Projet: Q-learning appliqué au jeu de Tetris

GUILLAUME ARRUDA 1635805
RAPHAEL LAPIERRE 1644671
École polytechnique de Montréal

Dans le cadre du cours
INF8702 - Infographie avancée

21 Avril 2016

Table des matières

1 Introduction

Dans le cadre du cours de techniques d'intelligence artificielle probabilistiques, nous avons décidé d'implémenter un agent intelligent pouvant jouer à Tetris et ce, en utilisant la technique de *Q-learning*. Pour se faire, nous avons utilisé une implémentation *open source* de Tetris écrite en Python. L'agent intelligent a aussi été réalisé dans ce langage de programmation. Le présent rapport décrira, dans les sections suivantes, qu'est-ce que le jeu de Tétris, une revue de la littérature sur le sujet, la théorie appliquée à notre agent intelligent, les résultats obtenues ainsi que nos expériences et finalement une analyse critique de l'approche utilisée pour règler le problème.

2 Revue de littérature

Avant de plonger dans la revue de littérature, voici une courte introduction sur la version du jeu de Tetris utilisée pour ce travail

2.1 Tetris

Le but du jeu de tetris est de remplir une grille de grandeur 10 par 22 à l'aide de pièces arrivant aléatoirement sur la grille de jeu. Il y a sept sortes de pieces différentes toutes formées de 4 blocs. Chaque bloc occupe un espace de 1 par 1 sur la grille. De plus, il est aussi possible de connaître, en plus de la pièce courante, la pièce suivante. Cela permet de mieux planifier son jeu. Lorsqu'une ligne de largeur 10 est complétée, elle est enlevé de l'espace de jeu. Le jeu est perdu lorsqu'une pièce atteint une hauteur plus grande que 22.

2.2 Revue de littérature

Plusieurs travaux ont été réalisé dans le but d'écrire un agent intelligent pouvant jouer à Tetris de manière efficace. Le premier article que nous avons consulté a été utile pour nous confimer que Tetris est bel et bien un jeu complexe. En effet selon [1], il est montré que le problème de maximiser le nombre de ligne, le nombre de tetris (4 lignes en même temps), minimizer la hauteur maximale de l'espace de jeu est un problème NP-complet.

Par la suite, nous avons chercher des articles de recherches montrant qu'il est bel et bien possible d'appliquer le *Q-learning* au jeu de Tetris. Zucker et Maas [5] ont montré que la technique est applicable. Par contre, ils ont du utiliser une technique de *features based Q-learning* plutôt que de représenter l'état total du jeu. En effet, sur une grille de 10 par 22, il y a 220 cases qui peuvent être soit occupé ou non. On se retrouve donc avec 2^{220} configurations possibles ce qui est définitivement trop grand pour être représenté convenablement en mémoire.

Finalement, [3] et [2] ont confirmé que les résultats obtenus en tentant de représenter complètement l'espace d'état sont très mauvais. Dans le cas de [3], les auteurs ont dû eux aussi représenter le jeu par un ensemble d'attributs tandis que dans le cas de [2], l'expérience a été un échec même en tentant de représenter le jeu par un ensemble d'attributs.

3 Approche théorique

3.1 Processus de décision Markovien

Il est important d'abord dans la section théorique de comprendre que l'algorithme de *Q-learning* permet, dans les bonnes conditions, d'obtenir la politique optimale permettant de résoudre un processus de décision Markovien. La politique optimale d'un tel système permet de connaître l'action permettant de maximiser la victoire[4].

3.2 Algorithme de *Q-learning* simple

La logique du *Q-learning* de base est assez simple. Il s'agit d'apprendre quoi faire dans les situations possibles en ajustant nos décisions futures basé sur une récompense obtenues par des décisions passées. On doit d'abord initialiser un vecteur contenant des poids $Q(s, a)$ ou s représente un état et a une action. En multipliant ces poids par notre vecteur d'état on trouve l'action qui maximise notre récompense et il s'agit de l'action choisi. L'ajustement des poids se fait ainsi :

$$\Delta Q(S_t, A_t) = \alpha(R_{t+1} + \gamma max Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{1}$$

Où R représente une fonction de récompense et α un taux d'apprentissage. Par contre, dans notre cas, l'espace d'état étant trop grand on représente l'état S par un sous-ensemble de caractéristique. L'algorithme de *Q-learning* ne change pas. Il est en effet invariant de la représentation de notre état.

3.3 *Q-learning* appliqué à Tetris

3.3.1 Caractéristiques

Dans le but de représenter notre jeu nous avons eu recours à un sous-ensemble de caractéristiques. Les voici :

- La hauteur de chaques colonnes (10 caractéristiques)
- La différence de hauteur entre les colonnes adjacentes (9 caractéristiques)
- La hauteur totale

- Le nombre de trous
- Une valeur de récompense

La valeur de récompense est donné dans notre cas par l'équation suivante :

$$R(l) = 2l - 1 \tag{2}$$

Dans cette équation l représente le nombre de lignes complétés par une action.

