UNIVERSITÉ AIX-MARSEILLE

PROJET MATHÉMATIQUES ET INFORMATIQUE
DU CCIE 2ÈME ANNÉE

TÉTRIS BOT

IMPLÉMENTATION ET RÉSOLUTION AUTOMATIQUE DU JEU TÉTRIS PAR DIFFÉRENTES MÉTHODES

Frédéric Muller - Lionel Ponton

31 MAI 2019

Le code source LaTeX de ce rapport ainsi que l'intégralité du projet est disponible sur
`https://github.com/Abunux/tetrisbot`

Artificial Intelligence is no substitute for natural stupidity.

Woody Allen

TABLE DES MATIÈRES

INTRODUCTION

Ce projet a été réalisé entre novembre 2018 et mai 2019 dans le cadre du DU CCIE de l'université d'Aix-Marseille et, plus précisément, pour le module « Projet mathématiques et informatique » sous la direction de M. Tristan Colombo. Le but est d'implémenter des agents qui jouent de manière autonome au célèbre jeu Tétris et d'essayer de les optimiser afin qu'ils y jouent le mieux possible.

La réalisation du projet s'est essentiellement déroulée en trois phases.

La première phase a consisté à la réalisation complète du moteur de jeu. Plutôt que de partir d'une base existante ou de reprendre un code « tout fait », nous avons entièrement programmé « notre » version de Tétris en respectant (presque) toutes les règles officielles du jeu disponibles à l'adresse https://tetris.fandom.com/wiki/Tetris_Wiki. Nous avons choisi une conception orientée objet afin d'avoir un moteur offrant une grande adaptabilité pour les différents procédés d'optimisation.
Nous avons défini un mode *humain* qui permet de jouer en utilisant le clavier pour déplacer et retourner les pièces comme dans un jeu de Tétris usuel et qui a permis de tester le moteur. Nous avons également programmé, dès la phase de conception du moteur, plusieurs agents simples :

- un agent purement aléatoire qui joue totalement au hasard (et donc de façon catastrophique) ;
- un agent par filtrage qui choisit parmi les coups possibles celui qui est optimal selon différents critères possibles (créer le moins de trous possibles, créer le plus de lignes possibles, augmenter le moins possibles la hauteur des colonnes, etc.) ;
- un agent par évaluation de coups qui cherche à maximiser une fonction d'évaluation de qualité du coup dépendant de plusieurs variables pondérées par différents paramètres.

La deuxième phase à consister à implémenter une optimisation par algorithme génétique. Le but était de déterminer les « meilleurs » paramètres pour l'agent par évaluation de coups.
Dans cette phase, différentes voies ont été envisagées pour ce qui est du codage choisi, du mode de sélection des « parents », du mode de reproduction et de création de la nouvelle génération.
Les résultats obtenus ont été très satisfaisants, avec des agents parvenant à jouer des centaines de milliers de pièces d'affilée. Certaines limitations sont cependant apparues notamment en raison du caractère aléatoire de la fonction d'évaluation de coups.

La troisième et dernière phase a consisté à implémenter une optimisation à l'aide d'ap-

prentissage par renforcement (*reinforcement learning*). Une longue période de documentation et de maîtrise des concepts et des techniques a été nécessaire.
Nous avons choisi d'implémenter un apprentissage mettant en jeu un Q-Learning par itération de fonctions de valeur. Cette méthode demande idéalement de pouvoir stocker une matrice modélisant tous les états et toutes les actions possibles pour l'agent pour chaque état. Ceci s'est avéré impossible pour un jeu présentant autant de configurations possibles que Tétris. En conséquence, nous avons décidé d'une part d'implémenter sur un jeu de pièces réduit à un seul domino plutôt que sur le jeu de Tétris usuel à 7 pièces et ensuite nous avons limités la phase d'apprentissage à un échantillon aléatoire d'états plutôt que d'essayer de stocker en mémoire tous les états possibles.
Nous obtenons un agent qui fonctionne mais dans un cadre assez limité.
Pour contourner ce problème, il est nécessaire de faire appel aux toutes dernières techniques d'apprentissage profond (*Deep Learning*) utilisant notamment des réseaux de neurones. Ceci a été abordé sur le plan théorique et algorithmique mais sans être implémenté pour notre moteur.

