Contrôle optimal stochastique et le jeu de Tetris

Christophe Thiery

Université Henri Poincaré – Nancy I

Contrôle optimal stochastique et le jeu de Tetris

MÉMOIRE

28 juin 2007

pour l'obtention du

Master Informatique de l'Université Henri Poincaré – Nancy I

(Spécialité Perception, Raisonnement et Interaction Multimodale)

par

Christophe Thiery

Composition du jury

Encadrant :	Bruno Scherrer
Professeurs :	Noëlle Carbonell
	Didier Galmiche
	Claude Godart
	Dominique Méry
Directeur de recherche :	Patrick Blackburn

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Avant-propos

J'ai effectué mon stage dans l'équipe Maia au Loria, du 26 février au 29 juin 2006. Le stage, intitulé "Contrôle optimal stochastique et le jeu de Tetris", intervient dans le cadre du Master Recherche spécialité PRIM (Perception, Raisonnement et Interaction Multimodale) et compte également dans la troisième année à l'Esial.

Je tiens à adresser mes remerciements à l'équipe Maia qui m'a accueilli, et à Bruno Scherrer qui m'a proposé ce stage et l'a encadré avec beaucoup d'attention.

Table des matières

Résumé **4**

1 Le problème de Tetris **5**
1.1 Pourquoi Tetris ? . . . 5
1.2 Implantation d'un simulateur . . . 6
1.2.1 Importance du temps d'exécution . . . 6
1.2.2 Règles simplifiées . . . 7
1.3 L'algorithme de Dellacherie . . . 8

2 Approche par Contrôle Optimal Stochastique **10**
2.1 Cadre . . . 10
2.1.1 Les Processus Décisionnels de Markov . . . 10
2.1.2 Politique et fonction de valeur d'une politique . . . 11
2.1.3 Politique optimale et fonction de valeur optimale . . . 12
2.2 Algorithmes . . . 13
2.2.1 Itération de la Valeur . . . 13
2.2.2 Itération de la Politique . . . 13
2.2.3 λ-Policy Iteration . . . 14
2.2.4 Version approximative de λ-Policy Iteration . . . 15
2.3 Modélisation de Tetris . . . 16
2.4 Expériences . . . 17
2.4.1 Itération de la Valeur . . . 17
2.4.2 λ-Policy Iteration . . . 18

3 Approche par Entropie Croisée **22**
3.1 Description de l'algorithme . . . 22
3.2 Application à Tetris . . . 23
3.3 Expériences . . . 24
3.3.1 Etude de l'influence des paramètres . . . 25
3.3.2 Notre meilleure heuristique : 40 millions de lignes . . . 27

Conclusion **29**

Bibliographie **31**

Annexes **31**

A Liste des critères **32**

B Expériences avec λ-Policy Iteration **38**

Résumé

Le jeu de Tetris est un problème complexe sur lequel s'affrontent de nombreuses techniques d'apprentissage automatique depuis une dizaine d'années. Le but du stage de recherche était d'étudier les algorithmes appliqués au jeu de Tetris, et en particulier ceux utilisant le contrôle optimal stochastique. Au cours de ce travail, nous avons étudié en particulier deux de ces approches, l'une utilisant le contrôle optimal stochastique (l'algorithme λ-Policy Iteration [2]) et l'autre utilisant la méthode d'entropie croisée [17]. Avec λ-Policy Iteration, nos résultats sont meilleurs qualitativement que dans l'expérience d'origine [2] et nous proposons une explication à cette différence : l'expérience d'origine comporterait une erreur d'implantation que nous pensons avoir identifiée. Nous nous sommes ensuite intéressés à la méthode d'entropie croisée, qui est un algorithme de recherche directe d'heuristique. Appliqué à Tetris [17], nos expériences confirment qu'il donne des résultats nettement plus élevés que les techniques faisant appel au contrôle optimal stochastique. Nous avons obtenu à l'aide de cette méthode une heuristique dont les performances dépassent d'un ordre de grandeur celles des meilleurs algorithmes à notre connaissance.

Chapitre 1

Le problème de Tetris

Dans ce chapitre, on introduit le problème de Tetris et on expose les caractéristiques qui font de lui un problème complexe et intéressant à étudier. On s'intéresse ensuite aux caractéristiques du simulateur développé au cours du stage, et on détaille enfin le fonctionnement de ce qui est, à notre connaissance, le meilleur algorithme actuel : celui de Pierre Dellacherie [7].

1.1 Pourquoi Tetris ?

Tetris est un célèbre jeu vidéo créé en 1985 par Alexey Pajitnov. Il s'agit d'un jeu de réflexion dans lequel le joueur doit empiler des pièces de différentes formes de manière à créer des lignes horizontales.

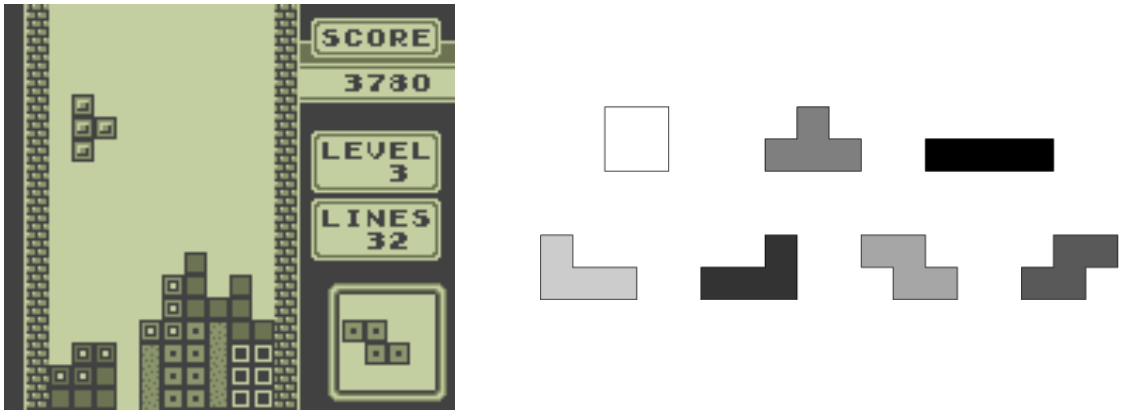


Fig. 1.1 – Le jeu de Tetris et les 7 pièces existantes

Les pièces tombent les unes après les autres depuis le haut de la zone de jeu. Le joueur peut faire pivoter la pièce courante et la déplacer horizontalement, afin de la placer comme il le souhaite sur les pièces déjà posées. Il existe 7 pièces différentes, représentées à la figure 1.1. Une fois la pièce posée, une nouvelle pièce arrive, choisie aléatoirement parmi les 7 pièces existantes (avec une loi de probabilité uniforme). A chaque fois qu'une ligne horizontale est pleine, celle-ci est supprimée. Tout ce qui est au-dessus descend d'une ligne et le joueur marque un point. La partie est perdue lorsqu'il n'y a plus de place en haut de la zone de jeu pour placer une nouvelle pièce. L'objectif est de réaliser le plus grand nombre de lignes possibles avant de perdre la partie. Pour plus d'informations sur le jeu de Tetris, le lecteur pourra se reporter à la page de Colin P. Fahey [9].

Le jeu de Tetris, aux règles pourtant simples, s'avère particulièrement difficile à résoudre pour un ordinateur. Il possède en effet les propriétés suivantes :

– **Grand espace d'états :** on peut considérer qu'un état à Tetris est constitué du mur (la configuration actuelle des cases) et de la pièce courante. Dans la version standard du jeu, il y a 200 cases (10 colonnes et 20 lignes) : chaque case pouvant être vide ou pleine, on a donc 2^{200} murs possibles[1], soit environ 10^{60}, à multiplier encore par le nombre de pièces possibles qui est de 7 (voir figure 1.1).
– **Un problème NP-complet :** il a été montré [8] que trouver la meilleure stratégie à Tetris est un problème NP-complet, même si la séquence des pièces qui arrive est connue à l'avance.
– **On est obligé de perdre :** il a été également prouvé [5] que toute partie de Tetris se termine nécessairement. Il existe (au moins) une séquence de pièces fatale, qui fera perdre la partie quelle que soit la stratégie employée. La probabilité qu'une telle séquence arrive lors d'une partie étant strictement positive, aucune partie ne peut être infinie. Cette propriété permet de comparer facilement les performances des algorithmes : en effet, comme tout algorithme finit par perdre, on peut l'évaluer en considérant le score moyen d'une partie.

Ces différentes propriétés font de Tetris un problème complexe et intéressant pour l'intelligence artificielle (voir l'annexe A qui mentionne les travaux dont nous avons connaissance).

1.2 Implantation d'un simulateur

Afin d'implanter les algorithmes existants et de mener des expériences, j'ai développé au cours du stage un simulateur de Tetris. Cette section décrit les caractéristiques de ce simulateur du point de vue du temps d'exécution et des variantes de Tetris qui ont été implantées.

1.2.1 Importance du temps d'exécution

Un des objectifs du stage étant d'implanter des algorithmes utilisant de l'apprentissage statistique, nous étions amenés à effectuer des simulations sur de nombreuses parties. Il faut noter en particulier que plus les algorithmes sont efficaces, plus les parties sont longues, et plus les temps d'exécution sont importants.

Pour reproduire les expériences de la littérature et potentiellement les améliorer, il était primordial que l'implantation soit la plus efficace possible. J'ai ainsi implanté un simulateur en langage C, où les calculs sont optimisés grâce à des opérations bit à bit. Avec une implantation classique, on stockerait naturellement l'état de chacune des 200 cases du jeu dans un tableau de 200 booléens. Dans mon simulateur, chaque ligne est stockée sur un entier. Les bits de cet entier représentent alors l'état de chaque case de la ligne : 1 pour une cellule pleine et 0 pour une cellule vide. Pour tester si une ligne est pleine, on compare l'entier qui la représente avec une valeur constante représentant une ligne pleine. De même, pour tester si une ligne comporte des trous, on effectue une opération bit à bit entre cette ligne et celle du dessus, au lieu de parcourir une par une les cases des deux lignes. Ainsi, on économise toutes les itérations sur les cases des lignes, ce qui permet de gagner beaucoup en temps d'exécution. J'ai également fait du *profiling* à l'aide de l'outil Valgrind (`http://www.valgrind.org`) pour identifier les parties critiques du code source et les optimiser en priorité.

En 2003, l'algorithme record de Pierre Dellacherie [7], présenté dans la section suivante, a mis deux jours pour jouer 36 parties et ainsi réaliser le score moyen de 650 000 lignes. Le programme qui le simulait [9] réalisait 160 lignes par seconde environ. En 2006, Szita et Lorincz ont exécuté une expérience avec un algorithme d'entropie croisée détaillé plus loin dans ce rapport [17]. L'exécution de cette expérience a duré au total un mois. J'ai implanté ces algorithmes sur mon simulateur. Sur une machine de bureau

[1] Notons que ce terme est légèrement surestimé, car certaines configurations sont impossibles (une ligne pleine, ou une ligne vide avec des cellules pleines au-dessus). Toutefois, ces murs impossibles ne changent pas l'ordre de grandeur qui est de 10^{60} murs.

classique, l'algorithme de Dellacherie réalise 10 000 lignes par seconde, et on peut reproduire l'expérience de Szita et Lörincz en moins d'une journée.

Le développement a été mené à l'aide d'une plate-forme de développement collaborative (SubVersion) sur le serveur Gforge de l'INRIA et sous le nom de projet "mdptetris". Le code source du projet est destiné à être rendu public afin que les personnes intéressées puissent reproduire nos simulations.

1.2.2 Règles simplifiées

Il est important de préciser que mon simulateur utilise des règles simplifiées. Ces règles ne correspondent par exactement au Tetris réel, dont Colin P. Fahey a proposé une spécification précise en 2003 ("Standard Tetris") [9]. Nos règles sont plus simples mais aussi plus proches des variantes utilisées par la plupart des algorithmes qui jouent à Tetris.

Pas de chute des pièces : Dans le vrai jeu de Tetris, la pièce courante tombe petit à petit du haut de l'écran et on a donc une limite de temps pour chaque coup. De plus, la vitesse des pièces augmente quand la partie dure, ce qui rend le jeu plus difficile pour un joueur humain.

Notre simulateur ne prend pas en compte la chute des pièces : il considère qu'il n'y a pas de limite de temps pour poser la pièce. En effet, les algorithmes prennent de toute façon une décision en un temps très court, et la pièce n'aurait pas le temps de commencer à descendre. Pour un joueur informatique sur notre simulateur, une action revient uniquement à choisir la rotation et la translation horizontale de la pièce courante. Par conséquent, contrairement au Tetris réel, on ne peut pas laisser volontairement descendre la pièce de quelques cases pour la translater ensuite, ce qui peut permettre de combler des trous[2].

Fin de la partie : Avec notre simulateur, la pièce courante n'apparaît dans la zone de jeu que lorsque le joueur (humain ou non) a décidé de la placer dans une certaine position (rotation + translation). La partie se termine si la pièce déborde du plateau une fois que le joueur l'a placée à l'emplacement choisi. Dans le Tetris réel, la pièce courante apparaît dans le haut de la zone de jeu dès qu'elle est générée, puis elle commence à descendre petit à petit. La fin de la partie est déclenchée si la pièce courante n'a pas assez de place pour apparaître. Notre gestion de la fin de la partie est donc moins contraignante que dans le Tetris standard. La majorité des algorithmes que nous avons étudiés gèrent la fin de la partie de la même manière que nous.

Taille du jeu : Notre simulateur est prévu pour fonctionner avec une taille variable du plateau de jeu. On peut ainsi simuler des parties sur une zone de jeu de taille standard (10×20) mais aussi sur des tailles plus petites pour que les parties durent moins longtemps ou pour étudier l'influence de la taille de la zone de jeu.

Prochaine pièce inconnue : Dans le jeu original, le joueur connaît la prochaine pièce qui va apparaître. Dans toute notre étude, on suppose que l'on connaît la pièce courante mais pas encore la suivante. Les algorithmes qui connaissent la pièce suivante à l'avance en plus de la pièce courante atteignent des performances largement supérieures [9, 4]. Tous les résultats obtenus en connaissant une seule pièce peuvent facilement s'appliquer au cas où deux pièces sont connues, d'où notre choix de considérer uniquement la pièce courante.

[2] A notre connaissance, aucun algorithme de Tetris n'exploite cette possibilité à l'heure actuelle.

1.3 L'algorithme de Dellacherie

On décrit ici les performances de l'algorithme écrit par Pierre Dellacherie en 2003 [7] et on détaille son fonctionnement. A notre connaissance, il s'agit de l'algorithme publiquement connu qui donne les meilleures performances : il réalise 650 000 lignes en moyenne par partie.