3.3.2 Modification de l'algorithem

Pour effectuer le travail nous nous sommes beaucoup inspiré de [5]. Les équations utilisés sont les suivantes. Les variations de nos poids θ sont calculées à l'aide de celles-ci

$$z_{t+1} = \beta z_t + \frac{\nabla q(\theta, x_{t+1}, u_{t+1})}{q(\theta, x_{t+1}, u_{t+1})} \tag{3}$$

$$\Delta_{t+1} = \Delta_t + \frac{t}{t+1}(r(x_{t+1}, u_{t+1})z_{t+1} - \Delta_t) \tag{4}$$

$$\theta \leftarrow \theta + \alpha\Delta \tag{5}$$

Finalement, à chaque fin de partie, à l'aide de la dernière équation, nos poids sont ajustés.

3.3.3 Action

L'action choisie par notre agent intelligent pour jouer à Tetris est celle qui maximise la multiplication du vecteur de représentation d'état par nos poids θ. Ainsi, pour chaque pièces à placer, toute les actions possibles sont envisagées et le vecteur d'état est remplie. Ceux-ci sont concaténés dans une matrice et le vecteur de poids pré multiplie cette matrice. Aussi, deux pièces sont considérés à la fois. La matrice de vecteurs d'états est donc remplie de toutes les combinaisons possibles de ces actions.

4 Expérience

Au cours de la réalisation de notre travail, plusieurs expérience ont été effectuées. Tout d'abord, le premier défi a été de trouver une fonction de récompense efficace. Dans la littérature, plusieurs personnes suggéraient d'utiliser le nombre de lignes complétées. Par contre, le problème avec une telle fonction est que la convergence est très longue. En effet, il est rare pour un agent intelligent initialisé de manière aléatoire de réussir à compléter des lignes. C'est pourquoi nous avons décidé de donner une récompense négative lorsqu'aucune lignes ne sont complétés2.

L'autre expérience qui a augmenté de beaucoup l'efficacité de notre agent intelligent était de considérer deux pièces à la fois plutôt qu'une. Cela a eu un effet immédiat sur les résultats obtenues.

Une autre expérience très importante a été l'ajustement du paramètre β. Ce paramètre controle l'importance des mouvements au fil de la partie. Plus le beta est près de 0, moins il garde une mémoire des mouvements ultérieurs. Un β de 1 accorde autant d'importances à tout les mouvements au cours de la partie. En faisant plusieurs expérience nous avons réalisé qu'il est mieux d'utiliser un β de valeur 1.

4.1 Résultats

Il est difficile de présenter une courbe d'apprentissage pour notre agent intelligent. En fait la courbe n'est pas intéressante car il apprend extrêmement rapidement. Voici un tableau des résultats obtenues.

Essai	Nombre de lignes complétées
1	0
2	412
3	22062

Il devient à partir de se niveau très long de tester jusqu'où l'agent intelligent peut continuer. Par contre, une chose intéressante est de le voir jouer au jeu. Ci-joint à ce rapport, un vidéo de l'agent intelligent qui commence avec des poids complètement aléatoires. On peut voir que dès la deuxième partie il est beaucoup plus performant. Le vidéo ne dure que 2 minutes dans le but de ne pas être trop long. Il est par contre clair que l'agent intelligent pourrait continuer à jouer très longtemps.

5 Analyse critique

L'approche utilisée pour apprendre le sujet choisi a été de d'abord comprendre la présentation de Rich Sutton [4]. Par la suite, nous avons cherché des articles qui expliquaient bien la méthode du *Q-learning* appliquée au jeu de Tetris. Puisqu'il s'agit d'un sujet assez simple, il n'a pas été trop difficile de trouver l'information nécéssaire à la réalisation de notre travail.

Références

[1] Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. erikdemaine.org/papers/Tetris_TR2002/paper.pdf, 2002.

[2] Michael Dunham and Andrew Alves. Tetris game-playing agents in python. http://www.cs.uml.edu/ecg/uploads/AIfall14/dunham_alves_tetris_game_playing.pdf, 2014.

[3] Alexander Gross, Jan Friedland, and Friedhelm Schwenker. Learning to play tetris by applying reinforcement learning methods. https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2008-118.pdf, 2008.

[4] Rich Sutton. Introduction to reinforcement learning with function approximation. https://webdocs.cs.ualberta.ca/~sutton/Talks/RLtutorialNIPS2015.pdf, 2015.

[5] Matt Zucker and Andrew Maas. Learning tetris. http://www.cs.cmu.edu/afs/cs/project/ACRL/www/TetrisReports/Zucker&Maas.pdf, 2009.