Le plan de ce rapport reprend pour l'essentiel ces trois grandes phases.
Dans la première partie, après avoir retracé l'historique de la genèse et du développement du jeu Tétris, nous décrivons les règles suivies, l'implémentation du moteur de jeu et des premiers agents.
Dans la deuxième partie, nous présentons le fonctionnement général des algorithmes génétiques avec les différents choix possibles puis nous détaillons l'implémentation que nous avons mise en œuvre pour Tétris.
Enfin, la troisième partie est consacré au reinforcement learning. Après avoir rappelé les bases mathématiques concernant les processus de décision markoviens qui conduisent aux équations de Bellman, nous présentons le principe général de Q-Learning par itérations de fonctions de valeur et l'implémentation que nous en avons fait sur un jeu simplifié. Nous terminons cette partie par une ouverture sur le deep-Q-Learning en détaillant, du point de vue théorique, comment les réseaux de neurones peuvent être utilisés pour « apprendre » à un agent à jouer à un jeu qui, tel Tétris, présente beaucoup trop de configurations pour être stockées en mémoire.

Première partie

Le moteur de jeu

Chapitre 1

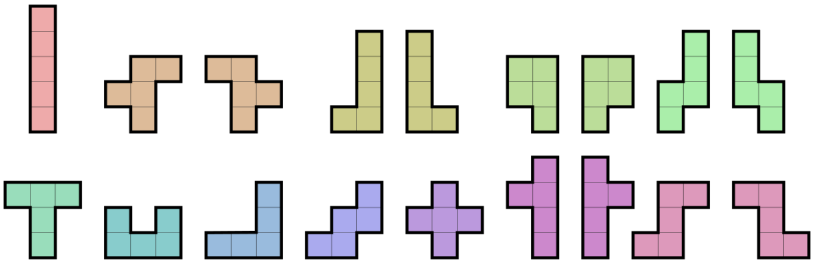
LE JEU TÉTRIS

1 L'origine du jeu

Tétris a été créé par Alekseï Leonidovitch Pajitnov au milieu des années 1980. Il était alors chercheur au centre informatique Dorodnitsyn de l'Académie des Sciences Soviétique, un laboratoire de recherche et développement de l'Union Soviétique. Il y travaillait à la reconnaissance de la voix humaine par les ordinateurs et, à ses heures perdues, s'amuser à programmer des jeux.

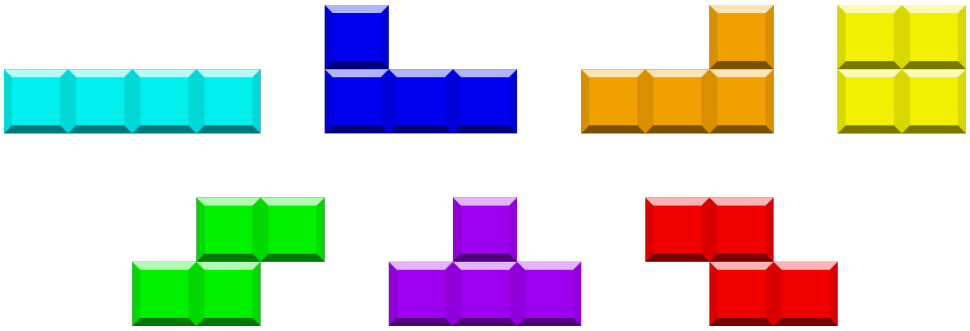
Alekseï Pajitnov
(Source :
`http://allrus.me/legendary-russian-game-programmer-alexey-pajitnov/`)

Il avait entendu parlé d'un puzzle créé par le mathématicien américain Salomon Golomb, le *pentomino*, dont le but était de recouvrir un rectangle avec des pièces de différentes formes, toutes obtenues par l'assemblage de 5 carrés identiques.



Les 18 pièces du Pentomino (en distinguant les pièces symétriques)
(Source : `https://fr.wikipedia.org/wiki/Pentomino`)

Il s'en procura un et, commençant à y jouer, découvrit que, malgré les apparences, ce n'était facile du tout ! Il eut alors l'idée d'en faire un jeu électronique dans lequel les pièces étaient choisies aléatoirement, l'une après l'autre, et à des intervalles de temps de plus en plus réduits. Il fit des tests mais devant le (trop) grand nombre de combinaisons qu'offraient les 18 pièces du pentomino, il décida d'opter pour une version simplifiée dans laquelle il n'y aurait que 7 types de pièces, toutes formées à l'aide de 4 carrés identiques. Ces pièces sont nommées les « tétraminos » [1], du grec *tetra* qui signifie *quatre*.