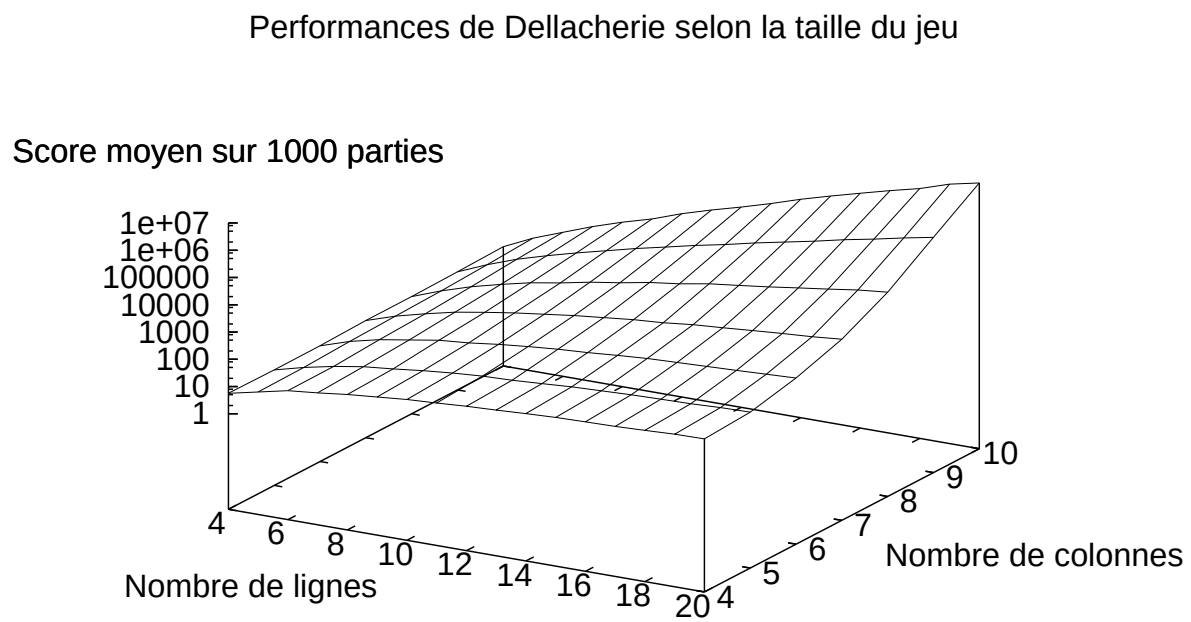


FIG. 1.2 – Score moyen (en échelle logarithmique) de l'algorithme de Dellacherie sur 1000 parties jouées en fonction de la taille du plateau de jeu.

Nous avons réimplémenté l'algorithme de Dellacherie sur notre simulateur, et nous l'avons exécuté en faisant varier la taille du plateau de 4×4 à 10×20. Le graphique de la figure 1.2 représente le score moyen réalisé sur 1000 parties pour chaque taille de plateau. Sur le plateau de taille standard (10×20), il réalise en moyenne 1.8 million de lignes et non 650 000 comme dans l'implantation d'origine. Cela provient des différences d'implantation décrites en 1.2.2 en ce qui concerne la fin de la partie.

L'algorithme de Dellacherie n'utilise aucune technique d'apprentissage. Son fonctionnement est basé sur une heuristique schématisée sur la figure 1.3. Dans une configuration donnée (un mur et une pièce courante), on essaye toutes les actions possibles. Autrement dit, on teste toutes les rotations et toutes les translations possibles de la pièce courante. Chacune de ces actions possibles engendre un nouveau mur. On évalue alors ces décisions afin de sélectionner la meilleure d'entre elles.

Le problème est donc de savoir comment évaluer une décision pertinemment, c'est-à-dire de trouver une bonne fonction d'évaluation. On souhaite que cette fonction heuristique donne une valeur élevée aux bonnes décisions et une valeur plus faible aux mauvaises. Dellacherie utilise une fonction d'évaluation qui calcule une somme pondérée des six critères détaillés à la figure 1.4. J'ai déterminé ces critères en analysant le code source du programme car ils n'étaient pas documentés de manière suffisamment précise dans [7].

Ainsi, la fonction heuristique proposée par Dellacherie peut s'écrire :

$$H(m, p, m') = -l + e - \Delta r - \Delta c - 4L' - W$$

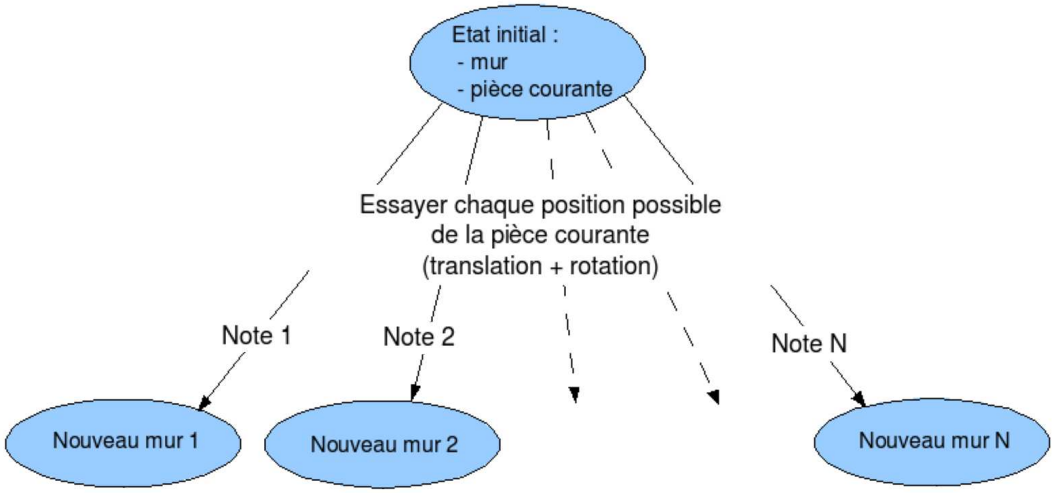


FIG. 1.3 – Principe de l'algorithme de Dellacherie

Critère	**Symbole**	**Poids**	**Description**	**Commentaires**
Landing height	l	-1	Hauteur du centre de la dernière pièce posée	Empêche d'augmenter la hauteur du mur
Eroded Piece Cells	e	1	(Nombre de lignes éliminées au dernier coup) $\times$ (Nombre de cellules éliminées dans la dernière pièce posée)	Incite à faire des lignes
Row transitions	Δr	-1	Nombre de transitions vide/plein ou plein/vide entre les cases de chaque ligne	Incite le mur à être homogènes
Column transitions	Δc	-1	Même chose pour les colonnes	
Buried Holes	L	-4	Nombre de cellules vides pour lesquelles il existe au moins une cellule pleine plus haut dans la même colonne	Pénalise les trous
Board wells	W	-1	$\sum_{p \in puits}(1 + 2 + \cdots + profondeur(p))$	Pénalise les puits[3]profonds.

FIG. 1.4 – Critères utilisés par Dellacherie

où m est le mur précédent, p est la pièce posée et m' est le nouveau mur obtenu.

Les techniques d'apprentissage étudiées au cours du stage se basent elles aussi sur une fonction heuristique qui réalise une combinaison linéaire de critères, mais elles cherchent à déterminer de manière automatique les coefficients de pondération pour un jeu de critères donné. Les critères utilisés par les différents algorithmes sont très variés et une liste détaillée est donnée en annexe A.

Nous allons maintenant détailler deux de ces techniques, la première faisant appel au contrôle optimal stochastique et la seconde étant une méthode de recherche directe des poids d'une liste de critères donnée.

[3]Un puits est une succession de cellules vides et non recouvertes dans la même colonne et telles quel leurs cases voisines à gauche et à droite soient toutes les deux pleines. Les puits profonds sont pénalisants car ils obligent à attendre une barre verticale.

Chapitre 2

Approche par Contrôle Optimal Stochastique

Cette section présente la première approche que nous avons étudiée pour résoudre le problème de Tetris : le contrôle optimal stochastique. On présente en particulier l'algorithme λ-Policy Iteration. On détaille ensuite la manière dont le contrôle optimal peut s'appliquer à Tetris, et on s'intéresse enfin aux expériences réalisées et aux résultats que nous avons obtenus par rapport aux résultats d'origine de Bertsekas et Ioffe [2] sur le problème de Tetris.

2.1 Cadre

2.1.1 Les Processus Décisionnels de Markov

Le contrôle optimal stochastique [16] considère un agent informatique devant prendre des décisions en interagissant avec son environnement de manière à maximiser un signal de récompense sur le long terme. A chaque instant t, l'agent est dans un état s_t et effectue une action a_t. Il obtient ensuite une récompense r_{t+1} et se retrouve dans l'état s_{t+1}.

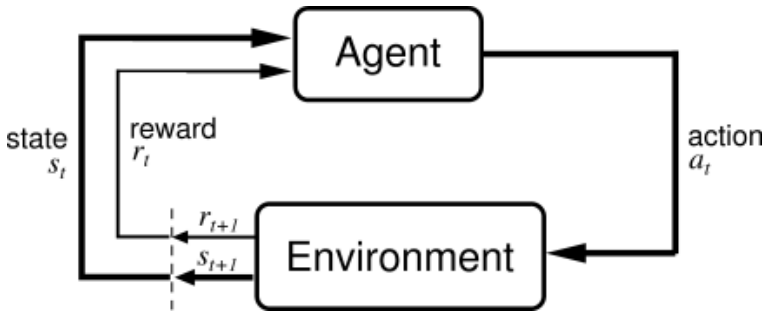


FIG. 2.1 – L'interaction agent/environnement d'après Sutton et Barto [16]

Le cadre des Processus Décisionnels de Markov permet de formaliser le contrôle optimal stochastique. Un Processus Décisionnel de Markov (PDM) est un quadruplet (S, A, T, R) où :

- S est l'espace d'états ;
- A est l'espace d'actions ;
- T est la fonction de transition : $T(s, a, s')$ est la probabilité d'arriver dans l'état s' sachant que l'on est dans l'état s et que l'on effectue l'action a.

– R est la fonction de récompense : $R(s,a) \in \mathbb{R}$ est la récompense instantanée reçue en effectuant l'action $a \in A$ depuis l'état $s \in S$.

L'objectif est de déterminer l'action optimale à effectuer dans chaque état s_t, c'est-à-dire la décision qui lui permet de maximiser la somme des récompenses futures :

$$R_t = E\left[r_{t+1} + r_{t+2} + \ldots + r_T\right]$$

où E désigne l'espérance mathématique. Notons qu'en général cette somme peut diverger lorsque le processus ne se termine pas. Il est alors usuel d'introduire alors un terme γ, un facteur d'actualisation compris entre 0 et 1 :

$$R_t = E\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right]$$

Ce facteur de pondération détermine la valeur des récompenses futures : lorsque γ est proche de 0, l'agent s'intéresse surtout aux récompenses à court terme. Si γ est proche 1, il cherchera plutôt à maximiser les récompenses à long terme. Lorsque $\gamma = 0$, l'agent ne s'intéresse qu'à la récompense immédiate.

2.1.2 Politique et fonction de valeur d'une politique

Comme le but est de choisir la meilleure action possible dans chaque état, on introduit les notions de politique et de fonction de valeur d'une politique. Une politique est une fonction $\pi : S \rightarrow A$ qui associe à chaque état l'action correspondante : $\pi(s_t) = a_t$. Lorsque l'on suit une politique π, la dynamique des états parcourus est celle d'une chaîne de Markov sur S dont les probabilités de transition sont $P(s'|s) = T(s, \pi(s), s')$.

On appelle fonction de valeur d'une politique π la fonction V^π qui associe à chaque état l'espérance du cumul des récompenses que l'on peut obtenir à partir de cet état en suivant la politique π :

$$V^\pi(s) = E_\pi[R_t | s_t = s]$$

où E_π désigne l'espérance mathématique correspondant à la chaîne de Markov induite par la politique π. La valeur d'un état représente ainsi sa qualité pour une politique donnée. Une propriété fondamentale de la fonction de valeur d'une politique π est le fait qu'elle vérifie une équation récursive, *l'équation de Bellman* [1] :

$$V^\pi(s) = R(s, \pi(s)) + \gamma . \sum_{s'} T(s, \pi(s), s') . V^\pi(s') \tag{2.1}$$

Ainsi, la valeur d'un état dépend de la récompense immédiate et de la valeur des états suivants. Cette équation récursive est le fondement de nombreux alorithmes liés aux Processus Décisionnels de Markov.

Il est usuel d'introduire l'opérateur B^π qui permet une notation condensée de l'équation (2.1) :

$$\left[B^\pi V\right](s) = \left(R(s, \pi(s)) + \gamma . \sum_{s'} T(s, \pi(s), s') . V^\pi(s') \right) \tag{2.2}$$

Cet opérateur est contractant [14] et son unique point fixe est la fonction de valeur V^π. V^π est la seule fonction de valeur qui vérifie $B^\pi V^\pi = V^\pi$.

L'équation 2.1 est un système linéaire ayant $|S|$ inconnues (les $V^\pi(s)$, pour $s \in S$). Réécrivons ce système sous forme vectorielle :

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

où V^π est le vecteur des valeurs de la politique π :

$$V^\pi = \begin{pmatrix} V^\pi(s_1) \\ \vdots \\ V^\pi(s_n)) \end{pmatrix}$$

R^π est le vecteur des récompenses obtenues en suivant la politique π :

$$R^\pi = \begin{pmatrix} R(s_1, \pi(s_1)) \\ \vdots \\ R(s_n, \pi(s_n)) \end{pmatrix}$$

et P^π est la matrice de transition induite par la politique π :

$$P^\pi = \begin{pmatrix} T(s_0, \pi(s_0), s_0) & \dots & T(s_0, \pi(s_0), s_n) \\ \vdots & T(s_i, \pi(s_i), s_j) & \vdots \\ T(s_n, \pi(s_n), s_0) & \dots & T(s_n, \pi(s_n), s_n) \end{pmatrix} \tag{2.3}$$

On obtient alors une expression de la fonction de valeur V^π :

$$\begin{aligned} V^\pi &= R^\pi + \gamma P^\pi V^\pi \\ V^\pi - \gamma P^\pi V^\pi &= R^\pi \\ V^\pi &= (I - \gamma P^\pi)^{-1} R^\pi \end{aligned} \tag{2.4}$$

2.1.3 Politique optimale et fonction de valeur optimale

Dans le cadre des PDM, l'objectif est de déterminer une politique optimale[4], c'est-à-dire une politique qui maximise l'espérance du cumul des récompenses futures. Pour cela, il est usuel d'introduire la notion de fonction de valeur optimale V^*, qui associe à chaque état la meilleure espérance possible des récompenses.

$$V^*(s) = \max_\pi V^\pi(s)$$

Il est établi qu'une telle fonction de valeur existe et qu'elle est unique [1]. Il peut exister plusieurs politiques optimales, qui partagent alors cette fonction de valeur. Si l'on connaît la fonction de valeur optimale, alors on en déduit facilement une politique optimale π^* en sélectionnant les actions menant aux états ayant la meilleure valeur (on parle de politique *gourmande* sur les valeurs de V^*) :

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma . \sum_{s'} T(s,a,s') . V^*(s') \right)$$

La fonction de valeur optimale vérifie également une équation récursive, appelée *équation d'optimalité de Bellman* [1] :

$$V^*(s) = \max_a \left(R(s,a) + \gamma . \sum_{s'} T(s,a,s') . V^*(s') \right) \tag{2.5}$$

Là aussi, on introduit un opérateur B^*

$$\left[B^* V\right](s) = \max_a \left(R(s,a) + \gamma . \sum_{s'} T(s,a,s') . V(s') \right) \tag{2.6}$$

pour avoir une notation condensée de l'équation (2.5) :

$$B^* V^* = V^*$$

L'opérateur B^* est contractant [14] et son unique point fixe est la fonction de valeur optimale V^*. V^* est ainsi la seule fonction de valeur vérifiant $B^* V^* = V^*$. Cet opérateur permet d'exprimer le fait qu'une politique π soit gourmande par rapport à une fonction de valeur V : on écrit alors $B^* V = B^\pi V$.

