta = \alpha + 1$. AB va produire beaucoup de coupes, et son exécution sera très peu coûteuse comparée à une exécution effectuée avec $\alpha = -\infty$ et $\beta = +\infty$. Le résultat d'une exécution de AB sera soit une borne inférieure sur v : $v \geq \alpha + 1$ si AB retourne $\alpha + 1$, soit une borne supérieure sur v : $v \leq \alpha$ si AB retourne α.

La classe d'algorithmes MTD [75] repose sur un mécanisme itéré des fenêtres à largeur minimale. (MTD signifie Memory Test Driver. "Memory" car pour être efficace cet algorithme doit utiliser les TT. "Test" car un appel de AB avec $\beta = \alpha + 1$ permet de tester si v est plus grand ou plus petit qu'une certaine valeur. "Driver" car il s'agit d'une classe d'algorithmes pilotant de manières différentes les appels itérés de AB.).

MTD(f) est l'instance la plus simple et la plus utilisée.

MTD(f) appelle itérativement AB avec $\alpha = \gamma$ et $\beta = \gamma + 1$. γ est initialisé avec une valeur quelconque (ou bien donnée par exemple par la précédente exécution de MTD(f)). A chaque itération, si la valeur de retour de AB est égale à γ, alors $v \leq \gamma$ et γ est décrémenté, sinon $v \geq \gamma + 1$ et γ est incrémenté. Après un nombre fini d'itérations, la borne inférieure et la borne supérieure de v sont égales, v est connue, MTD(f) s'arrête et le meilleur coup est lu dans la TT. A condition d'être couplée avec TT et ID, MTD(f) est une amélioration sensible de AB utilisée dans les programmes d'Échecs actuels.

1.6 Autres améliorations de Alpha-Beta

D'autres améliorations de AB existent. Principal Variation Search (PVS) est une recherche AB qui supppose que les noeuds sont déjà bien ordonnés par le générateur de coups et donc que la recherche est une vérification [71], [70]. Sur un noeud fils, PVS appelle PVS récursivement avec une fenêtre minimale pour vérifier que α n'est pas amélioré. Si c'est le cas, la recherche n'a pas coûté beaucoup, c'est l'intérêt de PVS. Sinon, on relance une recherche en mettant le β normal, ce qui coûte car c'est une seconde recherche. L'heuristique du coup nul [40] consiste à déterminer une première valeur de α en jouant un coup ne faisant rien, donc donnant le trait à l'adversaire, et en lançant une recherche à profondeur réduite (donc ayant un coût négligeable comparé à celui d'une recherche normale). Cette première valeur de α est significative et permet de lancer une recherche normale avec une bonne valeur initiale. L'heuristique de

l'histoire [84] consiste à enregistrer les coups produisant des coupes AB dans une table, et à les essayer en premier dans d'autres positions dans lesquelles ils sont possibles. Cela suppose qu'un coup puisse s'identifier et s'appliquer à des positions différentes. C'est le cas au Go où le lieu du coup est un bon identifiant, aux Échecs où la nature de la pièce, son origine et sa destination constituent aussi un identifiant utile.

Quiescence search [7] est une variante de AB consistant à n'engendrer que des coups urgents au sens du domaine considéré, et à effectuer la recherche jusqu'à une profondeur où il n'existe plus de coups urgents, et où la position est dite calme, donc évaluable de manière fiable. [80] est une étude sur la formule de back-up à partir des valeurs des noeuds fils dans la valeur du noeud parent. Enfin, [52] est un état de l'art des travaux sur Alpha-Beta.

1.7 Meilleurs en premier

D'autres algorithmes améliorent Minimax en explorant les noeuds suivant une stratégie du meilleur en premier. Proof Number Search (PNS) [3] compte le nombre de noeuds à explorer sous un noeud donné pour prouver sa valeur. PNS explore en premier les noeuds dont ce compteur est le plus faible. Best-First Search [58] appelle la fonction d'évaluation à tous les noeuds et explore le meilleur noeud en premier. SSS* [91] explore tous les noeuds en parallèle à la manière de A*. B* [8] suppose l'existence de deux évaluations, une évaluation optimiste et une évaluation pessimiste, et explore de manière à prouver que la valeur pessimiste du meilleur noeud fils est supérieure à la valeur optimiste du second noeud fils. [66], [85] définissent les noeuds conspirants comme étant les noeuds feuilles de l'arbre exploré dont l'évaluation influe sur la valeur minimax de la racine, et explorent ces noeuds en premier.

1.8 Algorithmes exploitant les menaces

Une menace est un coup pour un joueur qui menace de gagner s'il est suivi par un autre coup du même joueur. Par exemple aux Echecs, un échec est une menace, au Go-Moku aligner quatre pierres est une menace. Lorsqu'on détecte une menace, en analysant les conditions pour lesquelles la menace est vérifiée on peut répertorier le sous ensemble des coups possibles de l'adversaire qui permettent d'invalider la menace. Ils sont en général bien moins nombreux que tous les coups possibles et ce sont les seuls coups à envisager pour celui qui est menacé puisque tous les autres coups sont perdants. Le facteur de branchement est alors beaucoup plus petit ce qui permet de résoudre des problèmes beaucoup plus rapidement qu'en envisageant tous les coups possibles. L'utilisation de menaces pour sélectionner une petit nombre de coups à envisager a d'abord été faite sur des problèmes d'Echecs [73], puis des règles simples de menaces ont permis de résoudre le Go-Moku [1], et dans cette lignée des règles ont été automatiquement engendrées pour accélérer la résolution de problèmes de Go [17]. L'évolution de ces algorithmes utilisant des connaissances sur les menaces a été de remplacer les connaissances par des recherches détectant les menaces,

ce qui est plus simple à mettre en oeuvre et plus général [18, 19, 21, 95, 22]. Ces techniques ont par exemple permis de résoudre le Phutball 11x11 [20] et l'AtariGo 6x6 [9].

2 Recherche Monte-Carlo

Un changement majeur a eu lieu en matière de jeu de Go ces dernières années. En 1998, Martin Mueller (6ème Dan amateur au Canada) donnait le nombre astronomique de 29 pierres de handicap au programme "Many Faces of Go", au meilleur niveau de l'époque, et l'humain gagnait [67]. A l'inverse, en 2008, le programme MoGo, issu de l'école Française de Monte-Carlo Go, gagnait à 9 pierres de handicap contre Kim Myungwang, pourtant 8ème Dan pro. D'autres succès ont suivi, jusqu'à une victoire à handicap 6 contre un joueur professionnel.

Ces succès sont liés à des progrès algorithmiques majeurs. Les mêmes algorithmes ont eu de nombreuses applications à d'autres domaines : l'apprentissage actif [81], optimisation sur des grammaires [38], optimisation non-linéaire [82]. Par ailleurs, dans le même temps, des algorithmes proches se développaient en planification.

2.1 Evaluation Monte-Carlo

La première idée était l'utilisation du recuit simulé pour ordonner des listes de coups [13]. En effet, la technique la plus classique pour jouer à un jeu à information parfaite est la technique dite "alpha-beta" ; elle marche par exemple très bien pour les dames ou les échecs, mais sa faiblesse est qu'elle nécessite une fonction d'évaluation, c'est-à-dire une fonction, relativement rapide, qui à une position associe une évaluation de sa valeur pour chacun des joueurs. Typiquement, aux échecs ou aux dames, un expert humain peut facilement écrire une fonction qui estime la probabilité pour noir de gagner la partie à partir d'une position donnée ; rien de tel n'existe au jeu de Go. [13] proposa pour pallier à ce problème une idée nouvelle : jouer un certain nombre de parties, aléatoirement, jusqu'à obtention d'une probabilité de gain approchée (cf Alg. 1) : le Monte-Carlo Go était né.