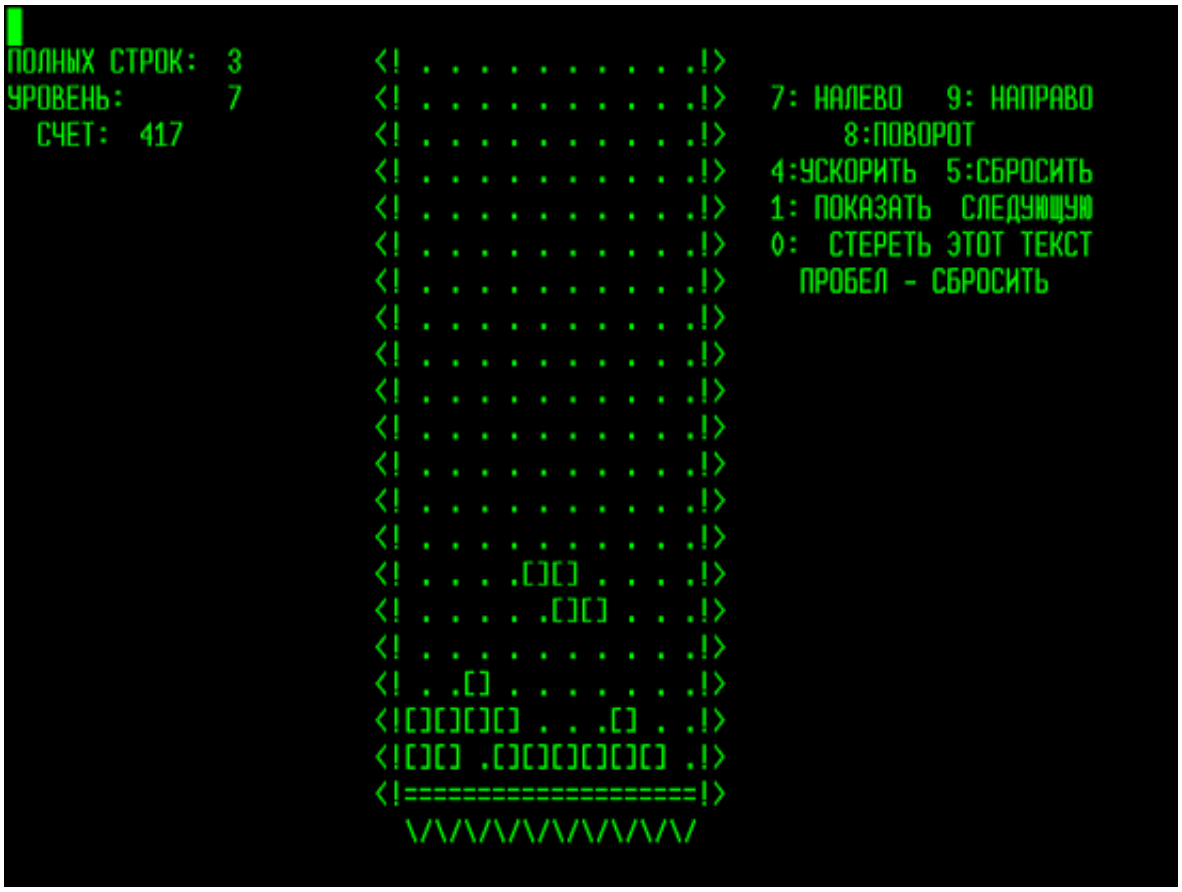
Les 7 tétraminos : I, J, L, O, S, T et Z.
(Source : `https://en.wiktionary.org/wiki/tetromino`)

Le jeu fut développé sur un Elektronica 60. Cet ordinateur ne disposait pas de fonctionnalités graphiques et les carrés formant les pièces furent alors représentés par un espace encadré de crochets : [ ].

2 Le principe du jeu

Le principe du jeu est le suivant : dans un champ de jeu rectangulaire, une pièce est choisie aléatoirement parmi les 7 tétraminos et se déplace du haut vers le bas. Le joueur peut faire tourner d'un, deux ou trois quarts de tour et déplacer horizontalement dans les deux sens la pièce afin de disposer le tétramino comme il le souhaite en bas du champ de jeu, sachant que les pièces successives s'empilent les unes sur les autres. Lorsque le joueur parvient à recouvrir une ligne complète du champ de jeu à l'aide de carrés des tétraminos, sans qu'il n'y ait plus aucun trou sur la ligne, celle-ci disparaît, rapportant un certain nombre de points, et la pile de tétraminos est décalée vers le bas, laissant ainsi plus de place dans le champ de jeu pour accueillir les pièces suivantes. Si, à un moment, le champ de jeu ne contient plus assez de place pour accueillir le tétramino suivant, la partie est perdue ! Ainsi, Tétris n'est pas un jeu dans lequel une partie se termine par la victoire du joueur. Il s'agit plutôt de faire le meilleur score possible.