L'équation $B^* V^* = V^*$ est non linéaire, on ne peut donc pas calculer la fonction de valeur optimale directement comme on le faisait pour calculer la valeur d'une politique (équation (2.4)). La section suivante présente des algorithmes qui permettent de déterminer la fonction de valeur optimale.

[4]Notons que notre définition d'une politique se restreint à des politiques déterministes. Ce n'est pas une limitation, car il a été montré que parmi les politiques optimales, il en existe au moins une qui est déterministe [14].

2.2 Algorithmes

Nous venons de voir que la fonction de valeur optimale jouait un rôle central dans le cadre du contrôle optimal, puisqu'une fois la fonction de valeur optimale V^* connue, on peut en déduire une politique optimale π^* en prenant la politique gourmande sur les valeurs de V^*. Nous allons maintenant présenter deux algorithmes permettant de calculer la fonction de valeur optimale : l'Itération de la Valeur et l'Itération de la Politique, qui sont deux algorithmes standards du contrôle optimal [16]. Puis on présentera λ-Policy Iteration, un algorithme qui unifie l'Itération de la Valeur et l'Itération de la Politique et auquel on s'intéressera plus particulièrement dans la suite de ce rapport.

2.2.1 Itération de la Valeur

L'algorithme Itération de la Valeur (*Value Iteration*) [1], issu de la programmation dynamique, est l'un des algorithmes de base des Processus de Décision Markoviens. Il permet de trouver la fonction de valeur optimale V^*. Pour cela, il part d'une fonction de valeur arbitraire, et l'améliore peu à peu à chaque itération jusqu'à converger vers la fonction de valeur optimale. L'étape d'amélioration consiste simplement à appliquer l'opérateur de Bellman B^* présenté plus haut.

Itération de la Valeur
Initialisation :
$t \leftarrow 0$
V_0 arbitraire
Répéter :
$V_{t+1} \leftarrow B^* V_t$
$t \leftarrow t+1$
Jusqu'à $\max_s \lvert V_t(s) - V_{t-1}(s)\rvert < \epsilon$

2.2.2 Itération de la Politique

L'Itération de la Politique (*Policy Iteration*) [1] est un autre algorithme itératif qui permet de calculer la fonction de valeur optimale. A chaque itération, on a une fonction de valeur V_t et une politique π_t. La politique π_{t+1} est alors choisie comme la politique gourmande sur les valeurs de V_t, puis V_{t+1} est calculée comme la valeur de la politique π_{t+1} (voir l'équation (2.4)), et ansi de suite.

Itération de la Politique
Initialisation :
$t \leftarrow 0$
V_0 arbitraire
Répéter :
π_{t+1} choisie telle que $B^{\pi_{t+1}} V_t = B^* V_t$
$V_{t+1} \leftarrow (I - \gamma P^{\pi_{t+1}})^{-1} R^{\pi_{t+1}}$
$t \leftarrow t+1$
Jusqu'à $\max_s \lvert V_t(s) - V_{t-1}(s)\rvert < \epsilon$

Avec l'Itération de la Valeur, la fonction de valeur V_{t+1} est simplement mise à jour par une application de l'opérateur de Bellman ($V_{t+1} = B^* V_t$). L'Itération de la Valeur économise ainsi la phase d'évaluation de la politique, qui est une phase coûteuse lorsque le nombre d'états est élevé. Mais l'Itération de la Valeur nécessite en général un plus grand nombre d'itérations que l'Itération de la Politique pour converger [16].

2.2.3 λ-Policy Iteration

λ-Policy Iteration (λPI) est un algorithme de contrôle optimal stochastique proposé par Bertsekas et Tsitsiklis en 1996 [2, 3]. Bien qu'il soit peu connu, il permet de présenter de manière unifiée les deux algorithmes que nous venons de voir : l'Itération de la Valeur et l'Itération de la Politique. Pour cela, un paramètre $\lambda \in [0, 1]$ spécifie si l'algorithme est plus proche de l'Itération de la Politique (λ proche de 1) ou de l'Itération de la Valeur (λ proche de 0).

Il existe également une version approximative de l'algorithme λPI, présentée dans la section suivante. Cette version approximative s'applique aux problèmes où le nombre d'états est élevé. Nous nous intéressons dans un premier temps à la version exacte.

λ-Policy Iteration est un algorithme itératif. A chaque itération, on considère un couple (V_t, π_t) où π_t est la politique courante et où V_t peut être vu comme une approximation de la fonction de valeur V^{π_t}.

λ-Policy Iteration
Initialisation :
$t \leftarrow 0$
V_0 arbitraire
Répéter :
π_{t+1} choisie telle que $B^{\pi_{t+1}} V_t = B^* V_t$
$V_{t+1} \leftarrow V_t + (I - \gamma\lambda P^{\pi_{t+1}})^{-1}(B^{\pi_{t+1}} V_t - V_t)$
$t \leftarrow t + 1$
Jusqu'à $\max_s \lvert V_t(s) - V_{t-1}(s)\rvert < \epsilon$

Comme dans l'Itération de la Politique, la nouvelle politique π_{t+1} est choisie comme la politique gourmande sur V_t, c'est-à-dire la politique qui sélectionne pour chaque état la meilleure action selon V_t. Puis la nouvelle fonction de valeur V_{t+1} est calculée comme $V_{t+1} = V_t + \Delta_t$ où Δ_t est le vecteur défini par

$$\Delta_t = (I - \gamma\lambda P^{\pi_{t+1}})^{-1}(B^{\pi_{t+1}} V_t - V_t) \tag{2.7}$$

où $P^{\pi_{t+1}}$ est la matrice de transition associée à la politique π_{t+1}. Il a été montré que λ-Policy Iteration converge vers un couple valeur-politique optimal [2].

Comme expliqué par ses auteurs [2], l'algorithme est lié à la notion de *différences temporelles* (TD) introduites dans le cadre de l'apprentissage par renforcement [16], une variante du contrôle optimal. On peut en effet montrer que :

$$\Delta_t(s) = \sum_{m=0}^{\infty} E_{\pi_{t+1}} \left[(\gamma\lambda)^m d_t(s_m, s_{m+1}) | s_0 = s\right] \tag{2.8}$$

où $d_t(s, s')$ est la différence temporelle[5] associée à chaque transition (s, s') :

$$d_t(s, s') = R(s, \pi_{t+1}(s)) + \gamma V_t(s') - V_t(s)$$

Généralisation de l'Itération de la Valeur et de l'Itération de la Politique

Comme évoqué en introduction de l'algorithme, $\lambda \in [0, 1]$ est un paramètre qui détermine si l'algorithme est plus proche de l'Itération de la Valeur ou de l'Itération de la Politique. En effet, considérons le cas où $\lambda = 0$. La mise à jour de la fonction de valeur devient alors :

$$V_{t+1} = V_t + B^{\pi_{t+1}} V_t - V_t = B^{\pi_{t+1}} V_t = B^* V_t$$

[5] Lorsque les différences temporelles sont égales à zéro, l'équation de Bellman est vérifiée [16].

On retrouve ainsi l'algorithme Itération de la Valeur (voir 2.2.1). De même, considérons le cas où $\lambda = 1$:

$$\begin{aligned} V_{t+1} &= V_t + (I - \gamma P^{\pi_{t+1}})^{-1}(B^{\pi_{t+1}} V_t - V_t) \\ &= (I - \gamma P^{\pi_{t+1}})^{-1}((I - \gamma P^{\pi_{t+1}} V_t) + B^{\pi_{t+1}} V_t - V_t) \\ &= (I - \gamma P^{\pi_{t+1}})^{-1}(V_t - \gamma P^{\pi_{t+1}})V_t + B^{\pi_{t+1}} V_t - V_t) \\ &= (I - \gamma P^{\pi_{t+1}})^{-1} R^{\pi_{t+1}} \end{aligned}$$

On retrouve ici l'algorithme Itération de la Politique (voir 2.2.2).

2.2.4 Version approximative de λ-Policy Iteration

Nous venons de présenter la version exacte de λ-Policy Iteration et de montrer comment elle généralisait les deux algorithmes standards que sont l'Itération de la Valeur et l'Itération de la Politique. Les auteurs de cet algorithme ont également proposé [2] une version approximative, destinée à résoudre des problèmes de grande taille en effectuant des simulations et en approximant la fonction de valeur.

Précisément, la fonction de valeur est approximée par une combinaison linéaire de la forme :

$$V^r(s) = r(0) + \sum_{k=1}^{K} r(k)\Phi_k(s)$$

où $r = (r(0) \dots r(K))$ est un vecteur de poids que l'on cherhe à estimer, et les $\Phi_k(s)$ sont des fonctions à valeurs réelles définies sur l'espace d'états. Chaque $\Phi_k(s)$ est une fonction de base fixée qui calcule une certaine caractéristique d'un état. La combinaison des $\Phi_k(s)$, pondérée par les $r(k)$, doit alors permettre d'approcher au mieux la fonction de valeur. Chaque valeur de r caractérise une fonction de valeur V^r sur l'espace d'états. On cherche ainsi à déterminer r de manière à ce que V^r soit le plus proche possible de la fonction de valeur optimale. L'intérêt de cette approximation est que le problème se réduit à calculer k valeurs, et le nombre de fonctions de base k est fixé et très inférieur au nombre d'états $|S|$.

De manière analogue à la version exacte de l'algorithme, on itère sur un couple valeur-politique (r_t, π_t). Pour un vecteur de poids r_t donné, π_{t+1} est choisi comme la politique gourmande sur les valeurs de V^{r_t} :

$$B^{\pi_{t+1}} V^{r_t} = B^* V^{r_t}$$

Pour la mise à jour de r_t, on utilise une approximation de Δ_t. On génère un ensemble de M trajectoires à l'aide de la politique π_{t+1}. Pour chaque trajectoire m, $(s_{m,0}, s_{m,1}, \dots, s_{m,N_m-1}, s_{m,N_m})$ désigne la séquence d'états parcourus dans la $m^{\text{ième}}$ partie, où s_{m,N_m} est l'état final. r_{t+1} est alors mis à jour en minimisant (au sens des moindres carrés) l'erreur du système linéaire suivant d'inconnue $\mathbf{r}$:

$$\begin{aligned} V^{\mathbf{r}}(s_{m,N_m}) &\simeq 0 \qquad (2.9) \\ V^{\mathbf{r}}(s_{m,N_m-1}) &\simeq V^{r_t}(s_{m,N_m-1}) + d_t(s_{m,N_m-1}, s_{m,N_m}) \\ V^{\mathbf{r}}(s_{m,N_m-2}) &\simeq V^{r_t}(s_{m,N_m-2}) + d_t(s_{m,N_m-2}, s_{m,N_m-1}) + \gamma\lambda d_t(s_{m,N_m-1}, s_{m,N_m}) \\ &\vdots \quad \vdots \\ V^{\mathbf{r}}(s_{m,k}) &\simeq V^{r_t}(s_{m,k}) + \sum_{s=k}^{N_{m-1}} (\gamma\lambda)^{s-k} d_t(s_{m,s}, s_{m,s+1}) \qquad (2.10) \\ &\vdots \quad \vdots \\ V^{\mathbf{r}}(s_{m,0}) &\simeq V^{r_t}(s_{m,0}) + \sum_{s=0}^{N_{m-1}} (\gamma\lambda)^{s} d_t(s_{m,s}, s_{m,s+1}) \end{aligned}$$

pour chaque trajectoire m, où les différences temporelles sont données par

$$d_t(s_{m,N_m-1}, s_{m,N_m}) = R(s_{m,N_m-1}, \pi_{t+1}(s_{m,N_m-1})) - V^{r_t}(s_{m,N_m-1}) \tag{2.11}$$

et pour tout $s < N_m - 1$

$$d_t(s_{m,s}, s_{m,s+1}) = R(s_{m,s}, \pi_{t+1}(s_{m,s})) + \gamma V^{r_t}(s_{m,s+1}) - V^{r_t}(s_{m,s})$$

On notera que les équations (2.9) et (2.11) concernent les états terminaux pour lesquels il n'y a plus de récompense.

Dans le système linéaire ci-dessus, on cherche à minimiser l'erreur commise en utilisant l'approximation de Δ_t. Si on générait un nombre infini de trajectoires, et si on avait un égalité à la place d'une approximation, alors l'équation (2.10) reviendrait au cas de la version exacte de l'algorithme (voir l'équation (2.8)).

2.3 Modélisation de Tetris

Nous venons de présenter le contrôle optimal stochastique. Revenons maintenant au problème de Tetris pour lui appliquer le contrôle optimal. Dans le cadre des Processus Décisionnels de Markov, on peut modéliser le problème de Tetris comme indiqué dans le tableau suivant :

Etat	$s = (m, p)$	Configuration du mur m et pièce courante p
Action	a	Orientation et translation horizontale de la pièce avant de la lâcher
Récompense	$R(s, a)$	Nombre de lignes réalisées en effectuant l'action a dans l'état s
Transition	$T(s, a, s')$	Probabilité d'arriver dans l'état s' après avoir effectué l'action a dans l'état s

Nous avons vu que toute partie à Tetris se terminait nécessairement [5], donc on peut prendre un facteur d'actualisation $\gamma = 1$ (voir section 2.1.1). A partir de cette modélisation, les résultats concernant les PDM s'appliquent au problème de Tetris : on sait qu'il existe une politique optimale à Tetris, et la fonction de valeur $V^\pi(s)$ d'un état représente le nombre de lignes moyen qui peuvent être faites jusqu'à la fin de la partie en suivant une politique π.

La fonction de récompense $R(s, a)$ est connue puisque pour un état s donné, on connaît à l'avance le nombre de lignes réalisées par toute action a. La fonction de transition $T(s, a, s')$ peut quant à elle se découper en deux phases (figure 2.2). Lorsque l'on pose une pièce, l'action qui est effectuée détermine la nouvelle configuration du mur, puis la nouvelle pièce est générée avec une distribution uniforme parmi les 7 pièces existantes.

FIG. 2.2 – Les deux phases d'une transition entre deux états. A partir d'un état $s = (m, p)$, l'action a détermine un nouveau mur m', puis un tirage aléatoire génère une nouvelle pièce p' pour donner le nouvel état $s' = (m', p')$.