Quoique la technique de Monte-Carlo soit ancienne (on fait en général remonter son utilisation intensive au projet Manhattan, *i.e.* au projet conduisant à la construction de la bombe atomique aux Etats-Unis pendant la seconde guerre mondiale, mais elle avait auparavant été signalée comme moyen curieux de faire des calculs approchés de π), son utilisation dans le contexte des jeux est alors neuve.

2.2 Fouille d'Arbre Monte-Carlo

La technique s'avère crédible, elle est reprise [11, 12] et améliorée en la combinant avec des connaissances et des recherches [10, 28]. Néanmoins, le vrai

Algorithm 1 Evaluation d'une position p par la technique de Monte-Carlo via n simulations.

Entrée : une position p, un nombre n de simulations.
for $i =\in \{1, \ldots, n\}$ **do**
  Soit $p' = p$.
  while p' n'est pas un état final **do**
    $c =$coup aléatoire parmi les coups légaux en p'
    $p' = transition(p', c)$
  end while
  if p' est gagnant pour noir **then**
    $r_i = 1$
  else
    $r_i = 0$
  end if
end for
Sortie : $\frac{1}{n} \sum_{i=1}^{n} r_i$ (probabilité estimée de gain pour noir).

"décollage" des performances aura lieu lorsque la technique sera combinée à une construction incrémentale d'arbre. L'algorithme ainsi obtenu, appelé "Fouille d'Arbre Monte-Carlo" (Monte-Carlo Tree Search [33, 30]) est présenté en Alg. 2. La structure T est un ensemble de situations, augmenté peu à peu (on démarre avec une structure vide à laquelle on ajoute, à chaque simulation aléatoire, la première situation de cette simulation qui n'a pas encore été stockée) ; cette structure possède, pour chacun des noeuds qu'elle contient, un nombre de victoires pour noir et un nombre de victoires pour blanc.

Cette technique est actuellement à l'œuvre dans tous les programmes efficaces de jeu de Go, mais aussi en Havannah, Hex, Phantom Go, Amazons.

Il n'est pas précisé dans l'Algorithme 2 quelle est la formule dite de "bandit". Une variante classique, quoique pas la plus utilisée dans le cas du Go, est la formule dite UCT (upper confidence tree [56]) qui suit :

$$banditFormula(v, d, n) = v/(v + d) + \sqrt{K \log(n)/(v + d)} \qquad (1)$$

(où K est une constante choisie empiriquement, n est le nombre de simulation au noeud considéré, v le nombre de victoires pour le coup considéré et d le nombre de défaites). Le premier terme $v/(v + d)$, dit d'exploitation, favorise les coups qui ont un bon taux de succès ; le second terme favorise les coups encore peu explorés ($v + d$ est petit) et est ainsi appelé terme d'exploration. La formule 1 n'est pas proprement définie pour $v+d = 0$; il est fréquent de spécifier $banditFormula(v, d, n) = F$ lorsque $v + d = 0$, pour une certaine constante F [46].

Différentes modifications de cette formule ont été proposées. La formule dite RAVE (Rapid Action Value Estimates), basée sur les valeurs dites AMAF (All Moves As First), est comme suit :

$$banditFormula(v, d, v', d') = \alpha(v + d)v/d + (1 - \alpha)v'/d'.$$

Algorithm 2 Evaluation d'une position p par la technique de Fouille d'Arbre Monte-Carlo via n simulations. Notations : ¬blanc= noir, ¬noir=blanc ; $transition(p, c)$ est la situation où l'on se retrouve si le joueur au trait joue le coup c en position p.

Entrée : a position p, a number n of simulations.
$T \leftarrow$ structure vide.
for $i =\in \{1, \ldots, n\}$ **do**
  Soit $p' = p$, $q = \emptyset$, $partie = \emptyset$.
  while p' n'est pas un état final //Faire une partie complète **do**
    if p' est dans T // Si p' est en mémoire **then**
      $j =$ joueur au trait en p' // Algorithme dit "bandit"
      for $c =$coup légal en p' **do**
        //Calculer le score pour tous les coups
        $p'' = transition(p', c)$
        $Score(c) = banditFormula(T(\neg j, p''), T(j, p''), T(j, p') + T(\neg j, p'))$
      end for
      $c =$coup parmi les coups légaux en p' maximisant $Score(c)$
    else
      if $q = \emptyset$ // On n'a pas encore trouvé l'état à rajouter **then**
        if T ne contient pas q **then**
          $q = p'$
        end if
        partie$\leftarrow$ partie + p'
      end if
      $c =$coup aléatoire parmi les coups légaux en p'
    end if
    $p' = transition(p', c)$
  end while
  Ajouter q dans T // si $q \neq \emptyset$
  if p' est gagnant pour noir **then**
    $r_i = 1$
  else
    $r_i = 0$
  end if
  for p' in partie **do**
    $T(r_i, p') = T(r_i, p') + 1$ // Incrémenter $T(r_i, p')$
  end for
end for
Sortie : $\frac{1}{n} \sum_{i=1}^{n} r_i$.

On utilise les mêmes critères v et d, et on utilise en outre :

- v' le nombre de victoires où le coup considéré c est joué par le joueur au trait avant d'être joué par son adversaire [1] ;
- d' le nombre de défaites où le coup considéré c est joué par le joueur au trait avant d'être joué par son adversaire.

$\alpha(.)$ est une fonction tendant vers 1, par exemple $\alpha(n) = n/(n+50)$:

- v' et d' étant beaucoup plus grands que 1, on les utilise avec confiance et faute de mieux lorsque v et d sont trop petits pour que le ratio $v/(v+d)$ ait un sens statistique ;
- puis, le nombre de simulations augmentant, on migre vers $v/(v+d)$ qui est plus fiable asymptotiquement que $v'/(v'+d')$.

Une deuxième modification importante [34, 62, 29] est l'utilisation d'heuristiques calées sur des bases de données ; une méthode simple pour utiliser une heuristique $h(p', c)$ (traduisant typiquement la fréquence avec laquelle le coup c est jouée en situation p', au vu de la configuration de la situation p' autour du coup c est :

$$banditFormula(v, d, p', c) = v/(v+d) + Kh(p', c)/(v+d)$$

pour une certaine constante K. Les meilleurs résultats s'obtiennent en combinant ces différentes approches [62].

Les avantages avancés de la méthode FAMC sont les suivants :

- passage à l'échelle : plus on augmente la puissance de calcul et plus le niveau monte ;
- très faible expertise humaine ; l'algorithme présenté en section 2 est indépendant du jeu (même l'heuristique $h(.,.)$ peut être calibrée automatiquement sur des bases de données, quoiqu'il soit bien établi que régler empiriquement les constantes correspondant à des motifs connus des experts améliore nettement les résultats).