1. On trouve aussi les appellations « tétrominos » ou « tétriminos ».



Première version de Tetris

(Source : `https://www.firstversions.com/2015/11/tetris.html`)

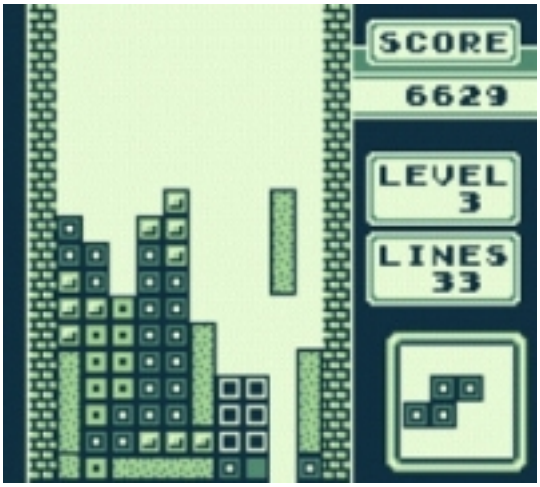
3 Diffusion du jeu

Sentant que le jeu serait plus attractif si les tétraminos apparaissaient « véritablement » à l'écran plutôt que leur représentation à l'aide des crochets, Pajitnov décida d'améliorer l'aspect visuel de son programme grâce à une version pour PC d'IBM intégrant une interface graphique. Pour cela, il fit appel à un jeune lycéen prodige de 16 ans, Vadim Gerasimov, qui lui avait été présenté par un autre programmeur du nom de Dmitri Pavlovsky. Gerasimov avait des compétences qui dépassaient largement celles de Pajitnov et Pavlovsky : il avait notamment appris seul à programmer dans un langage venu de l'Ouest : le Microsoft DOS. À l'issue de deux mois de travail, Gerosimov créa la première version « couleur » de Tétris qui intégrait également une table des meilleurs scores programmée par Pavlovsky. Selon Vadim Gerasimov, le nom *Tétris* résulte de la contraction des mots « tétramino » et « tennis », ce dernier étant le sport favori de Pajitnov.

Vadim Gerasimov
(Source : `http://www.stuff.co.nz/technology/games/2477794/Tetris-inventor-makes-waves-at-Google`)

Cette version se diffusa rapidement d'abord à Moscou puis dans les pays de l'ancien bloc soviétique. Une copie fut envoyée par Victor Brjabrin, le supérieur de Pajitnov, à l'Institut des Sciences Informatiques de Budapest où Robert Stein, un anglais d'origine hongroise travaillant pour une société de logiciels, découvrit le jeu. Stein vit le potentiel du jeu et envoya un fax au centre informatique Dorodnitsyn pour indiquer qu'il était intéressé par Tétris. Pajitnov lui répondit simplement qu'il était également intéressé et, à partir de cette simple réponse, Stein se mit à exploiter et vendre les droits de Tétris auprès de diverses compagnies occidentales, sans avoir signé le moindre accord avec les russes. Par la suite, il voulut établir un contrat en bonne et due forme avec Pajitnov et ses supérieurs mais ceux-ci étant confrontés à un monde inconnu pour eux, l'économie de marché, se montrèrent très méfiants et exigeants et les négociations n'aboutirent pas, ce qui n'empêcha pas Stein de continuer à exploiter le jeu. L'une des premières versions commerciales de Tétris fut celle éditée par Spectrum Holobyte sur PC IBM en 1986.
Par la suite, Elektronorgtechnica (abrégé ELORG), l'agence russe chargée de gérer les importations et exportations informatiques, repris le contrôle des négociations pour la gestion des droits et du marketing de Tétris, reprochant même à Pajitnov d'avoir donné un accord, fut-il de principe, à Stein, étant donné que, pendant l'ère soviétique, les créations intellectuelles des chercheurs russes étaient la propriété de l'état. Finalement, ELORG confirma l'accord avec Stein pour la gestion des droits du jeu mais seulement sur les ordinateurs et à l'exclusion de tout autre matériel électronique.
En 1989, les droits d'exploitation restants furent partagés entre Atari pour les bornes d'arcade et Nintendo pour les consoles. Minuro Arakawa, le président de la filiale américaine de Nintendo avait mandaté Henk Rogers, afin d'obtenir ces droits car il comptait faire de Tétris l'un des jeux phare de sa nouvelle console : la Game Boy. Celle-ci fut mise sur le marché en 1989 et connue un succès phénoménal avec des millions d'exemplaires vendus à

travers le monde. Si les ventes de la Game Boy participèrent à la diffusion de Tétris, celui-ci contribua également au succès de la console car le jeu était, dans un premier temps, implanté d'office sur la machine.



Le jeu Tétris sur Game Boy en 1989
(Source :
`https://de.wikipedia.org/wiki/Liste_der_erfolgreichsten_Computerspiele`)

Par la suite, Nintendo développa différentes variantes du jeu puis, après de l'effondrement de l'URSS en 1991, Pajitnov émigra aux États-Unis et récupéra finalement l'intégralité des droits de son jeu en 1996. Il fonda, avec Henk Rogers, The Tetris Compagny qui a depuis la charge exclusive de la gestion des droits du jeu. Ainsi, Pajitnov, qui n'a touché aucun droit d'auteur durant plus de 10 ans, peut aujourd'hui récolter les fruits de sa création.