Chaque état s est un couple (m, p) composé d'un mur $m \in M$ et d'une pièce courante $p \in P$. Ecrivons l'équation de Bellman (voir équation (2.1)) appliquée au problème de Tetris :

$$V^\pi(m, p) = R((m, p), \pi(m, p)) + \sum_{(m', p') \in S} T((m, p), \pi(m, p), (m', p')) + V^\pi(m', p')$$

Nous allons voir que l'on peut utiliser une fonction de valeur plus simple, définie sur l'ensemble des murs M et non sur l'ensemble des états complets $S = M \times P$ [2]. Le mur m' étant généré de manière déterministe, notons $f(m, p, a)$ le mur obtenu en effectuant l'action a depuis le mur m avec la pièce p. De plus, la pièce p' étant générée de manière uniforme, on obtient :

$$\begin{aligned} V^\pi(m,p) &= R((m,p),\pi(m,p)) + \sum_{(m',p')\in S} T((m,p),\pi(m,p),(m',p'))\, V^\pi(m',p') \\ &= R((m,p),\pi(m,p)) + \frac{1}{|P|}\sum_{p'\in P} V^\pi(f(m,p,\pi(m,p)),p') \end{aligned} \tag{2.12}$$

Pour une politique π fixée, on définit alors la nouvelle fonction de valeur $\widetilde{V}^\pi$ suivante sur l'espace des murs M :

$$\widetilde{V}^\pi(m) = \frac{1}{|P|}\sum_{p\in P} V^\pi(m,p) \tag{2.13}$$

On a d'après (2.12) :

$$V^\pi(m,p) = R((m,p),\pi(m,p)) + \widetilde{V}^\pi(f(m,p,\pi(m,p)))$$

et on obtient à partir de (2.13) une forme simplifiée de l'équation de Bellman :

$$\widetilde{V}^\pi(m) = \frac{1}{|P|}\sum_{p\in P} R((m,p),\pi(m,p)) + \widetilde{V}^\pi(f(m,p,\pi(m,p)))$$

De même, on peut réécrire l'équation d'optimalité de Bellman (2.1) sous la forme suivante [2] :

$$\widetilde{V}^*(m) = \frac{1}{|P|}\sum_{p\in P} \max_a R((m,p),a) + \widetilde{V}^*(f(m,p,a))$$

Cette nouvelle fonction de valeur possède moins d'états puisque la pièce courante n'entre plus en compte. En utilisant cette fonction de valeur, il n'y a pas de perte d'informations. En effet, si l'on connaît la fonction de valeur optimale sur les murs $\widetilde{V}^*$, on retrouve facilement la fonction de valeur optimale sur les états complets V^* :

$$V^*(m,p) = \max_a R((m,p),a) + \widetilde{V}^*(f(m,p,a))$$

2.4 Expériences

Maintenant que le contrôle optimal a été présenté et que nous avons modélisé Tetris dans ce cadre, intéressons-nous aux expériences de contrôle optimal réalisées au cours du stage.

2.4.1 Itération de la Valeur

Une première approche a été de simplifier le problème de Tetris de manière à diminuer le nombre d'états. Nous avons considéré deux espaces de taille réduite : 4×5 et 5×5 . Ainsi, le nombre d'états devient de l'ordre du million : $2^{4*5} = 2^{20} = 1\ 048\ 576$ et $2^{5*5} = 2^{25} = 33\ 554\ 432$ respectivement. L'algorithme Itération de la Valeur (autrement dit 0-Policy Iteration) devient alors utilisable.

Résultats

L'espace de la zone de jeu étant réduit de manière drastique, la politique optimale obtenue ne réalise que 12.6 lignes en moyenne sur l'espace de taille 4×5, et 13.7 lignes en 5×5. Ces scores sont une moyenne sur 50 000 parties jouées avec la politique obtenue après 100 itérations. En observant l'évolution des performances au cours des itérations, on remarque que des scores proches de l'optimal sont atteints après un petit nombre d'itérations : 90% du score optimal est atteint après 3 itérations en taille 4×5, et après 4 itérations en taille 5×5. Pour ces tailles de plateau, cela signifie qu'un contrôleur qui calculerait le nombre de lignes moyen en explorant tous les états accessibles 3 ou 4 coups à l'avance[6] donnerait des performances proches de l'optimal.

Un intérêt de connaître la fonction de valeur optimale est que l'on connaît ainsi le meilleur score qu'il est possible d'atteindre en moyenne. On peut alors s'en servir de référence en lançant des simulations sur un espace de 4×5 ou 5×5 avec d'autres politiques pour avoir une idée de leur qualité par rapport à la politique optimale. Ainsi, l'algorithme de Pierre Dellacherie réalise en moyenne 9.78 lignes en 4×5 et 10.76 lignes en 5×5, sur 50 000 parties jouées.

2.4.2 λ-Policy Iteration

Pour des tailles de jeu plus importantes comme 10×10 ou la taille standard (10×20), le nombre d'états est trop élevé pour résoudre l'équation de Bellman directement. Nous avons donc appliqué à Tetris la version approximative de λ-Policy Iteration, comme Bertsekas et Ioffe [2] l'ont fait en 1996.

Critères de Bertsekas et Ioffe

Pour leurs expériences, Bertsekas et Ioffe [2] ont proposé un ensemble de $2w+1$ fonctions de base Φ_k (ou critères) décrits sur la figure 2.3, où w est la largeur de la zone de jeu.

Critère	**Symbole**	**Description**	**Commentaires**
Column Height	h_k	Hauteur de la $k^{\text{ième}}$ colonne du mur	On a w valeurs (une par colonne)
Column Difference	Δh_k	Différence de hauteur entre chaque colonne et sa voisine : $\lvert h_k - h_{k+1} \rvert$	On a $w-1$ valeurs
Maximum height	H	Hauteur maximale du mur : $\max_k h_k$	Empêche de faire monter le mur
Holes	L	Nombre de cellules vides pour lesquelles il existe au moins une cellule pleine plus haut dans la même colonne	Empêche de faire des trous

FIG. 2.3 – Critères proposés par Bertsekas et Ioffe [2]

La fonction de valeur approximative est donc définie par :

$$V^r(s) = r(0) + \sum_{k=1}^{w} r(k) h_k + \sum_{k=1}^{w-1} r(k+w) \Delta h_k + r(2w) H + r(2w+1) L$$

Bien que les performances atteintes avec ces 22 critères aient été battues par la suite (notamment par Dellacherie [7]), de nombreuses publications les utilisent afin de comparer leurs résultats avec ceux de Bertsekas et Ioffe. Ainsi, ces critères à l'origine introduits pour illustrer λ-Policy Iteration sont maintenant considérés comme standards par beaucoup d'algorithmes de Tetris.

[6] Le nombre d'états à parcourir augmente de manière exponentielle avec la profondeur d'exploration, et pour explorer 4 coups à l'avance, il est déjà de l'ordre du milliard.

Nos résultats

Nous avons utilisé pour nos expériences le même jeu de critères que Bertsekas et Ioffe [2]. Nous avons également initialisé le vecteur de poids initial de la même manière [2] : $r(2w) = -10$, $r(2w+1) = -1$ et $r(k) = 0$ pour $k < 2w$. La politique gourmande correspondante atteint en moyenne une trentaine de lignes [2]. $M = 100$ parties étaient utilisées pour générer des échantillons et pour évaluer les performances atteintes (voir la description de la version approximative de λ-Policy Iteration, à la section 2.2.4). λ-Policy Iteration étant un algorithme stochastique, nous avons lancé l'algorithme 10 fois.

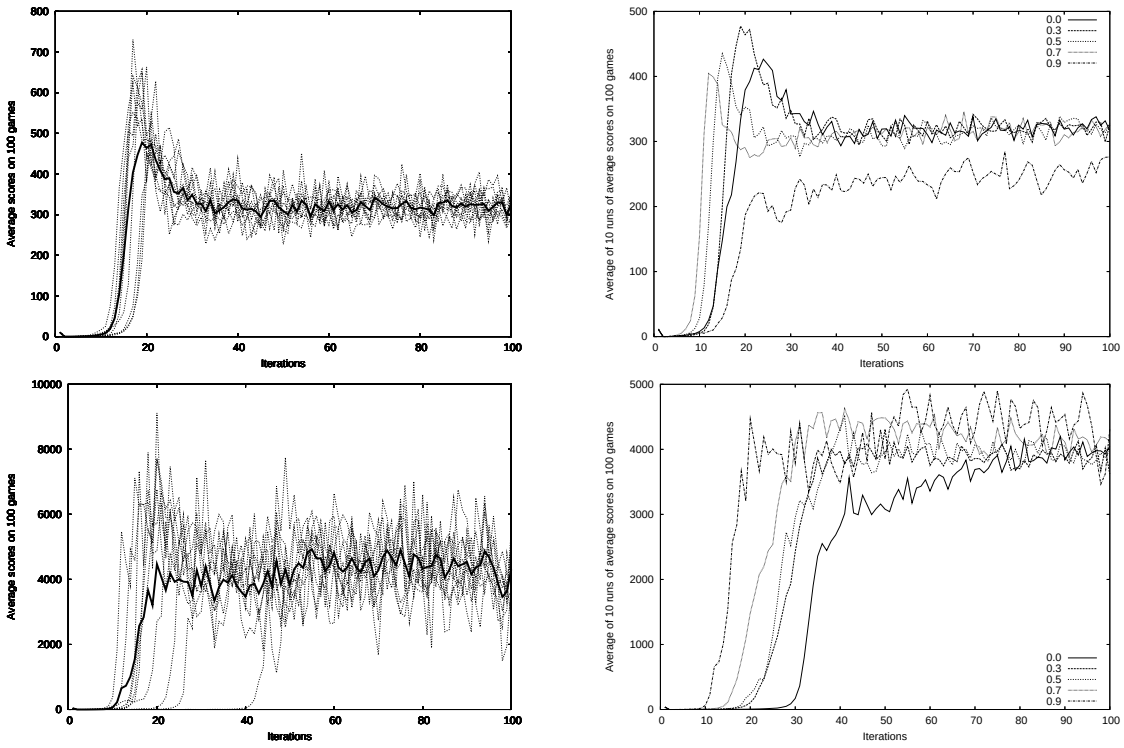


FIG. 2.4 – Score moyen sur 100 parties à chaque itération de λPI. **Haut** : taille 10×10. **Bas** : taille 10×20. **Gauche** : 10 exécutions de λPI et l'exécution moyenne (en gras), avec $\lambda = 0.3$ en haut et $\lambda = 0.9$ en bas. **Droite** : Moyenne de 10 exécutions de λPI avec différentes valeurs de λ ; la courbe ayant le plus haut pic (en haut : $\lambda = 0.3$, en bas : $\lambda = 0.9$) est la même que celle en gras à gauche.

Sur la figure 2.4 sont représentées les courbes d'apprentissage en taille 10×10 (en haut) et en taille 10×20 (en bas). Dans les deux cas, le graphe de gauche représente pour une valeur de λ donnée les 10 exécutions de λPI (chaque point étant le score moyen atteint à l'itération courante sur les $M = 100$ parties) et la moyenne point-à-point des 10 courbes (en gras). A droite est représentée la courbe moyenne de 10 exécutions pour chacune des valeurs suivantes de λ : 0.0, 0.3, 0.5, 0.7 et 0.9. Le graphe de gauche représente les 10 exécutions correspondant à la valeur de λ qui semblait être la meilleure d'après le graphe de droite (c'est-à-dire la courbe ayant le plus haut pic). En taille 10×10, la meilleure valeur expérimentale est $\lambda = 0.3$, tandis que c'est $\lambda = 0.9$ en 10×20. L'ensemble des courbes de cette expérience est donnée en annexe B.

On peut faire les observations suivantes :

– Bien que l'algorithme soit initialisé avec une politique capable de faire une trentaine de lignes, les performances tombent à zéro et c'est seulement après quelques itérations (typiquement autour de 10) qu'elles s'améliorent. Ce phénomène est dû au fait que la fonction de valeur initiale est mauvaise : avec les poids de départ (nous avons utilisés les mêmes que dans [2] afin de reproduire

leurs résultats), la fonction de valeur initiale est négative partout alors que la fonction de valeur optimale (le meilleur score moyen possible) est positif partout. Nous avons effectué des expériences en initialisant les poids de manière aléatoire, et les courbes d'apprentissage sont similaires.

– Mis à part lorsque $\lambda = 0.9$ en taille 10×10 (figure 2.4, à droite) où la variance est probablement trop importante lors de la phase de minimisation aux moindres carrés, les performances augmentent globalement plus tôt pour les grandes valeurs de λ, c'est-à-dire pour les valeurs qui rendent l'algorithme plus proche de l'Itération de la Politique. Ce n'est pas surprenant étant donné que l'Itération de la Politique requiert moins d'itérations que l'Itération de la Valeur en général [16]. Mais au final, les mêmes performances sont atteintes quelle que soit la valeur de λ.
– La courbe d'apprentissage typique comporte un pic après quelques itérations, baisse un peu, puis se stabilise. Une interprétation possible de cette chute est que lorsque les scores augmentent, l'architecture linéaire a plus de difficultés à approximer la fonction de valeur.
– Quantitativement, on observe que les meilleurs scores moyens atteints (les pics les plus hauts) sont d'environ 700 lignes en 10×10 et de 8000 à 9000 en 10×20.

Nos résultats diffèrent qualitativement et quantitativement de ceux d'origine, publiés dans [2], malgré le fait que les expériences aient été menées dans les mêmes conditions. A propos de leurs résultats, les auteurs écrivent[7] : "*Une observation intéressante et assez paradoxale est que des performances élevées sont atteintes après relativement peu d'itérations, mais les performances baissent ensuite significativement. Nous n'avons pas d'explications pour ce phénomène intriguant, qui est apparu avec toutes nos expériences*". Nous proposons une hypothèse qui expliquerait ces résultats "intriguants". Ces derniers pourraient être dus à une différence subtile d'implantation. En effet, nous avons réussi à reproduire des courbes d'apprentissages similaires à celles de [2] en modifiant légèrement notre implantation de λPI : nous avons supprimé les traitements particuliers des états terminaux indiqués dans les équations (2.9) et (2.11). Plus précisément, si on les remplace par les équations suivantes :

$$V^{\mathbf{r}}(s_{m,N_m}) \simeq V^{r_t}(s_{m,N_m}) \tag{2.14}$$

$$d_t(s_{m,N_m-1}, s_{m,N_m}) = R(s_{m,N_m-1}, \pi_{t+1}(s_{m,N_m-1})) + \gamma V^{r_t}(s_{m,N_m}) - V^{r_t}(s_{m,N_m-1}) \tag{2.15}$$

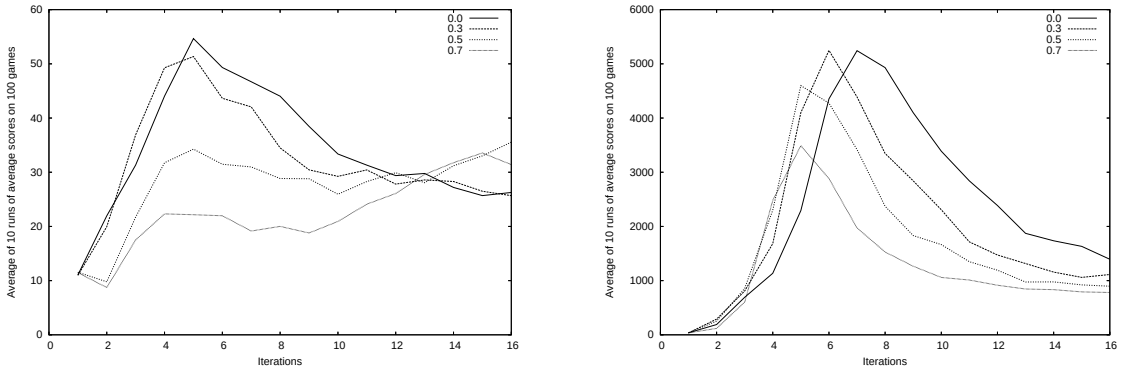


FIG. 2.5 – Score moyen à chaque itération de λPI, modifié de manière à ressembler aux résultats de [2]. Les courbes représentent la moyenne de 10 exécutions de l'algorithme. **Gauche** : taille 10×10. **Droite** : taille 10×20.

c'est-à-dire si on remplace la valeur de l'état terminal (0) par $V^{r_t}(s_{m,N_m})$, qui est la valeur calculée par la combinaison linéaire de critères à partir du mur final s_{m,N_m}, alors on obtient les résultats représentés

[7] Dans l'article original : "*An interesting and somewhat paradoxical observation is that a high performance is achieved after relatively few policy iterations, but the performance gradually drops significantly. We have no explanation for this intriguing phenomenon, which occurred with all of the successful methods that we tried*"

sur la figure 2.5. Cette figure représente les performances obtenues en fonction des itérations, en taille 10×10 (à gauche) et en taille 10×20 (à droite). L'évolution des performances correspond qualitativement aux courbes publiées dans [2] et illustre la citation ci-dessus à propos du "phénomène intriguant". Il faut noter que dans cette version modifiée, l'implantation de l'algorithme λ-Policy Iteration est erronée : en particulier, les cas de $\lambda = 0$ et de $\lambda = 1$ devraient correspondre à des versions approximatives de l'Itération de la Valeur et à l'Itération de la Politique [3]. Or, ce n'est plus le cas après cette modification.