En raison du passage à l'échelle de la méthode, des parallélisations ont été proposées [25, 45] ; toutefois, les résultats, s'ils sont numériquement bons au sens où l'algorithme parallèle gagne avec très grande probabilité contre l'algorithme séquentiel, n'en sont pas moins limités : il semble que les performances contre les humains ne montent pas aussi vite que les performances contre le code séquentiel. En effet, la méthode Monte-Carlo est parfois biaisée, car le Monte-Carlo est incapable d'évaluer certaines positions comme les courses de liberté (dites "semeais") ; par exemple, tel semeai sera évalué comme gagnant à 50% pour noir, alors que pour un joueur humain il est clair qu'il est gagné à 100% pour noir. Augmenter la puissance de calcul n'empêchera pas l'algorithme de se tromper sur ce semeai, et la limite actuelle en matière de Go par Monte-Carlo semble liée à des limitations profondes de ce type.

[1] La victoire est comptée si le coup est joué par le joueur au trait pendant la simulation *avant* d'être éventuellement rejoué par son adversaire ; il n'est pas requis que le coup soit joué dès le premier coup à partir de la situation étudiée.

2.3 Pour aller plus loin

Une première direction essentielle de recherche est la réduction des biais, *i.e.* la modification du Monte-Carlo, si possible de manière indépendante du problème (pour gagner en généricité), de façon à ce que le taux de victoire moyen dans les simulations aléatoires soit représentatif de la situation ; il est connu que les semeais ou certains problème compliqués de vie et mort, en jeu de Go, donnent de gros biais et il y a fort à parier qu'il en sera de même dans d'autres applications. Dans le cas du jeu de Go, [46] a exhibé un Monte-Carlo modifié beaucoup plus efficace que le tirage aléatoire simple parmi les coups légaux. [62] a montré qu'on pouvait beaucoup gagner en (i) traitant dans le Monte-Carlo des cas particuliers de Nakade (ii) dosant mieux la part aléatoire. L'extension de (i) aux semeais est une sorte de Graal jamais atteint. Des mélanges du Monte-Carlo avec des solveurs tactiques ont été tentés mais ne sont pas encore pleinement opérationnels [28].

On peut noter les meilleurs programmes généraux de jeu utilisent aussi la fouille d'arbre Monte-Carlo [43]. Le principe de la programmation générale de jeux est que le programme reçoit les règles du jeu juste avant de jouer. Une compétition de programmes a lieu tous les ans à AAAI ou à l'IJCAI et a été gagnée en 2009 par Ary, un programme français. La recherche sur les programmes généraux de jeux remonte par ailleurs aux travaux de Jacques Pitrat [74].

Cette section a été entièrement dédiée aux jeux complètement observables ; l'extension au non-observable a été faite de différentes manières [23, 81] et reste un sujet de recherche.

Enfin, dans certains jeux (*e.g.* le poker), la modélisation de l'adversaire est centrale [65].

3 Puzzles

On appelle ici puzzles des problèmes à un joueur où l'on cherche une suite de coups permettant d'atteindre une solution du problème. Certains algorithmes peuvent optimiser le nombre de coups de la solution ou son score.

3.1 A*

L'algorithme A* [49] permet de trouver des solutions ayant un minimum de coups à des puzzles variés. Les puzzles résolus par A* sont par exemple le Rubik's Cube [60], le Taquin ou Sokoban [53]. A* est aussi utilisé pour les jeux vidéo [27, 14, 92].

Pour chaque puzzle auquel on applique A*, il faut définir une heuristique admissible qui sera calculée pour chaque position rencontrée. Une heuristique est admissible si elle donne toujours une valeur qui est plus petite que le nombre réel de coups qu'il faudra jouer pour atteindre une solution à partir de la position.

3.1.1 Heuristique de Manhattan

L'heuristique admissible la plus communément utilisée est l'heuristique de Manhattan. Le principe de l'heuristique de Manhattan est de calculer très rapidement la solution d'un problème simplifié et d'utiliser le coût de cette solution comme minorant du coût de la solution réelle. Elle consiste à considérer pour chaque pièce prise séparément qu'on pourra l'amener vers son emplacement cible sans encombres et sans considérer les interactions avec les autres pièces. On fait ensuite la somme de ces valeurs pour toutes les pièces. Par exemple au Taquin on compte pour chaque pièce le nombre de coups pour la déplacer vers son emplacement cible s'il n'y avait pas d'autres pièces. Pour le Rubik's cube on calcule la même chose pour chaque cube, toutefois comme chaque coup déplace huit cubes, il faut diviser la somme pour tous les cubes par huit. Concernant les déplacements sur une carte, on calcule l'heuristique de Manhattan en négligeant les obstacles.

3.1.2 Développement de l'arbre de recherche

Le principe de A* est de développer à chaque étape la position qui a le chemin estimé le plus petit. On estime le coût d'un chemin en ajoutant le coût du chemin pris pour arriver à cette position au coût minimum du chemin qu'il reste à parcourir, donné par l'heuristique admissible. Ceci assure que lorsqu'on trouve une solution elle est sur un chemin de coût minimal (toutes les autres positions qu'on pourrait encore développer sont sur un chemin de coût supérieur). Dans les jeux, le coût d'un chemin est souvent le nombre de coups joués sur le chemin.

3.2 Monte-Carlo

Le succès récent des méthodes de Monte-Carlo pour les jeux à deux joueurs a renouvelé leur intérêt pour les puzzles. L'utilisation d'une méthode de Monte-Carlo pour un puzzle est justifiée lorsqu'on ne dispose pas de bonnes heuristiques pour guider la recherche. C'est par exemple le cas pour des puzzles comme le Morpion Solitaire ou SameGame. Pour ces jeux, mais aussi pour le Sudoku ou le Kakuro, une méthode de Monte-Carlo imbriquée donne de très bons résultats [24]. Le principe de cette méthode est de jouer des parties aléatoires au plus bas niveau, et pour les niveaux supérieurs de choisir à chaque fois le coup qui a donné le meilleur score d'une partie du niveau juste inférieur (par exemple chaque coup d'une partie de niveau un est choisi d'après le score d'une partie aléatoire de niveau zéro qui commence par ce coup). Par ailleurs, les méthodes décrites dans la section précédente "Recherche Monte-Carlo" sont générales et applicables aux Puzzles.

3.3 Pour aller plus loin

Il est possible pour certains problèmes d'utiliser une version de A* en profondeur d'abord avec approfondissement itératif appelée IDA* [59] ce qui permet essentiellement d'utiliser A* avec très peu de mémoire.

Algorithm 3 Recherche d'une solution de coût minimal avec A*

```
Entrée : une position p.
Ouverts ← {p}.
Fermés ← {}.
g[p] = 0
h[p] = chemin estimé à partir de p
f[p] = g[p] + h[p]
while Ouverts ≠ {} do
  pos = position dans Ouverts qui a le plus petit f
  if pos est la position cherchée then
    retourner le chemin jusqu'à pos
  end if
  retirer pos de Ouverts
  ajouter pos à Fermés
  for c =coup légal de pos do
    pos' = transition(pos, c)
    g' = g[pos]+coût de c
    if pos' n'est pas dans Fermés then
      if pos' n'est pas déjà dans Ouverts avec g[pos'] ≤ g' then
        g[pos'] = g'
        h[pos'] =chemin estimé à partir de pos'
        f[pos'] = g[pos'] + h[pos']
        ajouter pos' dans Ouverts
      end if
    end if
  end for
end while
retourner échec
```

[54] est une bonne revue des puzzles NP-complet.

4 Analyse rétrograde

L'analyse rétrograde permet de calculer à l'avance une solution pour chaque élément d'un sous-ensemble des configurations d'un jeu. Elle a permis de résoudre optimalement plusieurs jeux. Nous présentons d'abord son application aux finales de jeux à deux joueurs, puis aux patterns.