4 Les raisons du succès

Tétris possède deux caractéristiques qui en font un modèle de jeu vidéo et qui peuvent expliquer son succès : d'une part, il est facile à prendre en main mais, dans le même temps, il possède un côté addictif et, d'autre part, malgré son apparente simplicité, il ne peut exister que sur un support électronique. Le principe du jeu fait qu'il procure une satisfaction quasi immédiate puisqu'on arrive très rapidement à supprimer une première ligne, puis une deuxième, etc... Il fait partie de la famille des « jeux occasionnels » (*casual gaming*) car une partie ne demande pas un gros investissement en terme de temps et on peut y jouer quelques minutes puis arrêter pour refaire une partie plus tard sans n'avoir rien perdu de significatif.
Le choix de Pajitnov de réduire le nombre de pièces à 7 est également un élément déterminant. En effet, 7 est le nombre d'« objets [2] » différents que l'esprit humain peut mémoriser rapidement et sans trop d'effort alors qu'à partir de 8, cela devient beaucoup plus difficile.

2. Il faut comprendre ici le mot *objets* dans un sens très large : images, mots, idées, nombres, etc...

Ainsi, 7 est le nombre idéal de pièces puisqu'elles peuvent être mémorisées rapidement par le joueur.
Enfin, la simplicité de son programme et le peu de ressources qu'il requiert lui a permis d'être développé sur tous les supports de jeu successifs (ordinateur, borne d'arcade, console, smartphone...)

5 Peut-on gagner à Tétris?

La principale difficulté dans Tétris est l'accélération de l'arrivée des nouvelles pièces qui devient vite difficilement gérable. Il est clair que si cette accélération continue indéfiniment, on atteint un palier au-delà duquel le jeu n'est plus jouable, ne serait-ce que parce que le temps de déplacement de la nouvelle pièce devient inférieur au temps matériellement nécessaire pour la déplacer à l'aide des touches de la console ou de l'ordinateur.
Imaginons qu'on mette de côté cet aspect du jeu et que les pièces arrivent à intervalle régulier. Dans ce cas, peut-on gagner à Tétris? Telle quelle, la question n'a pas beaucoup de sens puisque le jeu ne prend fin qu'avec la défaite du joueur. Dans son mémoire de master, J. Brzustowski [2] étudie la question sous l'angle suivant : existe-t-il une stratégie qui permette de jouer à Tétris indéfiniment. Il montre que cela dépend essentiellement du choix aléatoire des pièces. En particulier, il prouve que, quelle que soit la stratégie adoptée, il existe une suite de tétraminos S et Z qui conduit inéluctablement à la fin de la partie.
Notons, cependant, que, dans le cas d'un choix au hasard des tétraminos, la probabilité qu'une telle suite sorte est très faible. De plus, Brzustowski met en évidence que le jeu n'est pas programmé pour faire perdre le joueur par le choix des tétraminos. C'est d'ailleurs le contraire puisque, dans la spécification officielle du jeu (*Guideline*) disponible sur le site de The Tetris Compagny[3], il est précisé que les pièces sont en fait choisies par vague de 7, une vague consistant en une permutation quelconque des 7 tétraminos : c'est le principe du « sac aléatoire » (*random bag*). Dans cette configuration, des stratégies gagnantes sont possibles mais Brzustowski constate également que face à la réalité du jeu (i.e. en tenant compte des contraintes temporelles), l'expérience acquise pas un joueur est plus efficace qu'une stratégie mathématique prédéfinie.

3. `http://tetris.wikia.com/wiki/Tetris_Guideline`

Chapitre 2

RÈGLES UTILISÉES DANS LE PROJET

Les règles que nous allons utiliser sont très proches des règles officielles publiées sur le site https://tetris.fandom.com/wiki/Tetris_Wiki, à savoir :

- La grille de jeu est composée de 10 colonnes et 22 lignes, plus deux lignes invisibles pour l'apparition de la pièce suivante. Lors de certaines phases de tests on pourra limiter les dimensions de la grille.
- Les pièces sont les tétraminos standards. Lors de certaines phases de tests on pourra utiliser un jeu de pièces réduit au seul domino rectangulaire.
- On utilise le principe du « 7-bag Random Generator » : les 7 pièces du sac sont mélangées et tirées successivement. Lorsque le sac est vide on en recrée un nouveau.
- Les pièces sont mises en jeu horizontalement sur la dernière ligne invisible et centrée en colonne (dans le cas d'une pièce ayant une largeur impaire, elle est centrée à gauche).
- À chaque tour une seule action est effectuée (soit un déplacement, soit une rotation, soit rien) et la pièce est ensuite automatiquement descendue d'une ligne si c'est possible. Dans le cas contraire, une nouvelle pièce est mise en jeu.
- Nous nous autorisons la possibilité de placer directement une pièce dans une colonne et une rotation donnée, à la condition que cette colonne soit effectivement accessible en un nombre indéfini de coup à partir de sa position de chute.
- Lorsqu'une pièce ne peut pas tourner car elle se trouve au bord de la grille, le mouvement est annulé (on ne fait pas le « floor kick »).
- Les points sont comptabilisés selon le nombre de lignes faites en un coup :
  - 1 ligne : 40 points
  - 2 ligne : 100 points
  - 3 ligne : 300 points
  - 4 ligne : 1200 points