En résumé, nous avons reproduit une approche existante [2], qui applique à Tetris la version approchée de λ-Policy Iteration. Alors que les auteurs observaient une chute paradoxale des performances après quelques itérations, nous obtenons des courbes usuelles d'apprentissage. Nous proposons une explication à ce phénomène, en mettant en évidence une erreur potentielle d'implantation.

Chapitre 3

Approche par Entropie Croisée

Nous venons d'étudier une approche du problème Tetris basée sur un algorithme de contrôle optimal stochastique, λ-Policy Iteration. D'autres approches basées sur le contrôle optimal ont été menées sur le problème de Tetris [10, 11, 12], en atteignant des performances similaires (de 3000 à 7000 lignes en moyenne). Ces résultats restent toutefois très inférieurs au score moyen de l'algorithme de Dellacherie [7] qui est de 650 000 lignes (ou 1.8 million de lignes sur notre simulateur). Récemment, Szita et Lörincz [17] ont appliqué à Tetris la méthode d'entropie croisée qui est une technique d'optimisation proche des algorithmes génétiques. Ils ont obtenu des résultats allant jusque 350 000 lignes de moyenne, dépassant ainsi les approches se basant sur le contrôle optimal. Nous nous sommes donc intéressés à cette méthode, bien qu'il ne s'agisse pas de contrôle optimal à proprement parler. Nous avons reproduit les mêmes expériences, et étudié l'influence des différents paramètres de l'algorithme en lançant de nombreuses simulations sur un jeu de taille 10×10. Finalement, nous avons obtenu à partir cette méthode une heuristique qui réalise des performances jamais atteintes jusqu'à présent à notre connaissance (40 millions de lignes en moyenne).

3.1 Description de l'algorithme

La méthode d'entropie croisée [6] est un algorithme stochastique itératif qui cherche à résoudre un problème d'optimisation de la forme

$$w^* = \arg\max_w S(w)$$

où S est une fonction que l'on souhaite maximiser. La méthode d'entropie croisée itère non pas sur des solutions, mais sur une distribution de solutions. On considére une famille de distributions paramétrées $\mathcal{F}$ et on cherche à déterminer une distribution de probabilité $f \in \mathcal{F}$, qui génère des solutions w les plus proches possibles de l'optimal w^*. Pour cela, à chaque itération, on considère une distribution $f_t \in \mathcal{F}$ qui peut génèrer des solutions w aléatoirement, et on cherche une distribution f_{t+1} qui soit meilleure. On fixe un seuil γ_t au-dessus duquel on considère qu'une solution w est de bonne qualité, c'est-à-dire $S(w) > \gamma_t$. Notons alors g_{γ_t} la distribution de probabilité qui génère uniformément les solutions dont la valeur est supérieure à ce seuil γ_t. On cherche donc une distribution $f_{t+1} \in \mathcal{F}$ qui soit la plus proche possible[8] de g_{γ_t} : ainsi, les solutions de meilleure qualité auront une plus grande probabilité d'être générés. L'idée de la méthode d'entropie croisée est que pour de nombreux types de distributions, cette distribution f_{t+1} peut être estimée à partir d'exemples générés par f_t.

[8] La *mesure d'entropie croisée* [6] (ou *distance de Kullback-Leibler*) permet d'exprimer une notion de distance entre deux distributions de probabilité.

Par exemple, dans le cas où la distribution f_t suit une loi gaussienne, il se trouve que la distribution gaussienne la plus proche de g_{γ_t} est celle ayant pour paramètres l'espérance et la variance des échantillons générées par loi g_{γ_t} [17]. On peut estimer ces paramètres à partir d'échantillons générés par la distribution f_t. On génère ainsi N solutions avec la distribution f_t, et on les évalue à l'aide de la fonction S. On sélectionne une proportion $\rho \in]0,1[$ des meilleures solutions (ce qui revient à fixer γ_t à un certain seuil), et les paramètres de la nouvelle distribution gaussienne f_{t+1} sont alors la moyenne et la variance empiriques des solutions conservées.

3.2 Application à Tetris

Voyons maintenant comment Szitza et Lörincz [17] ont appliqué l'algorithme d'entropie croisée au problème de Tetris. L'algorithme se base bien sur une fonction qui évalue la qualité d'une décision, mais contrairement aux techniques de contrôle optimal, cette fonction n'est pas une estimation de l'espérance de la récompense future : il s'agit simplement d'une fonction heuristique qui évalue chaque décision possible par une somme pondérée de critères et sélectionne la meilleure d'entre elles, à la manière de l'algorithme de Dellacherie (voir figure 1.3). Cette fonction d'évaluation peut s'écrire de la manière suivante :

$$H_w(s) = \sum_{i=1}^{m} w_i \Phi_i(s)$$

où les Φ_i sont des fonctions de bases (critères) telles que celles proposées par Bertsekas et Ioffe [2] et présentés précédemment (voir figure 2.3). w est le vecteur de poids que l'on cherche à régler de manière à obtenir la meilleure heuristique possible. Le problème revient à trouver un vecteur w qui induise une stratégie de bonne qualité.

Dans l'approche de Szita et Lörincz [17], chaque poids du vecteur w est régi par une loi gaussienne. Nous sommes donc dans le cas gaussien expliqué ci-dessus, pour lequel la mise à jour de la distribution peut se faire en calculant la moyenne et la variance d'un échantillon des meilleurs vecteurs. Les fonctions de base Φ_k qu'ils ont choisies sont les 22 critères introduits par Bertsekas et Ioffe [2] (figure 2.3). Voici plus en détail le fonctionnement de l'algorithme qu'ils ont proposé pour appliquer la méthode d'entropie croisée à Tetris :

Entropie croisée bruitée appliquée à Tetris [17]

Initialisation :
- $\mu_i \leftarrow 0; \sigma_i \leftarrow 100$ pour i de 1 à m
- $t \leftarrow 0$
- $\rho \leftarrow 0.1, N \leftarrow 100, k \leftarrow 1$

Répéter :
- $t \leftarrow t + 1$
- Générer N vecteurs de poids w, avec $w_i \sim \mathcal{N}(\mu_i, \sigma_i)$
- Evaluer chacun de ces vecteurs en jouant k parties
- Sélectionner les $\rho * N$ meilleurs vecteurs
- Mettre à jour μ_i et σ_i pour i de 1 à m :
  - $\mu_i \leftarrow$ (moyenne du poids i parmi les vecteurs sélectionnés)
  - $\sigma_i^2 \leftarrow$ (variance du poids i parmi les vecteurs sélectionnés) $+Z_t$
- Evaluer les nouveaux paramètres en jouant 30 parties avec les poids moyens μ_i

Jusqu'à convergence ($\sigma_i = 0$ pour tout i)

Z_t est un terme de bruit, ajouté à la mise à jour de la variance afin d'éviter une convergence trop rapide vers un optimum local. Ce terme peut être constant ou décroître à chaque itération. On parle

alors de la méthode d'entropie croisée bruitée (*noisy cross-entropy*).

3.3 Expériences

Szita et Lörincz ont effectué trois expériences, avec trois types de bruit : pas de bruit ($Z_t = 0$), un bruit constant ($Z_t = 4$) et un bruit linéairement décroissant ($Z_t = \max(5 - t/10, 0)$). Dans ce dernier cas, le bruit décroît linéairement pendant 50 itérations puis reste à zéro ensuite. Ils indiquent avoir obtenu les meilleurs résultats avec le bruit linéairement décroissant, en réalisant une moyenne de 350 000 lignes. Il s'agit à notre connaissance du meilleur score réalisé par un algorithme d'apprentissage, bien qu'il soit encore inférieur au score de l'algorithme de Dellacherie.

Dans l'article original de Szita et Lörincz [17], chacune de ces trois expériences n'a été exécutée qu'une seule fois et le tout a duré un mois. Nous avons souhaité reproduire les mêmes expériences, mais en effectuant 10 fois chacune des trois simulations. En effet, nos essais préliminaires montraient que plusieurs exécutions avec les mêmes paramètres pouvaient donner des résultats très différents. Ces 10 exécutions ont duré une semaine sur notre simulateur.

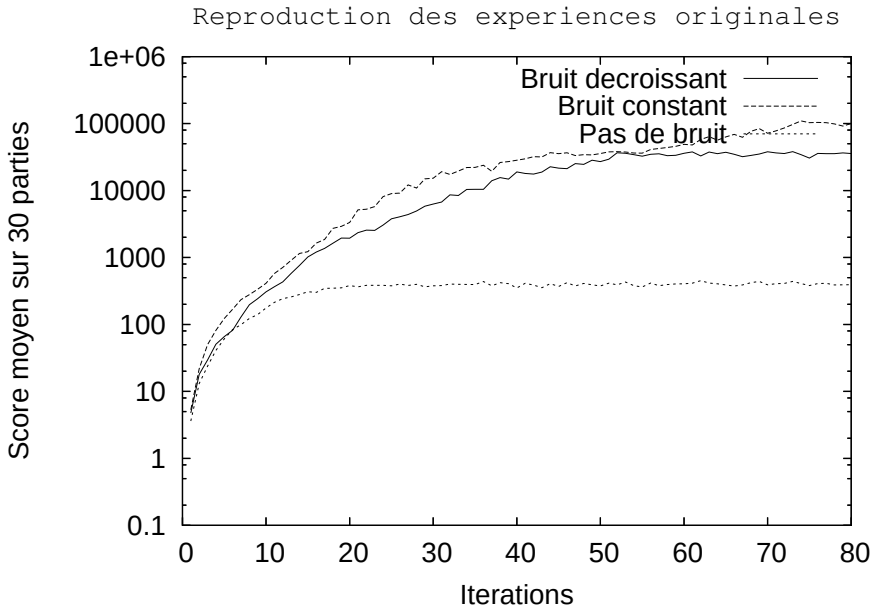
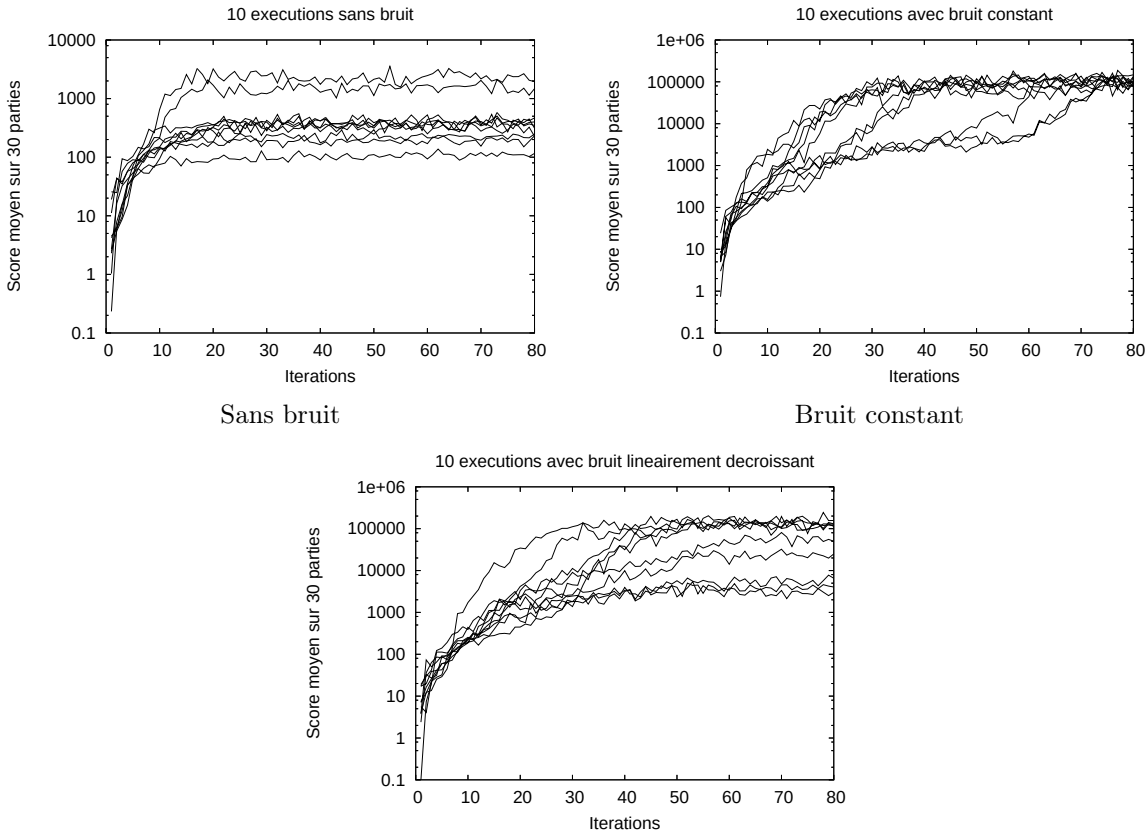


FIG. 3.1 – Notre reproduction de l'expérience de [17]. Chaque courbe est la moyenne de 10 exécutions (en échelle logarithmique) pour un type de bruit donné : nul, constant ou linéairement décroissant.

Chacune des trois courbes de la figure 3.1 représente la moyenne des 10 exécutions effectuées. Ces expériences confirment que l'utilisation du bruit permet d'améliorer sensiblement les performances. Toutefois, nos courbes n'atteignent pas les mêmes valeurs que celles de Szita et Lörincz [17], en particulier pour le cas non bruité et le bruit linéairement décroissant. Cela pourrait être dû à des différences d'implantation au niveau des règles du jeu et de la gestion de la fin de partie, et au fait qu'ils n'ont exécuté qu'une fois l'algorithme pour générer leurs courbes.