4.1 Bases de données de finales

Le principe d'une base de données de finales est de calculer la valeur exacte de chaque position possible d'une finale. Par exemple aux Échecs on peut calculer pour chaque position qui contient cinq pièces ou moins sa valeur exacte, Ken Thompson a ainsi calculé les valeurs de positions contenant jusqu'à six pièces [94].

L'algorithme utilisé pour calculer les valeurs des positions est un algorithme d'analyse rétrograde. Son principe est de commencer par répertorier toutes les positions de Mat. Pour chaque position gagnée, il déjoue un coup blanc et un coup noir et vérifie en rejouant tous les coups possibles à profondeur deux le statut de la position déjouée. Il trouve ainsi de nouvelles positions gagnées. Il continue ce processus de découverte de positions gagnées tant qu'il en trouve de nouvelles.

Les bases de données de finales sont très utilisées aux Échecs et permettent de jouer certaines finales instantanément et mieux que n'importe quel joueurs humain. Elles ont même permis de découvrir des classes de positions gagnées que tous les joueurs croyaient nulles avant leur calcul par Ken Thompson (en particulier la finale Roi-Fou-Fou-Roi-Cavalier).

L'analyse rétrograde ne s'applique pas qu'aux Échecs. Chinook, le programme de dames anglaises (Checkers) qui a résolu ce jeu [87], fait aussi un usage intensif des bases de données de finales. Un autre jeu populaire qui a été complètement résolu à l'aide de l'analyse rétrograde est l'Awari [83]. Suite à l'analyse rétrograde de l'Awari, il existe maintenant un programme capable de jouer parfaitement toutes les positions d'Awari instantanément.

4.2 Bases de données de patterns

Lorsqu'on cherche à améliorer A* ou IDA* pour un problème, il est naturel de chercher à améliorer l'heuristique admissible. Améliorer l'heuristique admissible consiste à lui faire trouver des valeurs plus grandes pour un temps de calcul à peu près équivalent. En effet si l'heuristique donne des valeurs plus grandes tout en restant admissible, elle permettra à A* de couper les branches qui ne sont pas sur le plus court chemin plus tôt et donc de trouver une solution plus rapidement.

Une façon efficace d'améliorer l'heuristique admissible est d'effectuer des précalculs de nombreuses configurations de sous-problèmes plus simples que le problème original et d'utiliser les valeurs trouvées par ces précalculs comme heuristique admissible. Si l'on prend l'exemple du Taquin, l'heuristique de Manhattan consiste à considérer chaque pièce comme indépendante des autres. Toutefois si on prend en compte les interactions entre certaines pièces, l'heuristique sera améliorée. On calcule donc pour chaque configuration possible d'un sous ensemble des pièces le nombre de coups qu'il faudra pour toutes les mettre à leur place en prenant en compte les interactions. Par exemple pour le Taquin 4x4, on peut calculer toutes les configurations possibles des huit premières pièces en ignorant les autres pièces et en mettant donc des vides dans les emplacements non occupés par une des huit premières pièces [37]. On calcule alors avec un algorithme d'analyse rétrograde du même type que ceux utilisés pour les Échecs, les Checkers ou l'Awari le nombre de coups minimal pour ranger chaque configuration possible. Pour utiliser cette base de données de patterns (i.e. de configurations) on repère pour chaque position la configuration des huit premières pièces et on renvoie la valeur précalculée stockée à l'indice correspondant à la configuration, ce qui est très rapide.

L'utilisation de bases de données de patterns n'est pas limitée au Taquin. On peut aussi par exemple calculer le nombre de coups nécessaires pour ranger toutes les configurations des huit cubes de coin au Rubik's cube et s'en servir comme heuristique admissible [60]. Que ce soit au Taquin, au Rubik's cube ou pour d'autre problèmes, les base de données de patterns permettent des gains de rapidité très importants par rapport à l'heuristique de Manhattan (mille fois plus vite pour le Taquin 4x4 et nécessaires pour résoudre optimalement le Rubik's cube).

Pour le Rubik's cube on peut aussi combiner deux bases de données de patterns, par exemple en calculant une base pour les huit cubes de coin et une base pour six des douze cubes de bord. On prend alors pour une position le maximum des deux valeurs trouvées sur la position à évaluer. En effet les coups bougent à la fois des cubes de coin et des cubes de bord, on ne peut donc pas faire la somme des deux heuristiques. Pour d'autres problèmes comme le Taquin ce n'est toutefois pas le cas : si on a deux bases pour deux ensembles disjoints de pièces, on peut additionner les deux valeurs tout en restant admissible puisqu'aucun coup de la première base n'est compté dans la deuxième base (les pièces d'une base ne sont pas dans l'autre base) [41].

Pour des problèmes comme les tours de Hanoi avec quatre tiges il peut aussi être intéressant de compresser les bases de données de pattern pour les faire tenir en mémoire. La compression consiste à ne stocker qu'une valeur pour un ensemble de configurations (on stocke le minimum des nombres de coups des éléments de l'ensemble) [42].

L'utilisation de précalculs pour améliorer l'heuristique admissible et accélérer la recherche n'est pas limitée aux puzzles. Ainsi pour le calcul de plus court chemin sur une carte de jeu vidéo il est possible de précalculer la distance d'un seul point à tous les autres points de la carte puis d'utiliser l'inégalité triangulaire sur les distances pour calculer une heuristique admissible pour n'importe quel

couple de points [27].

On peut aussi calculer des bases de données de patterns pour les jeux à deux joueurs. Au Go par exemple on peut précalculer toutes les formes vivantes contenues dans un rectangle de taille donnée [26] et accélérer ainsi la résolution de problèmes de vie et de mort.

5 Jeu Vidéo

A coté des travaux importants sur les jeux classiques, de nouveaux types de jeux ont ces trente dernières années eux-aussi fait appel au domaine de l'intelligence artificielle. Ces jeux qu'on appelle "jeux vidéo" ont commencé à se distinguer de leurs prédécesseurs en donnant généralement un rôle important à la visualisation comme mode d'interaction, et en privilégiant dans un premier temps le jeu d'action où les réflexes priment sur la réflexion. Bien que visant généralement le grand public, ils nécessitent de l'intelligence artificielle aussi bien pour servir d'aversaire artificiel au joueur humain (comme le fait l'IA des jeux classiques) que pour peupler leurs mondes virtuels d'acteurs ou personnages qui les rendent vivants, crédibles et engageants. Il existe donc une communauté active d'individus en majorité à la frontière de la recherche et de l'industrie s'intéressant à ce domaine, comme en témoigne une série d'ouvrages spécialisés régulièrement mise à jour [76, 77, 78]. Beaucoup de chercheurs en IA se tournent vers le jeu vidéo comme un domaine d'étude très riche [61]. Les jeux vidéo sont des plateformes d'expérimentation pour plusieurs approches de l'Intelligence Artificielle : la simulation y est facile, soit en mode IA contre IA soit IA contre humain, les utilisateurs étant souvent plus faciles à trouver que pour d'autres types d'expérimentations. Citons par exemple des travaux sur le jeu Tetris [93] et sur le problème de la Patrouille Multi-Agents [4]. Ces dernières années, certaines plateformes ouvertes, qui facilitent grandement l'entrée dans ce domaine, ont même été proposeés aussi bien dans le domaine des jeux d'action que des jeux de stratégies. Enfin nous verrons que des chercheurs venus vers ce domaine pour y tester ou comparer leurs techniques de prédilection y trouvent de nouvelles problématiques de recherche issues de l'industrie ; ces nouvelles problématiques sont susceptibles de renouveler le champ de l'Intelligence Artificielle.