  On pourra également s'autoriser à prendre comme nombre de points le carré du nombre de lignes faites.
- Dans la mesure où on se contente de faire jouer des agents, il n'y a pas de temps ni de niveaux.

Chapitre 3

IMPLÉMENTATION DU MOTEUR DE JEU

1 Structures de données

Le moteur de jeu est construit autour de trois classes :

- `Tetramino` : les blocs
- `Board` : la grille de jeu
- `TetrisEngine` : le moteur de jeu qui fait le lien entre les deux classes précédentes

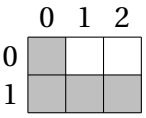
2 La classe Tetramino

La classe `Tetramino` est responsable de le gestion des blocs et de leurs rotations. Elle est implémentée dans le fichier `tetramino.py`.

2.1 Initialisation

La représentation d'un bloc est codée par la liste des tuples de la forme $(ligne,colonne)$ des coordonnées de ses cases occupées, le coin en haut à gauche ayant pour coordonnées `(0,0)` :

Par exemple, pour le bloc "J" placé horizontalement :



Ce bloc sera codé par la liste `[(0,0), (1,0), (1,1), (1,2)]`

Lors de l'initialisation d'un bloc, on va donner :

- Son `id` qui permettra d'identifier son type si besoin
- La liste de ses rotations dans le codage précédent
- Les coordonnées de son coin supérieur gauche et de son coin inférieur droit afin d'accélérer les tests de positionnement ultérieurs

2.2 Bornes de la pièces

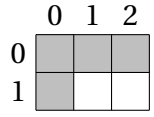
2.2.1 Coins

Les coins supérieurs gauches et inférieurs droits sont codés en dur dans le constructeur de la pièce et sont récupérés avec la méthode `getCorners(self)` qui renvoie un tuple de la forme $(i_{\min}, j_{\min}, i_{\max}, j_{\max})$.

2.2.2 Cellules du bas

Nous allons aussi avoir besoin, pour chaque colonne, de l'indice maximum de la ligne occupée.

Par exemple sur la pièce suivante :



La colonne $j = 0$ admet sa cellule la plus en bas en $i_{\max}(j) = 1$.
La méthode `getBottomCell(self, j)` renvoie cet indice pour la colonne j et la méthode `getLowerCells(self)` renvoie la liste des tuples $(i_{\max}(j), j)$ pour j allant de $j_{\min}$ à $j_{\max}$.
Sur l'exemple précédent,on obtiendrait : `[(0,1), (1,0), (2,0)]`.

2.3 Rotations

La rotation courante du bloc est mémorisée dans un attribut `glyph_index` qui correspond à l'indice courant dans la liste de ses rotations.
On peut :

- Soit tourner le bloc dans un sens avec la méthode `rotate(self, direction)`, où `direction` vaut `'H'` pour le sens horaire et `'T'` pour le sens trigonométrique.
- Soit le placer directement dans une rotation désirée avec la méthode `setDirection(self, i)` où `i` est l'indice de la rotation voulue.

2.4 Méthodes utilitaires

Cette classe implémente une méthode `copy(self)` qui renvoie une copie du bloc ainsi qu'une méthode `toArray(self)` pour avoir une représentation matricielle dans une matrice carrée dans laquelle chaque case occupée vaut `id` et les case vides 0. Cette dernière méthode sera utile pour la partie réseaux de neurones pour entrer la pièce sous la forme d'un vecteur 0/1.

2.5 Sacs de pièces

Après la définition de la classe `Tetramino`, on définit les constantes correspondant à chaque bloc et on en fait la liste dans des constantes `BLOCK_BAG`.
On dispose de trois type de sacs de pièces :

- `RAPID_BLOCK_BAG` : Chaque bloc a un nombre limité de rotation suivant ses symétries. Par exemple le bloc "I" n'a que deux rotations. Ce sac est utile pour les placements directs des pièces et économise des calculs inutiles.
- `CLASSIC_BLOCK_BAG` : Chaque bloc (sauf le "O") a quatre rotations pour coller aux règles du Tétris dans lesquelles un bloc tourne autour du centre de sa matrice carrée. Ce sac sera utile pour simuler le comportement d'un vrai joueur humain.
- `DOMINO_BLOCK_BAG` : Ce sac contient un seul domino et sera utilisé pour l'implémentation de l'apprentissage profond sur un jeu simplifié.