Le détail des 10 courbes de chaque expérience est donné figure 3.2. On constate que le bruit constant est plus fiable que le bruit linéaire, car avec du bruit linéaire, les valeurs atteintes sont très variables selon les exécutions : de 5000 à 250 000 lignes. Avec du bruit constant au contraire, toutes les exécutions atteignent des performances équivalentes.



Sans bruit

Bruit constant

Bruit linéairement décroissant

FIG. 3.2 – Détail des 10 courbes de chaque expérience de la figure 3.1. **Sans bruit :** la courbe d'apprentissage se stabilise vers l'itération 20, à une valeur qui varie beaucoup selon les exécutions (de 100 à 3000 lignes). **Bruit constant :** les 10 exécutions atteignent toutes des performances équivalentes, de l'ordre de 100 000 lignes, même si elles les atteignent plus ou moins rapidement. **Bruit linéairement décroissant :** les performances se stabilisent vers l'itération 50 (lorsque le bruit devient nul), mais elles atteignent des valeurs très variables, de 5000 à 250 000 lignes, et en moyenne, les résultats sont moins bons qu'avec le bruit constant.

3.3.1 Etude de l'influence des paramètres

La méthode d'entropie croisée bruitée appliquée à Tetris comporte plusieurs paramètres : le nombre de vecteurs de poids générés à chaque itération ($N = 100$), le nombre de parties jouées pour évaluer chaque vecteur généré ($k = 1$), la proportion de vecteurs sélectionnés à chaque itération ($\rho = 0.1$) et le type de bruit. Nous avons essayé de changer ces paramètres. En particulier, nous voulions augmenter k afin de sélectionner les vecteurs de poids avec plus de précision. En effet, plusieurs parties jouées avec une même stratégie donnent des scores très variables [9] donc il semblait intéressant d'évaluer les vecteurs de poids générés en jouant plus d'une partie. Nous avons aussi essayé différentes valeurs de N et de ρ. Nous avons pris les valeurs suivantes pour chaque paramètre : $N \in \{60, 100, 200, 300\}$, $k \in \{1, 2, 5, 10\}$ et $\rho \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$. Enfin, nous avons essayé sans bruit et avec du bruit constant. Nous ne nous sommes pas intéressés au bruit linéaire qui donne des résultats trop aléatoires selon les exécutions (voir figure 3.2). L'algorithme a été exécuté 10 fois avec chaque combinaison de paramètres, sur un jeu de taille 10×10 pour que les parties durent moins longtemps, et en stoppant l'algorithme après 80 itérations. Nous souhaitions ainsi mettre en évidence l'influence de chaque paramètre et déterminer si un choix

optimal de paramètres se dégageait.

Influence du bruit

Sans bruit, l'algorithme converge trop rapidement (la variance tombe trop vite à zéro) et les scores atteints sont très inférieurs à ceux atteints avec du bruit. Nous avons observé cela dans toutes nos simulations, quelles que soient les valeurs des autres paramètres.

Influence de ρ

$\rho \in]0, 1[$ est la proportion des vecteurs sélectionnés à chaque itération. Sans bruit, nous venons de voir que l'algorithme convergeait trop vite pour atteindre de bonnes performances. Nous avons observé que les grandes valeurs de ρ donnent des meilleurs résultats. Cela s'explique par le fait qu'avec une grande valeur de ρ, le nombre de vecteurs sélectionnés est plus important, donc leur variance est plus élevée. Ainsi, la variance tombe moins vite à zéro et l'algorithme met plus de temps à converger.

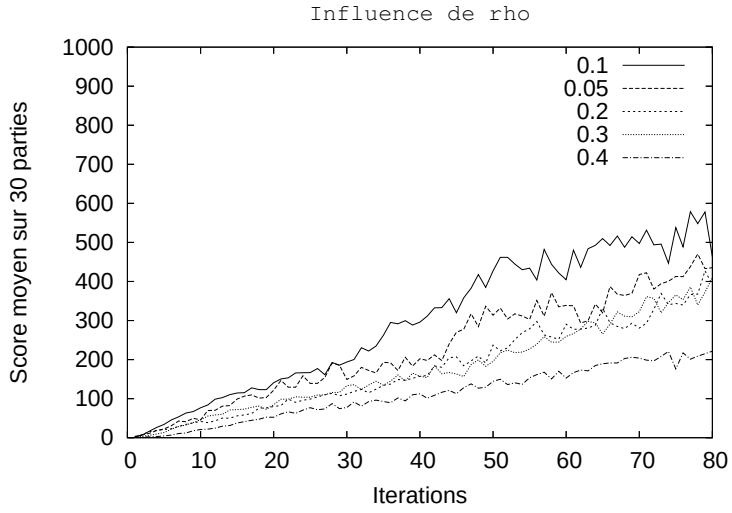


FIG. 3.3 – Performance moyenne de 10 exécutions en 10×10 avec $rho \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$, $k = 1$, $N = 100$ et un bruit constant. Le meilleur résultat est atteint pour $\rho = 0.1$.

Avec du bruit constant, la valeur proposée dans [17] ($\rho = 0.1$) semble donner les meilleurs résultats. La figure 3.3 représente la courbe d'apprentissage moyenne de 10 exécutions avec différentes valeurs de ρ, $k = 1$, $N = 100$ et un bruit constant. Le meilleur résultat est atteint pour $\rho = 0.1$. Pour les valeurs de N et k, $\rho = 0.1$ est la plupart du temps la meilleure valeur. Nos résultats confirment donc le choix de [17] de prendre $\rho = 0.1$.

Influence de N et k

Comme expliqué plus haut, nous souhaitions savoir si les résultats sont meilleurs en changeant le nombre de vecteurs générés (N) et le nombre de parties jouées pour évaluer un vecteur (k). Augmenter N et k rend le processus de sélection plus efficace, puisque l'on génère plus de vecteurs et on les sélectionne de manière plus précise. Les vecteurs sélectionnés sont donc meilleurs en général. Comme nous nous y attendions, de bonnes performances sont atteintes en moins d'itérations lorsque N et k sont élevés (figure 3.4, à gauche). Cependant, le temps d'exécution d'une itération augmente lui aussi. Avec $N = 100$ et $k = 1$, 100 parties sont jouées à chaque itération, alors qu'avec $N = 300$ et $k = 10$, 300 parties sont jouées. Avec des valeurs élevées de N et k, le nombre d'itérations nécessaires pour converger est plus faible, mais chaque itération prend donc plus de temps.

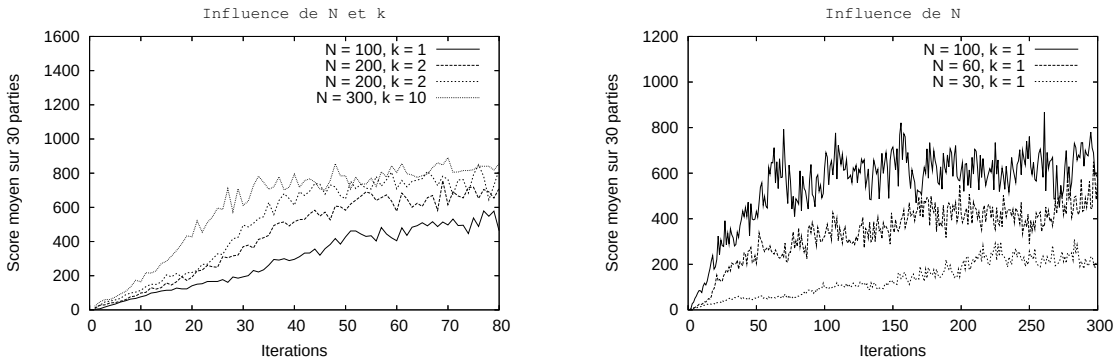


FIG. 3.4 – Gauche : 10 exécutions avec with différentes valeurs de N et k sur 80 itérations, $\rho = 0.1$ et un bruit constant. Droite : 10 exécutions avec $N \in \{30, 60, 100\}$, $k = 1$, $\rho = 0.1$ et un bruit constant sur 300 itérations.

Le problème est donc de savoir si les performances finalement atteintes avec des petites valeurs de N et k sont inférieures ou non à celles atteintes avec des grandes valeurs. Nos expériences ont montré que pour les grandes valeurs de N, et dans une moindre mesure de k, en plus de converger en moins d'itérations, l'algorithme converge vers de meilleurs scores (figure 3.4, droite). Nous en concluons qu'il est utile d'augmenter N et k car de meilleurs scores sont atteints. L'augmentation du temps d'exécution de chaque itération est compensée par la diminution du nombre d'itérations nécessaires.

3.3.2 Notre meilleure heuristique : 40 millions de lignes

Nous venons d'étudier les différents paramètres de l'algorithme de Szita and Lörincz [17] afin de montrer leur influence. Parallèlement à cette étude, nous voulions également montrer l'influence du choix des critères. Nous avons essayé un nouveau jeu de critères, composé des critères de Bertsekas et Ioffe (figure 2.3), des critères de Dellacherie (figure 1.4), et d'un nouveau critère que nous proposons. Ce nouveau critère est la somme des profondeurs de trous, c'est-à-dire le nombre de cellules pleines pour lesquelles il existe au moins une cellule vide en-dessous dans la même colonne. Le but de ce critère est d'empêcher de poser une pièce au-dessus d'un trou, afin de pouvoir supprimer les trous plus facilement.

Nous avons lancé la méthode d'entropie croisée en taille 10×20 avec les paramètres d'origine de Szita et Lörincz [17] : $N = 100$, $k = 1$, $\rho = 0.1$ et un bruit linéairement décroissant. La figure 3.5 représente la courbe d'apprentissage obtenue au bout de trois semaines d'exécution, en comparaison avec les trois courbes correspondant à notre implantation de l'expérience de Szita et Lörincz. Les meilleures performances sont atteintes à l'itération 32 avec un score moyen de 40 441 752 lignes sur 30 parties jouées. La partie la plus longue a été réalisée à l'itération 26, avec 160 278 005 lignes. Ainsi, la fonction heuristique que nous avons obtenue fait mieux que l'algorithme qui était jusqu'ici le meilleur à notre connaissance : celui de Dellacherie [7], qui réalisait 1.8 million de lignes en moyenne sur notre simulateur. La figure 3.6 donne l'ensemble des poids correspondant à notre heuristique. Notons cependant que notre heuristique se base sur celle de Dellacherie.

A la suite des conclusions de notre étude sur l'influence du bruit et des autres paramètres (voir la section précédente), nous avons ensuite lancé d'autres exécutions avec ce jeu de critères, en utilisant des valeurs différentes de N et k, et surtout du bruit constant qui est plus fiable que le bruit linéaire. Jusqu'à présent, ces exécutions atteignent jusqu'à 20 millions de lignes en moyenne mais n'ont pas encore convergé.

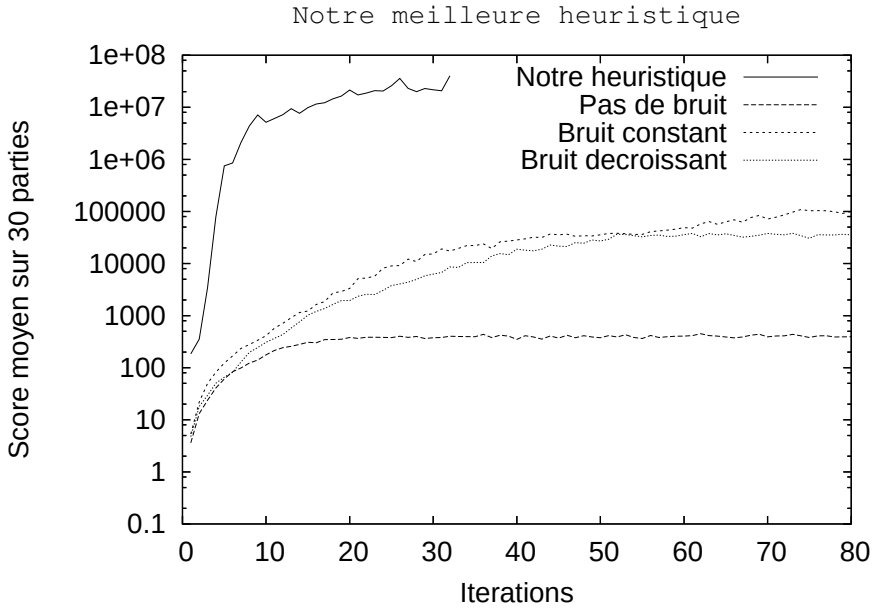


FIG. 3.5 – Exécution avec notre jeu de critères, en comparaison avec les courbes de notre implantation de l'expérience de Szita et Lörincz [17].

Critère	**Symbole**	**Poids**	**Critère**	**Symbole**	**Poids**
Column Height	h_1	-1.15	Maximum height	H	1.00
	h_2	-4.29	Holes	L	-58.29
	h_3	-2.74	Landing height	l	-35.53
	h_4	0.70	Eroded Piece Cells	e	7.45
	h_5	-2.73	Row transitions	Δr	-21.82
	h_6	-2.90	Column transitions	Δc	-61.31
	h_7	1.21	Board wells	W	20.25
	h_8	0.24	Holes depth (profondeur des trous)	D	-5.93
	h_9	-2.42			
	h_{10}	-2.74			
Column Difference	Δh_1	-4.71			
	Δh_2	-3.41			
	Δh_3	-12.15			
	Δh_4	-0.89			
	Δh_5	-10.44			
	Δh_6	-3.34			
	Δh_7	-7.49			
	Δh_8	-2.49			
	Δh_9	-6.10			

FIG. 3.6 – Les poids constituant notre meilleure heuristique obtenue. Score moyen sur 30 parties jouées : 40 441 752 lignes. Voir les figures 1.4 et 2.3 pour la définition de chaque critère.

Conclusion

Au cours de ce travail, nous avons étudié des techniques d'apprentissage appliquées à Tetris, qui est un problème complexe nécessitant l'utilisation de techniques approximatives. Nous avons implanté un simulateur performant pour pouvoir reproduire certaines de ces approches et proposer des améliorations. Le meilleur algorithme à notre connaissance, écrit par Pierre Dellacherie [7], réalise sur notre simulateur 1.8 million de lignes, sans pourtant utiliser de techniques d'aprentissage.

Nous nous sommes intéressés à une approche de contrôle optimal stochastique, l'algorithme λ-Policy Iteration [2]. Cet algorithme est particulièrement intéressant car il unifie deux techniques classiques du contrôle optimal stochastique : l'Itération de la Valeur et l'Itération de la Politique. De plus, une version approximative lui permet d'être adapté aux problèmes de grande taille. Les courbes d'apprentissage obtenues par les auteurs sur le problème de Tetris [2] chutaient de manière inexpliquée après quelques itérations. Nous n'avons pas observé ce phénomène et nous proposons aujourd'hui une explication à cette baisse, qui serait dûe à une erreur d'implantation.

Nous nous sommes également intéressés à une technique d'optimisation globale, la méthode d'entropie croisée [17], car cette approche donnait pour le problème de Tetris des résultats supérieurs aux techniques de contrôle optimal stochastique. Nos expériences confirment ces bons résultats, et en modifiant la fonction d'évaluation pour y ajouter les fonctions de base (ou critères) de Dellacherie [7] et un critère original, nous avons atteint un score moyen de 40 millions de lignes, établissant ainsi à notre connaissance la meilleure heuristique pour le problème de Tetris.