5.1 Introduction

Nous allons voir que, alors que certaines catégories de jeux vidéo peuvent être vues, du point de vue de l'IA, comme des extensions des jeux classiques, d'autres renouvellent complètement la problématique. Dans une première catégorie de jeux, on placera tout d'abord les Jeux de Stratégie modernes qui, comme leurs aînés, mettent en scène un conflit ou une compétition entre deux ou plusieurs camps, représentant chacun une armée, civilisation, ou autre faction, dans un contexte historique, imaginaire voire fantastique. Des exemples bien connus sont Age of Empires (Microsoft), un classique grand public, la série

Total War (Creative Assembly) qui combine niveaux stratégiques et combats tactiques 3D temps réel à des époques antiques et moyen-ageuses, la série Civilisation de Sid Meier, ou Hearts of Iron (Paradox Interactive). Les innovations sont sensibles et vont au-delà de l'immersion visuelle et sonore. Outre le déplacement de pièces (unités combattantes, etc.) dans un environnement 2D ou 3D qui rappelle les jeux classiques et beaucoup de wargames, on peut être amené à gérer aussi une économie (collecte de ressources, production, budget,...), une diplomatie (négociations d'alliances,...) voire les priorités de la recherche et de l'innovation (technologies militaires, idées de civilisation, innovations politiques). Ces niveaux multiples de simulation, du plus tactique (déplacement d'unités sur une carte), au plus stratégique (priorités à long terme,...) et leurs interactions complexes, ajoutent de nouveaux niveaux de complexité où l'impact des choix à moyen ou long terme est difficilement prédictible. Le jeu se déroule de manière classique au tour par tour, ou plus souvent en "temps réel" ou à une vitesse choisie par le joueur en fonction de l'intensité du jeu. Du point de vue de la complexité, un élément essentiel qui distingue ces jeux des jeux plus classiques est le parallélisme : que ce soit pour un wargame ou pour un jeu de "grande stratégie", toutes les unités peuvent recevoir du joueur des instructions indépendantes à chaque instant ou tour de jeu. L'ensemble des décisions et actions envisageables devient donc problématique à énumérer, et encore plus à évaluer, car sa taille croît exponentiellement avec le nombre d'unités. Les approches présentées plus haut issues de l'intelligence artificielle des jeux classiques, à base d'exploration d'un arbre de jeu, deviennent inopérantes.

Le premier point, lié à la complexité des jeux modernes, explique en partie un phénomène qui pourrait paraître a priori surprenant : l'IA des jeux vidéo utilise peu les techniques issues des recherches en IA sur les jeux classiques. Pourtant, le jeu vidéo est l'un des rares domaines où l'industrie s'est en grande partie appropriée la notion et surtout le vocable d'Intelligence Artificielle pour en faire un argument commercial. On parle dans l'industrie de l'*IA d'un jeu* comme la partie du logiciel qui gère et produit les comportements automatisés de l'adversaire ou des Personnages Non-Joueurs qui peuplent l'environnement. La question de savoir si cette *IA* utilise des techniques issues du domaine de recherche de l'IA proprement dit y est vu, peut-être à juste titre, comme secondaire. Comme nous allons le voir dans la section 2, une majeure partie des Jeux Vidéos actuels font en effet appel à des techniques qui peuvent sembler assez basiques du point de vue de la recherche en IA, mais qui ont de nombreux avantages du point de vue du concepteur de jeux tout en permettant une certaine complexité. Nous allons aussi voir comment des travaux récents, issus du monde de la recherche autant que de l'industrie, font évoluer ce domaine en partant d'une IA scriptée et figée vers une IA adaptative.

Cependant, un rapide panorama des problématiques de l'IA se doit d'en brosser les autres aspects. Une dimension importante est la notion de Personnage Non-Joueur, ces créatures qui peuplent le monde d'un jeu, en particulier les Jeux d'Aventure et Jeux de Rôle, mais on peut généraliser cette notion aux Jeux de Sports, de simulation, et aux Jeux de Tir à la première personne (ou FPS pour First-Person Shooters), catégorie très populaire. On dépasse ici la

notion d'adversaire, mais on s'approche de la notion d'acteur (qui doit suivre les instructions du metteur en scène) voire d'un agent autonome qui doit agir, réagir, interagir de manière crédible et intéressante avec l'histoire et le joueur humain. Tous les travaux récents sur les agents autonomes intelligents trouvent un débouché, et un champ d'expérimentation, particulièrement séduisant dans cette problématique [32], mais on va voir plus bas qu'on est amené dans ce cadre à approfondir, redéfinir ou même dépasser certains objectifs des travaux classiques sur les agents rationnels.

5.2 IA dans l'industrie du jeu vidéo - scripts et évolutions

Les jeux vidéo modernes doivent être vus comme des oeuvres interactives qui s'inscrivent en partie dans une culture cinématographique où un auteur imagine une histoire et met en scène des personnages. Ils y ajoutent une dimension clé qui est l'interactivité : l'évolution de l'histoire est fortement impactée par les actions du joueur, qui vont l'orienter dans une direction ou une autre ; on parle alors de narration interactive [72, 68]. Cet héritage explique en partie la réticence des concepteurs des jeux les plus scénarisés vis à vis de l'idée même d'une Intelligence Artificielle conduisant à des agents autonomes : les PNJ y sont alors vus comme des acteurs avant tout, qui doivent suivre les indications du concepteur et orienter l'évolution de l'histoire dans une des directions voulues. Les jeux les plus scénarisés font parties de la grande catégorie des roller-coasters (montagnes russes), ces jeux qui sont conçus précisément pour faire vivre au joueur une suite d'émotions planifiées où s'enchaîne des pics d'intensité, des moments de détente puis de suspense, etc. Ces jeux roller-coasters s'opposent aux jeux sandbox (bacs à sable), où le joueur construit lui-même l'histoire à partir d'un monde/environnement au contenu et à l'interactivité riche.

Dans la première catégorie, les jeux scénarisés ont le plus souvent une approche scriptée de l'Intelligence Artificielle. Des comportements, ou séquences d'actions sont déclenchées quand certaines conditions sur l'état du jeu ou de l'avatar du joueur sont réunies. Les PNJ ainsi programmés produisent les comportements atttendus par le concepteur au moment où il les attend et répond donc en grande partie aux besoins du jeu vidéo. Néanmoins, cette approche a ses propres limitations. Le coût de développement est élevé car toutes les situations intéressantes doivent être prévues au moment de la conception ce qui suppose que les comportements des joueurs soient suffisament contraints... ce qui va à l'encontre du sentiment d'immersion recherché pour la plupart des jeux vidéo. D'autre part, cela tend à conduire vers un jeu figé. La même situation généralement reproduit les mêmes comportements. Ceci peut entraîner une forme de lassitude pour le joueur, mais une autre conséquence est que le joueur peut exploiter ces répétitions pour piéger le jeu : en prévoyant trop aisément la réaction de l''IA" à ses propres actions, il obtient un avantage qui nuit rapidement à l'intêret du jeu.