3 La classe Board

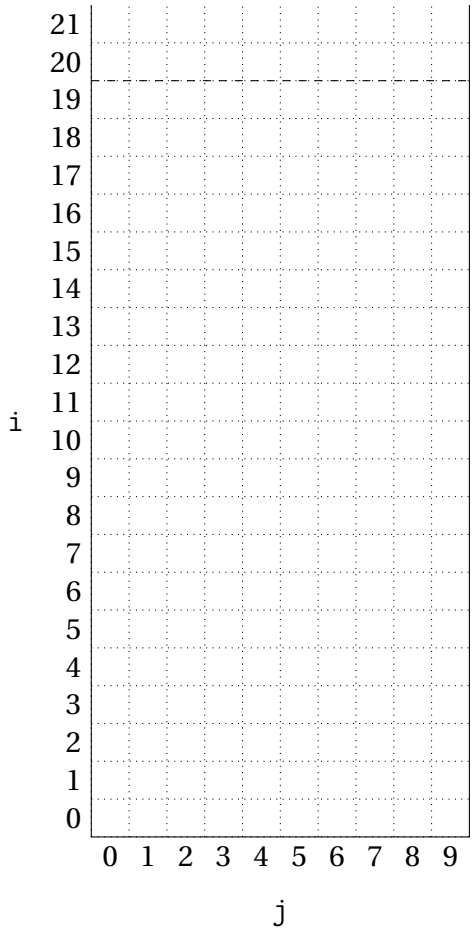
La classe `Board` implémente la grille, ses méthodes de gestion (mise à jour des cellules, traitement des lignes, ...) ainsi que les outils statistiques (nombre de trous, hauteur maximum, ...). Elle est implémentée dans le fichier `board.py`.

3.1 Constructeur

Le constructeur de la classe `board` admet deux paramètres :

- Sa largeur : `width=10` par défaut
- Sa hauteur : `height=22` par défaut

La grille en elle-même est stockée dans la liste double `self.grid` qui a pour largeur `width` et pour hauteur `height+2` (avec les deux lignes invisibles du dessus).
`self.grid[i][j]` correspond au contenu de la case située à la ligne `i` dans la colonne `j`. Ce contenu est soit `0` pour une case vide, soit l'id de la pièce dont fait partie cette case.
Notons que la ligne d'indice 0 correspond à la ligne du bas :



Enfin, les différents indicateurs statistiques sont initialisés.

3.2 Gestion des cellules

La gestion des cellules de la grille est gérée par des getters et setters qui présentent peu d'intérêt et dont les noms sont explicites.

Notons toutefois les méthodes suivantes qui seront utilisées lors de la suppression des lignes pleines :

- `isLineFull(self, i)` qui teste si la ligne `i` est vide.
  Il s'agit simplement de tester, pour chaque cellule de la ligne donnée si elle est vide et, si c'est le cas, renvoyer `False` (sinon renvoyer `True` à la fin du test).
- `removeLine(self, i)` qui supprime la ligne `i` de la grille et rajoute une ligne vide en haut.
  Pour se faire, on utilise le fait que les listes en Python sont implémentées par des listes chaînées. Pour supprimer une ligne, il suffit donc de la détruire avec `del` et d'en ajouter une vide en fin de liste.

3.3 Récupération des caractéristiques de la grille

Les méthodes suivantes permettent de récupérer, dans des attributs spécifiques les différentes caractéristiques de la grille :

- `columnHeight(self, j)` : renvoie la hauteur de la colonne `j`. Le principe de l'algorithme est de partir du haut de la grille et de décrémenter cette hauteur tant que la cellule visitée est vide :

  ```
  i = hauteur de la grille
  Tant que i>=0 et la cellule (i,j) est vide :
     i = i-1
  Renvoyer i+1
  ```

  Le parcours se fait en partant du haut car sinon la boucle s'arrêterait au premier trou trouvé.
  Notons que cette fonction renvoie bien la hauteur de la colonne et non l'indice de la ligne de la case la plus haute.
- `getColumnHeights(self)` : renvoie la liste des hauteurs des colonnes.
- `getMaxHeight(self)` et `getSumHeights(self)` : renvoient respectivement la hauteur maximum et la somme des hauteurs des colonnes.
- `getBumpiness(self)` : renvoie la somme des valeurs absolues des différences de hauteurs entre les colonnes consécutives.
- Pour la détermination du nombre de trous, on procède de la façon suivante : un trou est défini comme une cellule vide dans une colonne qui contient une cellule pleine au-dessus (cellule dominée).
  - `isDominated(self, i, j)` : teste si la cellule `(i,j)` est dominée.

```
Si la cellule (i,j) n'est pas vide :
   Renvoyer Faux
Sinon :
   Pour k allant de i+1 à (hauteur de la colonne j) - 1 :
       Si la cellule (i,j) n'est pas vide :
           Renvoyer Vrai
   Renvoyer Faux
```