A l'heure actuelle, la méthode d'entropie croisée donne pour le problème de Tetris des résultats supérieurs aux techniques de contrôle optimal stochastique. Ces deux approches se basent sur une fonction d'évaluation sous forme de combinaison linéaire de critères, mais dans le cadre du contrôle optimal stochastique, cette fonction d'évaluation est la fonction de valeur et elle représente la moyenne du score futur. Dans le cas de Tetris, il semble donc qu'estimer la fonction de valeur par des méthodes de contrôle optimal ne permet pas d'explorer un champ de politiques aussi intéressant qu'avec des techniques de recherche directe d'heuristiques telles que la méthode d'entropie croisée.

Nous concluons également de ce travail l'importance du choix des critères utilisés pour construire une fonction d'évaluation heuristique. En effet, si l'algorithme de Dellacherie donnait de meilleurs résultats à Tetris que les techniques d'apprentissage, c'était grâce à un choix pertinent de critères, et c'est aussi en se basant sur ces critères efficaces que nous avons obtenu des scores inégalés à notre connaissance.

Nos résultats sur l'algorithme λ-Policy Iteration et la méthode l'entropie croisée appliqués à Tetris ont donné lieu à la soumission d'un article à la conférence d'apprentissage automatique NIPS'07 (Neural Information Processing Systems).

Perspectives

Les résultats que nous avons obtenus ouvrent de nombreuses pistes à explorer. D'abord, nous venons de mentionner l'importance d'un choix de critères pertinent : une piste possible serait par exemple d'établir une méthode qui sélectionnerait de manière automatique les critères les plus pertinents pour

évaluer un état.

Avec le contrôle optimal stochastique, nous avons résolu le problème de Tetris de manière exacte sur des zones de jeu de taille réduites (4×5, 5×5). Une question se pose alors : comment exploiter ces résultats exacts pour le Tetris standard (10×20) ? Une idée possible serait d'utiliser dans l'espace de taille 10×20 une fenêtre de taille 5×5 qui pourrait parcourir tout l'espace de jeu. On pourrait ainsi utiliser de manière locale l'information exacte donnée par la fonction de valeur optimale obtenue en 5×5.

Notre travail sur Tetris ouvre enfin des perspectives sur d'autres problèmes de grande taille où un agent doit apprendre ou faire de la planification. Des applications comme le contrôle d'un robot autonome dans un environnement incertain constituent des problèmes difficiles à mettre en oeuvre, et le contrôle optimal approché ou la méthode d'entropie croisée pourraient permettre d'aborder efficacement ces problèmes de grande taille.

Bibliographie

[1] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[2] D. Bertsekas and S. Ioffe. Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, MIT, 1996.

[3] D.P. Bertsekas and J.N. Tsitsiklis. *Neurodynamic Programming*. Athena Scientific, 1996.

[4] N. Böhm, G. Kókai, and S. Mandl. An Evolutionary Approach to Tetris. In *The Sixth Metaheuristics International Conference (MIC2005)*, 2005.

[5] H. Burgiel. How to lose at tetris. *Mathematical Gazette*, page 194, 1997.

[6] P. de Boer, D. Kroese, S. Mannor, and R. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 1(134) :19–67, 2004.

[7] P. Dellacherie and C. P. Fahey. Best one-piece Tetris-playing algorithm in the world, 2003. `http://colinfahey.com/tetris/tetris.html`.

[8] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *Proc. 9th International Computing and Combinatorics Conference (COCOON 2003)*, pages 351–363, 2003.

[9] C. P. Fahey. Tetris AI, Computer plays Tetris, 2003. `http://colinfahey.com/tetris/tetris.html`.

[10] V. Farias and B. van Roy. *Tetris : A study of randomized constraint sampling*. Springer-Verlag, 2006.

[11] S. Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems (NIPS 14)*, pages 1531–1538, 2001.

[12] M. G. Lagoudakis, R. Parr, and M. L. Littman. Least-squares methods in reinforcement learning for control. In *SETN '02 : Proceedings of the Second Hellenic Conference on AI*, pages 249–260, London, UK, 2002. Springer-Verlag.

[13] R. E. Llima. Xtris readme, 2005.

[14] M. Puterman. *Markov Decision Processes*. Wiley, New York, 1994.

[15] J. Ramon and K. Driessens. On the numeric stability of gaussian processes regression for relational reinforcement learning. In *ICML-2004 Workshop on Relational Reinforcement Learning*, pages 10–14, 2004.

[16] R.S. Sutton and A.G. Barto. *Reinforcement Learning, An introduction*. BradFord Book. The MIT Press, 1998.

[17] I. Szita and A. Lörincz. Learning Tetris Using the Noisy Cross-Entropy Method. *Neural Computation*, 18(12) :2936–2941, 2006.

Annexe A

Liste des critères

Une décision à Tetris est évaluée comme une somme pondérée de différents critères tels que la hauteur du mur ou le nombre de trous. Nous avons vu que le choix de ces critères était primordial pour les performances d'un algorithme. On répertorie ici les critères, très variés, utilisés dans les différentes approches.

Sans apprentissage

Dans cette section, on s'intéresse aux algorithmes qui n'utilisent pas de techniques d'apprentissage automatique.

Pierre Dellacherie (2003) : le record [7]

Auteur(s) : Pierre Dellacherie
Date : 2003
Type d'algorithme : Poids fixés à la main
Moyenne : 631 167 lignes (Prochaine pièce connue : Non)

Critère	Poids	Explications	Commentaires	Vu chez
Landing height	-1	Hauteur de la dernière pièce posée	Pénalise le fait d'empiler des pièces en hauteur	[4, 13]
Eroded Piece Cells	1	(Nombre de lignes éliminées au dernier coup) * (Nombre de cellules éliminées dans la dernière pièce posée)	Incite à faire des lignes et maximise la contribution de la pièce	
Row transitions	-1	Nombre de transitions vide-plein ou plein-vide entre les cases de chaque ligne	Incite le mur à être homogène	[4]
Column transitions	-1	Même chose sur les colonnes		
Buried Holes	-4	Nombre de trous (cases vides pour lesquelles il existe au moins une case pleine dans la colonne au-dessus d'elle)	Pénalise les trous	Tous
Board wells	-1	$\sum_{p \in puits}(1 + 2 + \cdots + profondeur(p))$	Pénalise les puits[1] profonds car ils obligent à attendre une grande barre verticale	

[1] Un puits est une succession de cellules vides et non recouvertes dans la même colonne et telles quel leurs cases voisines à gauche et à droite soient toutes les deux pleines. Les puits profonds sont pénalisants car ils obligent à attendre une barre verticale.

Colin P. Fahey (2003) : une pièce en avance [9]

Contrairement à la plupart des algorithmes de cette étude, celui de Colin P. Fahey connaît en avance la prochaine pièce qui va tomber. Grâce à cette information supplémentaire, les résultats de cet algorithme sont nettement meilleurs que s'il ne connaissait que la pièce courante.

Auteur(s) : Colin P. Fahey
Date : 2003
Type d'algorithme : Poids fixés à la main
Moyenne : 2 000 000 lignes (Prochaine pièce connue : Oui)

Critère	Poids	Explications	Commentaires	Vu chez
Shadowed Holes	-0.65	Nombre de trous	Pénalise les trous	Tous
Pile height weighted cells	-0.10	Cellules occupées pondérées par leur hauteur : $\sum_{c \in cellules} hauteur(c)$	Pénalise les cellules hautes	[4]
Sum of well heights	-0.20	Somme des profondeurs des puits : $\sum_{p \in puits} profondeur(p)$	Pénalise les puits[1] profonds car ils obligent à attendre une grande barre verticale	[4]
Rows eliminated	0.30	Nombre de lignes éliminées au dernier coup	Incite à faire des lignes	[12, 4]
Total occupied cells	0	Nombre de cellules occupées	Critère inutilisé (le poids est à zéro)	[4]

Algorithmes avec apprentissage

On s'intéresse maintenant aux techniques utilisant des techniques d'apprentissage pour trouver les meilleurs poids : contrôle optimal ou algorithmes génétiques.

Böhm, Kokai et Mandl : algorithmes génétiques avec une pièce en avance [4]

Auteur(s) : Niko Böhm, Gabriella Kokai, Stefan Mandl
Date : 2004
Type d'algorithme : Utilisation d'algorithmes génétiques pour déterminer les poids
Prochaine pièce connue : Oui
Moyenne : 586 103 lignes

Critère	Explications	Commentaires	Vu chez
Pile height	Hauteur de la plus haute cellule	Pénalise la hauteur du mur	[13, 2, 12, 15]
Holes	Nombre de trous	Pénalise les trous	Tous
Holes 2	Même chose, mais des cases vides adjacentes dans une colonne comptent pour un seul trou		
Removed Lines	Lignes supprimées au dernier coup	Incite à faire des lignes	[12, 9]
Altitude Difference	Différence maximale de hauteur entre 2 colonnes	Donne une hauteur homogène	
Maximum Well Depth	Profondeur du puits le plus profond	Pénalise les puits profonds	
Sum of all wells	Somme des profondeurs des puits	Pénalise les puits profonds	[9]
Landing height	Hauteur de la dernière pièce posée	Pénalise le fait d'empiler	[7, 13]
Blocks	Nombre de cellules occupées	Incite à réduire le mur	[9]
Weighted Blocks	Cellules pondérées par leur hauteur	Pénalise les cellules hautes	[9]
Row transitions	Transitions entre les cases sur les lignes	Pénalise les damiers	[7]
Column transitions	Même chose sur les colonnes		

Bercot-Blondeel : xtris [13]

Cet algorithme est implanté dans xtris, une version de Tetris pour X-Window, développée par Roger Espel Llima. Il réalise en moyenne 50 000 lignes et les poids ont été appris à l'aide d'un algorithme génétique.

Auteur(s) : Laurent Bercot, Sébastien Blondeel
Date : 1996
Type d'algorithme : Utilisation d'algorithmes génétiques pour déterminer les poids
Moyenne : 50 000 lignes (Prochaine pièce connue : Non)

Critère	Poids	Explications	Commentaires	Vu chez
Height	110	Hauteur de la plus haute cellule	Pénalise la hauteur du mur	[2, 12, 4, 15]
Holes	450	Nombre de trous	Pénalise les trous	Tous
Frontier	260	Nombre de transitions entre une cellule vide une cellule pleine ou l'inverse, en ligne et en colonne	Incite à arranger les briques de manière homogène	
Drop	290	Hauteur de la dernière pièce posée	Pénalise le fait d'empiler des pièces en hauteur	[7, 4]
Pit	190	Somme des profondeurs des puits de profondeur supérieure ou égale à 2	Pénalise les puits profonds car ils obligent à attendre une barre verticale	
Ehole	80	Somme pondérée des trous, indiquant s'ils sont difficiles à remplir	Pénalise les trous difficiles à remplir	

Bertsekas [2], Kakade [11], Farias [10], Szita [17]

Bertsekas et Ioffe ont défini ce jeu de critères en 1996 pour illustrer l'algorithme λ-Policy Iteration, comme expliqué dans la section 2.4.2. Ces critères ont ensuite été réutilisés par trois autres approches d'apprentissage :

- Natural policy gradient [11] : moyenne de 6800 lignes
- Programmation linéaire + Bootstrap [10] : moyenne de 4274 lignes
- Méthode d'entropie croisée [17] : moyenne de 348 895 lignes

Auteur(s) : Dimitri P. Bertsekas et Sergey Ioffe
Date : 1996
Type d'algorithme : λ-Policy Iteration
Prochaine pièce connue : Non
Moyenne : 3183 lignes

Critère	Explications	Commentaires	Vu chez
Hauteur des colonnes	Hauteur de chaque colonne du mur (il y a donc 10 valeurs)	Pénalise le fait d'empiler en hauteur	[15]
Différences de hauteur	Différence de hauteur entre chaque colonne et sa voisine (il y a donc 9 valeurs)	Pénalise l'hétérogénéité des colonnes	[15]
Hauteur maximale	Hauteur de la colonne la plus haute du mur	Pénalise le fait d'empiler en hauteur	[13, 12, 4, 15]
Trous	Nombre de trous	Pénalise les trous	Tous

Lagoudakis (2002) : moindres carrés [12]

Auteur(s) : Michail G. Lagoudakis, Ronald Parr, Michal L. Littman
Date : 2002
Type d'algorithme : Least-Squares Policy Iteration
Prochaine pièce connue : Non
Moyenne : 1000 à 3000 lignes

Critère	Explications	Commentaires	Vu chez
Hauteur maximale	Hauteur de la plus haute cellule du mur	Pénalise le fait d'empiler en hauteur	[2, 13, 4, 15]
Trous	Nombre de trous	Pénalise les trous	Tous
Différence totale de hauteur	Somme des différences de hauteur entre chaque colonne et sa voisine	Pénalise l'hétérogénéité des colonnes	
Hauteur moyenne	Hauteur moyenne des colonnes	Pénalise le fait d'empiler en hauteur	[15]
+ Variation de chacun de ces quatre critères par rapport au coup précédent			
Variation de score	Nombre de lignes réalisées au dernier coup	Incite à faire des lignes	[9, 4]

Driessens (2004) : Relationnal Reinforcement Learning [15]

Auteur(s) : 2004
Date : Jan Ramon et Kurt Driessens
Type d'algorithme : Relationel Reinforcement Learning - KBR
Prochaine pièce connue : Non
Moyenne : 50 lignes

Critère	Explications	Commentaires	Vu chez
Hauteur des colonnes	Hauteur de chaque colonne du mur (il y a donc 10 valeurs)	Pénalise le fait d'empiler en hauteur	[2]
Hauteur maximale	Hauteur de la plus haute colonne du mur		[2, 13, 12, 4]
Hauteur moyenne	Hauteur moyenne des colonnes		[12]
Hauteur minimale	Hauteur de la colonne la moins haute		
Ecart maximal	(Hauteur maximale) - (Hauteur moyenne)	Pénalise l'hétérogénéité des colonnes	
Ecart minimal	(Hauteur moyenne) - (Hauteur minimale)		
Différences de hauteur	Différence de hauteur entre chaque colonne et sa voisine (il y a donc 9 valeurs)		[2]
Trous	Nombre de trous	Pénalise les trous	Tous
Puits	Nombre de puits	Pénalise les puits	
Profondeur des trous	Profondeur moyenne où les trous sont enterrés	Evite de trop recouvrir les trous	

Annexe B

Expériences avec λ-Policy Iteration

Dans cette section, on donne l'ensemble des courbes obtenues au cours de nos simulations avec l'algorithme λ-Policy Iteration. On donne d'abord les courbes correspondant à notre implantation non modifiée, puis les courbes de la version dans laquelle nous avons introduit une erreur d'implantation pour tenter de reproduire le "phénomène intriguant" observé par Bertsekas et Ioffe dans leur expérience d'origine [2] (voir la section 2.4.2). Les courbes de gauche représentent les résultats sur une zone de jeu de taille 10×10, et celles de droite correspondent à la taille standard 10×20.