Du point de vue technique, différentes approches ont été utilisées pour modéliser les comportements des PNJ. On a longtemps utilisé de simples machines à états finis. Les limitations des machines à états finis en terme de richesse de

représentation ont conduit à l'utilisation plus récemment de machines à états finis hiérarchiques [48] qui permettent de définir des états et des transitions généralisés et donc de mettre en commun certains aspects d'états similaires. De même, les arbres de comportements (behaviour trees [44]) visent à factoriser un maximum d'informations communes à plusieurs états ; le "comportement" y devient l'entité clé au détriment de l'état. Ces dernières années, pour la mise en oeuvre de ces modèles, de véritables langages de scripts puissants (tel LUA[51], utilisé entre autres dans le célèbre jeu de rôle multi-joueurs en ligne World of Warcraft) ont été proposés en partie pour répondre aux besoins du jeu vidéo. Ils ont l'avantage d'offrir un bon compromis entre vitesse d'exécution et une programmation externe au moteur du jeu, qui permet de gérer l'IA dans les dernières étapes du cycle de développement, et de la raffiner après commercialisation voire même de proposer aux utilisateurs (les joueurs) d'y contribuer par eux-mêmes.

5.3 Voies de recherches : IA adaptative et planification

Pour pallier aux limitations des approches scriptées citées plus haut, des tentatives ont été faites ces dernières années pour combiner les avantages des techniques de scripting appréciées de l'industrie du jeu vidéo et des capacités d'adaptation telles que celles étudiées par des chercheurs en Intelligence Artificielle, par exemple dans la communauté de l'apprentissage automatique. Un exemple important est le Dynamic Scripting [90]. Dans sa version initiale, cette approche vise à apprendre, par l'expérience, à associer un score à chaque script prédéfini par le concepteur du jeu. Cet apprentissage permet ensuite de sélectionner les scripts les plus adaptés à une situation donnée. Les versions ultérieures du Dynamic Scripting ont ajouté un niveau d'apprentissage par renforcement qui permet de modifier les scripts eux-mêmes ce qui en fait une véritable méthode d'apprentissage qui garde cependant la possibilité d'utilisation en entrée de scripts pré-existants, proposés par le concepteur.

Dans un contexte majoritairement académique, des recherches proposent depuis quelques années de concevoir l'IA des jeux de manière assez radicalement différente en plaçant à son coeur des techniques d'apprentissage qui ont un rôle triple : éviter les comportements figés des approches très scriptées classiques, offrir une possibilité d'adaptation du jeu (de son IA) au joueur (puisque c'est en jouant que l'IA adaptative va développer sa stratégie), et d'un point de vue de développement, offrir une alternative à l'approche classique qui implique la programmation puis le débogage de nombreux scripts par plusieurs programmeurs (certains studios de jeux vidéo emploient plusieurs dizaines de programmeurs de scripts d'IA en phase de production). La résistance à l'utilisation de ce type d'approche par l'industrie a plusieurs raisons : les techniques d'apprentissage actuelles convergent lentement sur des problèmes complexes, la paramétrisation est ardue, les comportements induits sont difficilement prévisibles et il est difficle de garantir a priori que le résultat obtenu soit acceptable par le concepteur et finalement par le joueur. D'un autre coté, l'industrie reconnait que les méthodes adaptatives utilisant l'apprentissage sont certainement

une voie prometteuse, comme en témoigne la préface enthousiaste de [76] qui qualifie l'apprentissage de *Next Big Thing*, pour l'IA des jeux vidéo, avant d'y consacrer dix chapitres de son ouvrage.

De par leur nature, la plupart des jeux vidéo se prêtent bien à une modélisation dans laquelle un ou plusieurs agents (PNJ) interagissent avec leur environnement en jouant. Ils reçoivent le plus souvent une évaluation de leurs actions, par exemple grâce à l'évolution d'un score de jeu. Ils semblent donc adaptés à des approches à base d'apprentissage par renforcement. Ce sont cependant des domaines où les espaces d'états et d'actions se révèlent plus grands que pour la plupart des applications connues de ces techniques, et incitent donc à améliorer ces méthodes, par exemple par des approches de factorisation des Processus Décisionnels Markoviens sous-jacents [39] ou par une décomposition hierarchique de l'apprentissage et la construction de représentations adaptées [63], ce qui a conduit à des solutions viables pour des jeux aussi variés que les FPS ou des jeux de stratégie de type wargame historique respectivement. Dans ce dernier cas en particulier, la décomposition hierarchique de la prise de décision et de l'apprentissage, ainsi que l'adaptation automatique par abstraction des représentations est rendue nécessaire par le parallélisme qui fait d'un wargame une véritable simulation multi-agents [32].

A coté de ces approches utilisant l'apprentissage intensément, il ne faut pas négliger des approches qui utilisent la planification. Les travaux présentés dans [39] sont d'ailleurs un exemple intéressant combinant l'apprentissage par renforcement et l'utilisation d'un modèle lui-même appris par expérience. Des approches utilisant des techniques sophistiquées de planification, en particulier à base de réseaux de tâches hierarchiques (HTN), deviennent une direction de recherche importante [50] dont certains jeux commerciaux, en particulier des jeux de stratégie et wargames complexes tels que Total War (Creative Assembly), Airborne Assault (Panther Games), ou Armed Assault (Bohemia Interactive), s'inspirent grandement.

5.4 Nouvelles problématiques de recherche pour l'IA des Jeux Vidéo

Nous avons présenté dans les pages précédentes des problématiques d'intelligence artificielle pour le jeu vidéo comme un prolongement des recherches sur les jeux classiques. L'hypothèse sous-jacente y était que l'objectif est de créer une IA qui joue mieux, c'est à dire dont le niveau de performances approche, atteint, voire dépasse, celui des joueurs humains. Cet objectif, tellement évident qu'il est souvent tu par les chercheurs en intelligence artificielle, est sérieusement remis en question par le domaine du jeu vidéo. En effet, si pour certains types de jeux, qu'ils soient classiques (échecs ou plus récemment Go) ou "modernes" (jeux de stratégie), la difficulté première pour un concepteur d'IA est de proposer un défi de bon niveau au joueur humain, ce n'est pas le cas pour de nombreux jeux pour lesquels les capacités des machines les rendent d'ores et déjà aptes à dépasser le niveau de la plupart des joueurs. La question pour la recherche en IA se pose alors différemment : le but n'est plus d'atteindre le niveau humain,

il est de proposer un adversaire, ou un compagnon de jeu, avec lequel le joueur humain appréciera la confrontation. On touche là à des notions complexes liées à l'idée de plaisir de jouer et d'amusement qui sont au coeur du jeu mais que les sciences dures ont peu abordé jusqu'à maintenant.

Beaucoup de travaux, à l'intersection de l'intelligence artificielle, de la psychologie et des sciences sociales s'attèlent à la définition et la mesure du plaisir et de l'amusement du joueur. Des théories issues de l'esthétique et du cinéma sont invoquées d'un coté, des approches expérimentales qui observent l'activité du joueur et ses paramètres physiologiques (rythme cardiaque, etc.) pour évaluer son intêret sont appliquées de l'autre. Ceci peut guider la conception de l'IA pour que les comportements qu'elle produit induisent la satisfaction ou l'amusement du joueur. Un cas particulier de cette problématique qui fait l'objet de nombreux travaux ces dernières années est le problème de l'équilibrage du niveau de jeu. Comment faire pour que l'IA joue au bon niveau quelque soit l'adversaire et ses évolutions dans le temps? C'est souvent en s'inspirant de la théorie psychologique du *flow* [35, 36] qui associe le sentiment de bien-être à un bon équilibre entre niveau de compétence et difficulté de la tâche à accomplir que le jeu vidéo traite de cette question [31]. En particulier, [6] s'intéresse à l'équilibrage dynamique du niveau de jeu, en proposant une méthode détournant l'usage classique de l'apprentissage par renforcement pour que l'action sélectionnée ne soit plus nécessairement celle qui a la meilleure valeur. Tout en apprenant à mieux jouer (en améliorant ainsi leur *compétence*), les agents apprennent aussi à adapter leur niveau de jeu, leur *performance*, en sélectionnant quand nécessaire des actions qu'ils jugent sub-optimales de leur point de vue, parce qu'ils estiment qu'elles seront plus adaptées au niveau de leur adversaire, le joueur humain.

Les travaux sur le niveau de jeu sont facilités par l'existence de mesures objectives, comme le score de jeu. D'autres aspects des jeux quelquefois plus difficilement mesurables ont un impact très fort sur la perception du joueur et à son immersion dans l'histoire. La crédibilité des Personnages Non Joueurs en est un bon exemple, particulièrement important dans les jeux d'aventure et les jeux de rôles qui supposent en général des interactions complexes (dialogues, négociations, ...) entre avatar du joueur et PNJ. Tout un champ de recherche se développe actuellement autour de cette notion assez subtile. En visant la crédibilité plus que le réalisme, on se place dans un cadre ludique plus que de simulation ; on s'autorise à travailler dans des mondes imaginaires où les règles du monde réel ne s'appliquent pas toutes mais où on recherche toujours une forme de cohérence interne qui contribue à ce que le joueur reste "pris" dans l'histoire. Dans ce cadre, il faut parfois dépasser le but d'avoir des PNJ aux comportements performants et rationnels. Pour qu'ils soient crédibles, il faut plutôt qu'ils aient une personnalité reconnaissable qui influence leurs actions sur le long terme, et qu'ils réagissent émotionnellement de manière crédible aux évènements du monde et aux interactions avec les autres personnages. Les PNJ des jeux deviennent donc un champ important pour les travaux sur l'*affective computing* [79]. [69]par exemple propose un modèle computationnel permettant de simuler la dynamique émotionnelle et sociale des PNJ en tenant compte de

leur personnalité, dans un contexte donné.

Les quelques problématiques introduites ci-dessus donnent une idée de la richesse des recherches sur l'IA des jeux vidéo. Cette liste est cependant loin d'être exhaustive. En se déplaçant du simple rôle d'adversaire à celui de PNJ, l'IA a étendu son champ d'action, mais elle se voit déjà sollicitée pour d'autres aspects du jeu vidéo. Peut-elle contribuer à la conception des Jeux ? à la mise en scène, par exemple pour le choix du positionnement automatique de la caméra de façon à donner la vue la plus intéressante de l'action au joueur, ou pour adapter ou composer la musique en fonction de l'état du jeu et du joueur ? Tout ces défis constituent des nouvelles frontières aussi bien pour la recherche en IA que pour le jeu vidéo.

6 Conclusion

Nous avons présenté dans ce chapitre les algorithmes classiques en programmation des jeux : l'Alpha-Beta et ses améliorations pour les jeux à deux joueurs et à somme nulle, A* pour les puzzles. Nous avons aussi présenté des approches plus récentes et en plein développement comme les algorithmes Monte-Carlo et les applications de l'IA aux jeux vidéo.

Comme nous l'avons vu, l'Intelligence Artificielle dans les jeux recouvre de nombreux algorithmes et soulève des problèmes variés qui ne sont pas nécéssairement spécifiques aux jeux. Les thèmes de recherche les plus actifs ces dernières années sont liés aux méthodes de Monte-Carlo et aux jeux vidéo.

Références

[1] L. Victor Allis, H. Jaap van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12 :7–23, 1996.

[2] L.V. Allis. *Searching for solutions in games and Artificial Intelligence*. PhD thesis, Vrije Universitat Amsterdam, 1994.

[3] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66 :91–124, 1994.

[4] Alessandro Almeida, Geber Ramalho, Hugo Santana, Patricia Azevedo Tedesco, Talita Menezes, Vincent Corruble, and Yann Chevaleyre. Recent advances on multi-agent patrolling. In *SBIA*, volume 3171 of *Lecture Notes in Computer Science*, pages 474–483. Springer, 2004.

[5] T. Anantharaman, M. Campbell, and F. Hsu. Singular extensions : adding selectivity to brute force searching. *Artificial Intelligence*, 43(1) :99–109, 1989.

[6] Gustavo Andrade, Geber Ramalho, Alex Sandro Gomes, and Vincent Corruble. Dynamic game balancing : An evaluation of user satisfaction. In *AAAI conference on Artificial Intelligence and Interactive Digital Entertainement*, pages 3–8, 2006.

[7] D. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43 :85–98, 1990.

[8] H. Berliner. The B* Tree Search Algorithm : a best-first proof procedure. *Artificial Intelligence*, 12 :23–40, 1979.

[9] F. Boissac and T. Cazenave. De nouvelles heuristiques de recherche appliquées à la résolution d'atarigo. In *Intelligence artificielle et jeux*, pages 127–141. Hermes, 2006.

[10] Bruno Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. *Inf. Sci.*, 175(4) :247–257, 2005.

[11] Bruno Bouzy and Tristan Cazenave. Computer Go : An AI oriented survey. *Artificial Intelligence*, 132(1) :39–103, 2001.

[12] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In *ACG*, volume 263 of *IFIP*, pages 159–174. Kluwer, 2003.

[13] B. Bruegmann. Monte carlo go. *Unpublished*, 1993.

[14] V. Bulitko, M. Lustrek, J. Schaeffer, Y. Bjornsson, and S. Sigmundarson. Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, 32(1) :419–452, 2008.

[15] M. Campbell, A.J. Hoane, and F-H Hsu. Deep Blue. *Artificial Intelligence*, 134 :57–83, 2002.

[16] M. Campbell and T. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20 :347–367, 1983.

[17] T. Cazenave. Metaprogramming Forced Moves. In *ECAI 1998*, pages 645–649, Brighton, UK, 1998.

[18] T. Cazenave. Iterative Widening. In *Proceedings of IJCAI-01, Vol. 1*, pages 523–528, Seattle, 2001.

[19] T. Cazenave. Abstract Proof Search. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2002.

[20] T. Cazenave. Gradual abstract proof search. *ICGA Journal*, 25(1) :3–15, 2002.

[21] T. Cazenave. A Generalized Threats Search Algorithm. In *Computers and Games 2002*, volume 2883 of *Lecture Notes in Computer Science*, pages 75–87, Edmonton, Canada, 2003. Springer.

[22] T. Cazenave. Generalized widening. In *ECAI 2004*, pages 156–160, Valencia, Spain, 2004. IOS Press.

[23] T. Cazenave. A Phantom-Go program. In *Advances in Computer Games 2005*, volume 4250 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2006.

[24] T. Cazenave. Nested Monte-Carlo search. In *IJCAI 2009*, pages 456–461, Pasadena, USA, July 2009.

[25] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101, 2007.

[26] Tristan Cazenave. Metarules to improve tactical go knowledge. *Inf. Sci.*, 154(3-4) :173–188, 2003.

[27] Tristan Cazenave. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *CIG*, pages 27–33, 2006.

[28] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and monte-carlo in the game of go. *IEEE CIG 2005*, pages 171–175, 2005.