Notre implantation

Les figures B.1 à B.4 représentent pour chaque valeur de λ les courbes des 10 exécutions que nous avons obtenues avec notre implantation de λ-Policy Iteration. La figure B.5 correspond à la moyenne des 10 exécutions pour toutes les valeurs de λ.

Reproduction des courbes d'origine

On donne ensuite les courbes que nous avons obtenues en introduisant une erreur d'implantation (voir section 2.4.2) de manière à obtenir des courbes ayant une allure similaire à celles de Bertsekas et Ioffe [2], où les performances chutent après quelques itérations. Les figures B.6 à B.9 représentent pour chaque valeur de λ les courbes des 10 exécutions, et la figure B.10 correspond à la moyenne des 10 exécutions pour toutes les valeurs de λ.

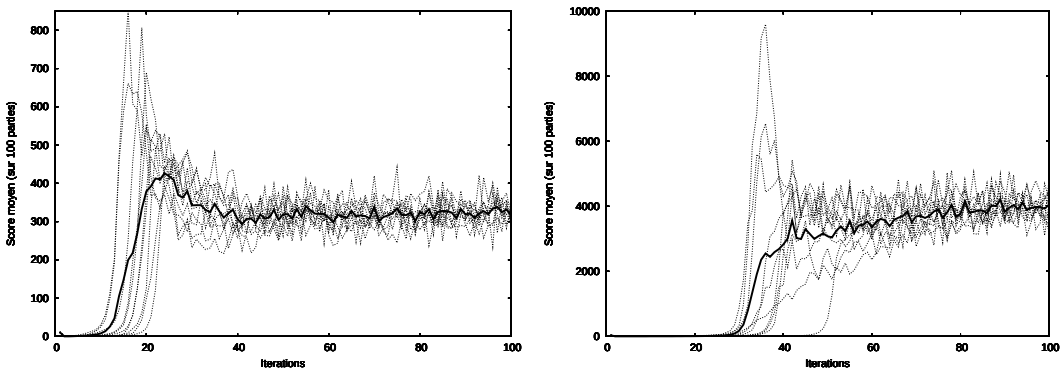


FIG. B.1 – 10 exécutions avec notre version, $\lambda = 0.0$. Gauche : taille 10×10. Droite : taille 10×20.

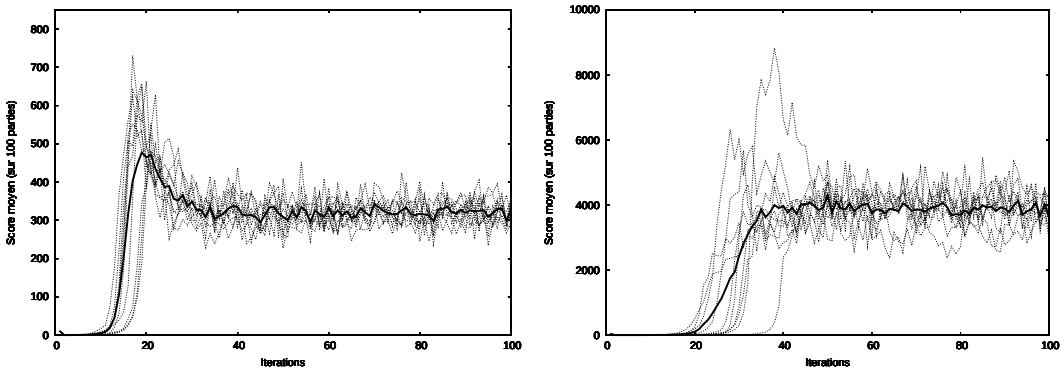


FIG. B.2 – 10 exécutions avec notre version, $\lambda = 0.3$. Gauche : taille 10×10. Droite : taille 10×20.

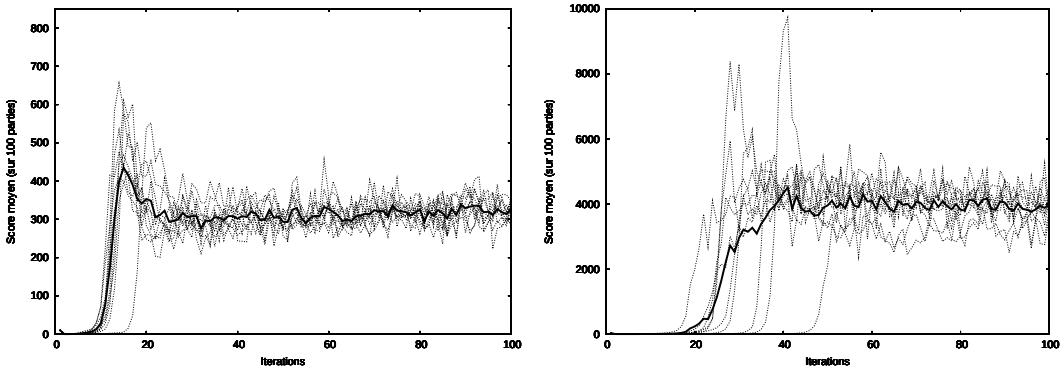


FIG. B.3 – 10 exécutions avec notre version, $\lambda = 0.5$. Gauche : taille 10×10. Droite : taille 10×20.

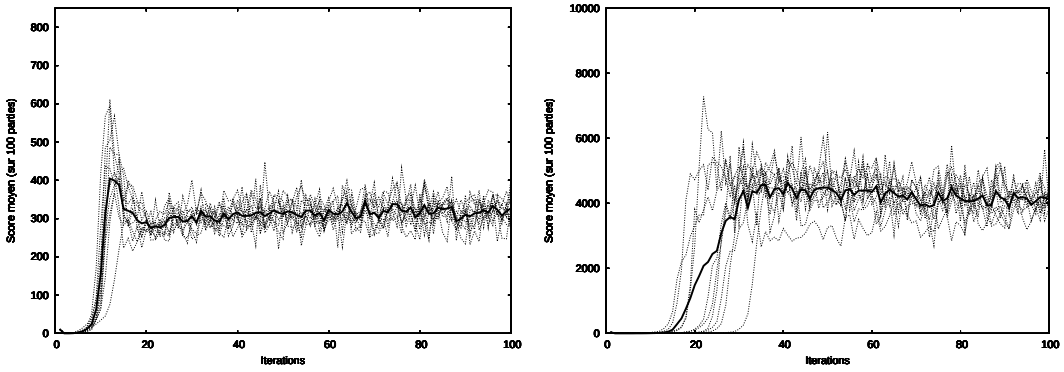


FIG. B.4 – 10 exécutions avec notre version, $\lambda = 0.7$. Gauche : taille 10×10. Droite : taille 10×20.

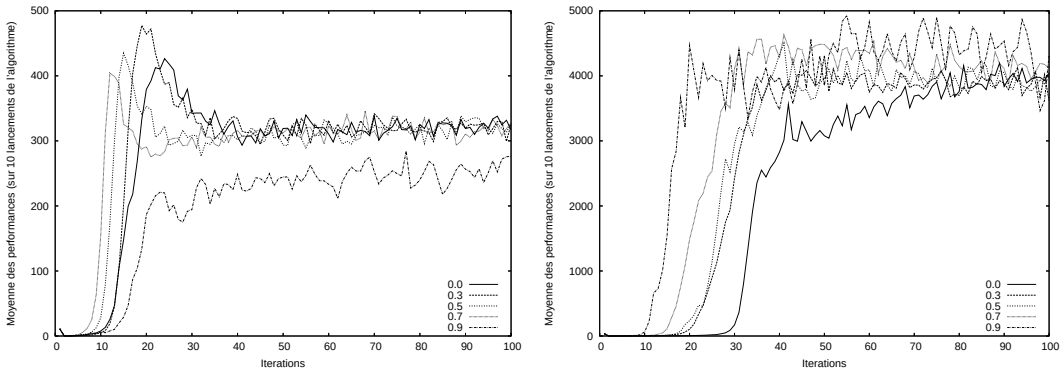


FIG. B.5 – Moyenne de 10 exécutions notre version, pour différentes valeurs de λ. Gauche : taille 10×10. Droite : taille 10×20.

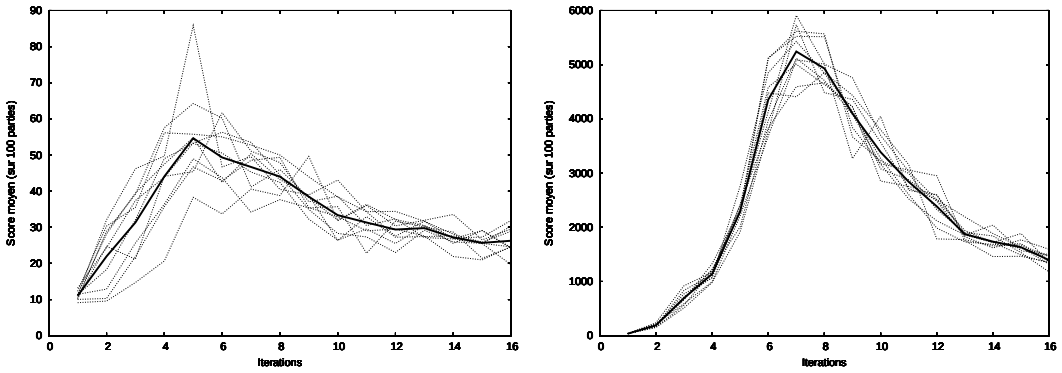


FIG. B.6 – 10 exécutions avec la version modifiée, $\lambda = 0.0$. Gauche : taille 10×10. Droite : taille 10×20.

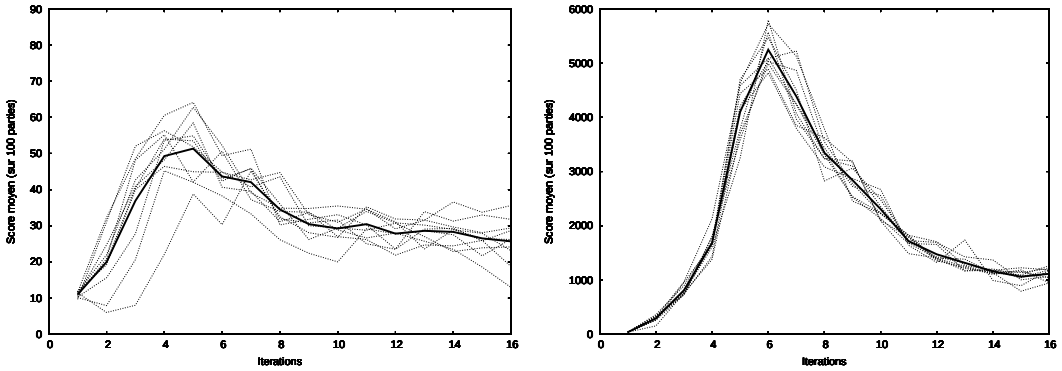


FIG. B.7 – 10 exécutions avec la version modifiée, $\lambda = 0.3$. Gauche : taille 10×10. Droite : taille 10×20.

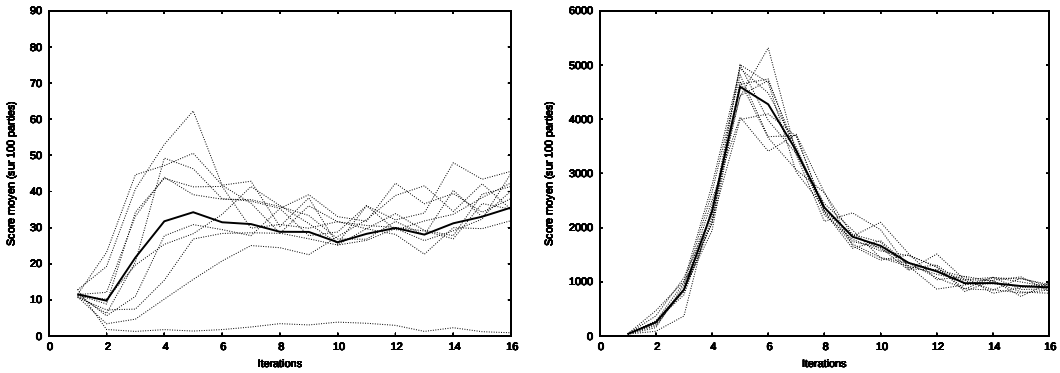


FIG. B.8 – 10 exécutions avec la version modifiée, $\lambda = 0.5$. Gauche : taille 10×10. Droite : taille 10×20.

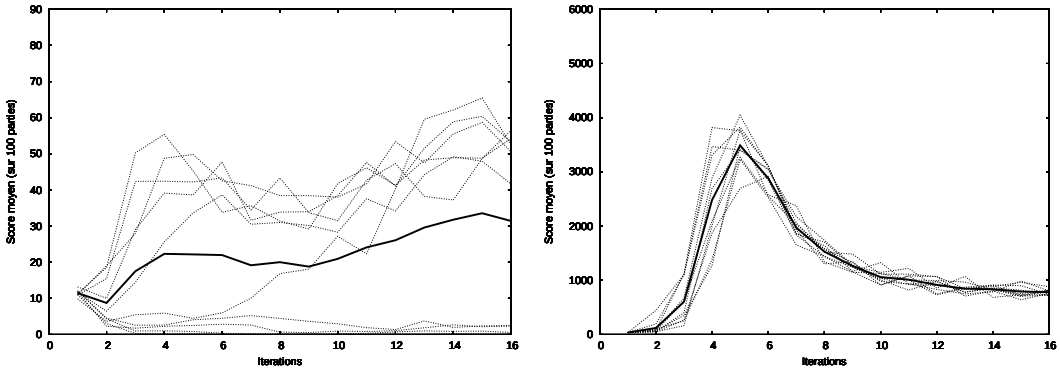


FIG. B.9 – 10 exécutions avec la version modifiée, $\lambda = 0.7$. Gauche : taille 10×10. Droite : taille 10×20.

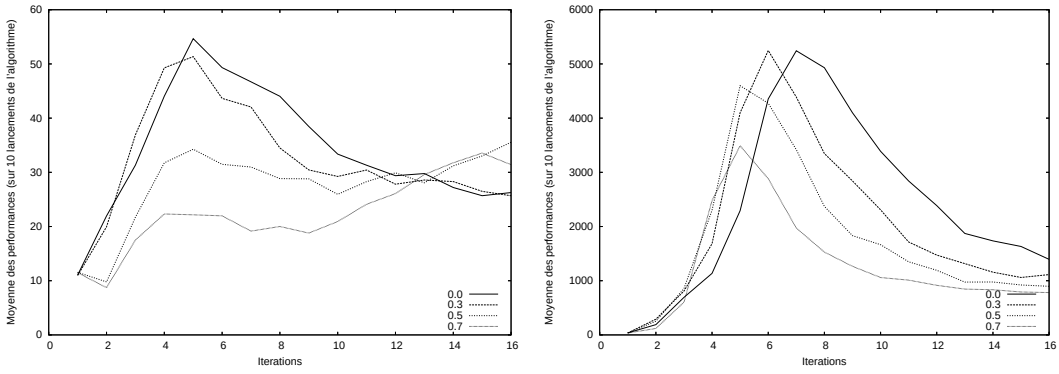


FIG. B.10 – Moyenne de 10 exécutions avec la version modifiée, pour différentes valeurs de λ. Gauche : taille 10×10. Droite : taille 10×20.