[29] G. Chaslot, M. Winands, J. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3) :343–357, 2008.

[30] Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Strategies for Computer Go. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[31] J. Chen. Flow in games (and everything else). *Communications of the ACM*, 50(4) :34, 2007.

[32] Vincent Corruble and Geber Ramalho. *Jeux vidéo et Systèmes Multi-Agents*, pages 235–264. IC2 Series. Hermès Lavoisier, 2009. isbn : 978-2-7462-1785-0.

[33] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.

[34] Rémi Coulom. Computing "Elo ratings" of move patterns in the game of go. *ICGA Journal*, 4(30) :198–208, 2007.

[35] M. Csikszentmihalyi and I.S. Csikszentmihalyi. *Beyond boredom and anxiety*. Jossey-Bass San Francisco, 1975.

[36] M. Csikszentmihalyi and M. Csikzentmihaly. *Flow : The psychology of optimal experience*. Harper & Row New York, 1990.

[37] J. C. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, 4(14) :318–334, 1998.

[38] Frédéric De Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada, 2009.

[39] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Apprentissage par renforcement factorisé pour le comportement de personnages non joueurs . *Revue d'Intelligence Artificielle*, 23(2) :221–251, 4 2009.

[40] C. Donninger. Null move and deep search : selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3) :137–143, 1993.

[41] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)*, 22 :279–318, 2004.

[42] Ariel Felner, Richard E. Korf, Ram Meshulam, and Robert C. Holte. Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)*, 30 :213–247, 2007.

[43] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[44] G. Flórez-Puga, M. Gómez-Martın, B. Dıaz-Agudo, and P.A. González-Calero. Dynamic Expansion of Behaviour Trees. In *AAAI conference on Artificial Intelligence and Interactive Digital Entertainement*, pages 36–41, 2008.

[45] S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203, 2008. To appear.

[46] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Rapport de recherche INRIA RR-6062, 2006.

[47] R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt chess program. In *Fall Joint Computing Conference*, volume 31, pages 801–810, New york ACM, 1967.

[48] David Harel. Statecharts : A visual formalism for complex systems, 1987.

[49] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2) :100–107, 1968.

[50] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, 2005.

[51] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *HOPL III : Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[52] A. Junghanns. Are there practical alternatives to Alpha-Beta ? *ICCA Journal*, 21(1) :14–32, March 1998.

[53] A. Junghanns and J. Schaeffer. Sokoban : Enhancing general single-agent search methods using domain knowledge. *Artif. Intell.*, 129(1-2) :219–251, 2001.

[54] G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1) :13–34, 2008.

[55] D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6 :293–326, 1975.

[56] L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. In *ECML'06*, pages 282–293, 2006.

[57] R. Korf. Depth-first Iterative Deepening : an Optimal Admissible Tree Search. *Artificial Intelligence*, 27 :97–109, 1985.

[58] R. Korf and D. Chickering. Best-first search. *Artificial Intelligence*, 84 :299–337, 1994.

[59] R. E. Korf. Depth-first iterative-deepening : an optimal admissible tree search. *Artificial Intelligence*, 27(1) :97–109, 1985.

[60] R. E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI-97*, pages 700–705, 1997.

[61] John E. Laird. Research in human-level ai using computer games. *Commun. ACM*, 45(1) :32–35, 2002.

[62] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 2009 (accepted).

[63] Charles Madeira and Vincent Corruble. Strada : une approche adaptative pour les jeux de stratégie modernes. *Revue d'Intelligence Artificielle*, 23(2) :293–326, 2009.

[64] T. Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1) :3–19, 1986.

[65] Raphaël Maîtrepierre, Jérémie Mary, and Rémi Munos. Adaptative play in texas hold'em poker. In *European Conference on Artificial Intelligence - ECAI*, 2008.

[66] D. McAllester. Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, 35 :287–310, 1988.

[67] M. Müller. Computer go. *Artificial Intelligence*, 134(1-2) :145–179, 2002.

[68] S. Natkin. *Jeux vidéo et médias du XXIe siècle : quels modèles pour les nouveaux loisirs numériques ?* Vuibert, 2004.

[69] Magalie Ochs, Nicolas Sabouret, and Vincent Corruble. Simulation de la dynamique des emotions et des relations sociales de personnages virtuels. *Revue d'Intelligence Artificielle*, 23(2) :327–358, 2009.

[70] J. Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14 :113–138, 1980.

[71] J. Pearl. SCOUT : a simple game-searching algorithm with proven optimal properties. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 143–145, 1980.

[72] K. Perlin. Toward interactive narrative. In *International Conference on Virtual Storytelling*, pages 135–147. Springer, 2005.

[73] J. Pitrat. A program for learning to play chess. In C. H. Chen, editor, *Pattern Recognition and Artificial Intelligence*, pages 399–419. Academic Press, New York, 1976.

[74] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.

[75] A. Plaat, J. Schaeffer, W. Pils, and A. de Bruin. Best-first fixed depth minimax algorithms. *Artificial Intelligence*, 87 :255–293, November 1996.

[76] S. Rabin. *AI game programming wisdom.* Charles River Media, 2002.

[77] S. Rabin. *AI Game Programming Wisdom, Vol. 2, Charles River Media.* Charles River Media, 2003.

[78] S. Rabin. *AI Game Programming Wisdom 3 (Game Development Series).* Charles River Media, 2006.

[79] J. Rickel, S. Marsella, J. Gratch, R. Hill, D. Traum, and W. Swartout. Toward a new generation of virtual humans for interactive experiences. *IEEE Intelligent Systems*, pages 32–38, 2002.

[80] R. Rivest. Game-tree searching by min-max approximation. *Artificial Intelligence*, 34(1) :77–96, 1988.

[81] P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, 2009.

[82] Philippe Rolet, Michele Sebag, and Olivier Teytaud. Optimal robust expensive optimization is tractable. In *Gecco 2009*, page 8 pages, Montréal Canada, 2009. ACM.

[83] John W. Romein and Henri E. Bal. Solving awari with parallel retrograde analysis. *IEEE Computer*, 36(10) :26–33, 2003.

[84] J. Schaeffer. The history heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transcations on Pattern Analysis and Machine Intelligence*, 11(11) :1203–1212, November 1989.

[85] J. Schaeffer. Conspiracy Numbers. *Artificial Intelligence*, 43 :67–84, 1990.

[86] J. Schaeffer and J. van den Herik. Games, Computers, and Artificial Intelligence. *Artificial Intelligence*, 134 :1–7, 2002.

[87] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, July 2007.

[88] C.E. Shannon. Programming a computer to play Chess. *Philosoph. Magazine*, 41 :256–275, 1950.

[89] D.J. Slate and L.R. Atkin. Chess 4.5 - the northwestern university chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[90] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3) :217–248, 2006.

[91] G.C. Stockman. A minimax algorithm better than Alpha-Beta? *Artificial Intelligence*, 12 :179–196, 1979.

[92] Nathan R. Sturtevant, Ariel Felner, Max Barrer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *IJCAI*, pages 609–614, 2009.

[93] C. Thiery and B. Scherrer. Construction d'un joueur artificiel pour Tetris. *Revue d'Intelligence Artificielle*, 23(2-3) :387–407, 2009.

[94] K. Thompson. 6-piece endgames. *ICCA Journal*, 19(4) :215–226, 1996.

[95] T. Thomsen. Lambda-search in game trees - with application to Go. In T. Anthony Marsland and I. Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2002.

[96] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[97] A. Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2) :69–73, 1990.