  - `getNbHoles(self)` : renvoie le nombre de trous en comptant pour chaque cellule si elle est dominée.
- `updateStats(self)` : met à jour toutes les caractéristiques de la grille

3.4 Gestion des lignes

La méthode `processLines(self)` supprime les lignes pleines et renvoie le nombre de lignes supprimées :

```
nb_lignes = 0
hauteur_max = hauteur de la grille
i = 0
Tant que i < hauteur_max :
   Si la ligne i est pleine :
       Enlever la ligne i
       hauteur_max = hauteur_max-1
       nb_lignes = nb_lignes +1
   Sinon :
       i = i + 1
Renvoyer nb_lignes
```

3.5 Méthodes utilitaires

- `copy(self)` : renvoie une copie de la grille.
- `__str__(self)` : renvoie une chaîne de caractère pour affichage de la grille.
- `printInfos(self)` : affiche les caractéristiques de la grille (pour les tests).

4 La classe TetrisEngine

Cette classe fait le lien entre les deux précédentes et implémente le déroulement de la partie. Elle est implémentée dans le fichier `tetris_engine.py`.

4.1 Constructeur et attributs

Les paramètres du constructeur sont les suivants :

- `width` et `height` : les dimensions de la grille
- `getMove` : c'est la fonction qui va renvoyer, à chaque tour, le coup à jouer. C'est cette fonction que vont implémenter les agents (fonction de callback).
- `max_blocks` : le nombre maximum de blocs à jouer pour limiter les temps des essais (0 pour jouer jusqu'à la fin de la partie)
- `base_block_bag` : le sac dans lequel on va tirer les pièces, comme défini dans le module `tetramino.py`
- `temporisation` : en secondes, un temps de pause entre deux mouvements
- `silent` : si `True`, ne produit aucun affichage (pour les essais sur plusieurs parties)
- `random_generator_seed` : la graine du générateur aléatoire pour reproduire des tests si besoin
- `agent_name` et `agent_description` : le nom et la description de l'agent, pour l'affichage.

Le constructeur va également définir les attributs suivants :

- Gestion de la grille :
  - `self.board` : la grille de jeu dans laquelle les pièces évoluent.
  - `self.fixed_board` : la grille de jeu statique dans laquelle les pièces sont placées. Cette grille est une copie de `self.board` à la fin de chaque coup.
- Gestion des blocs :
  - `self.block_bag` : la sac contenant les pièces à jouer
  - `self.block` : le bloc courant en jeu
  - `self.block_position` : les coordonnées du coin en haut à gauche du bloc courant
  - `self.next_block` : le bloc suivant
  - `self.nb_blocks_played` : le nombre de pièces déjà jouées
- Gestion des scores :
  - `self.score` : score total de la partie
  - `self.score_on_move` : score gagné sur le dernier coup (pour la fonction d'évaluation)
  - `self.total_lines` : le nombre total de lignes faites sur la partie
- Gestion de la partie :
  - `is_running` : Flag pour indiquer que la partie est en cours
- Gestion du temps :
  Différentes variables pour chronométrer différents aspects de la partie (temps par coup, temps moyen par coup, temps total pour les coups, temps de jeu,...)

4.2 Génération des pièces

La génération d'une nouvelle pièce dans le jeu se fait avec la méthode `getNewBlock()` et suit l'algorithme suivant :

- Si le sac de pièces est vide, on en recrée un nouveau que l'on mélange grâce à la méthode `generateNewBlockBag()`
- On tire une pièce du sac avec la méthode `generateNewBlock()`
- On place la pièce sur la ligne du dessus au centre de la grille avec la méthode `setBlockInitPosition()`

4.3 Déplacements et rotations

La méthode `isMoveValid(self, block, new_position)` teste si le bloc peut être placé dans une nouvelle position candidate.
Pour cela on teste d'abord qu'elle ne sort pas de la grille, puis que les cases qui seraient occupées dans la nouvelle position sont bien vides.

Pour déplacer un bloc, on commence par l'effacer de la grille avec la méthode `eraseBlock()`. Cette méthode se contente de vider les cellules de la grille occupée actuellement.
Ensuite on le place dans son nouvel emplacement en mettant l'id du bloc dans les cellules qu'il occupe.

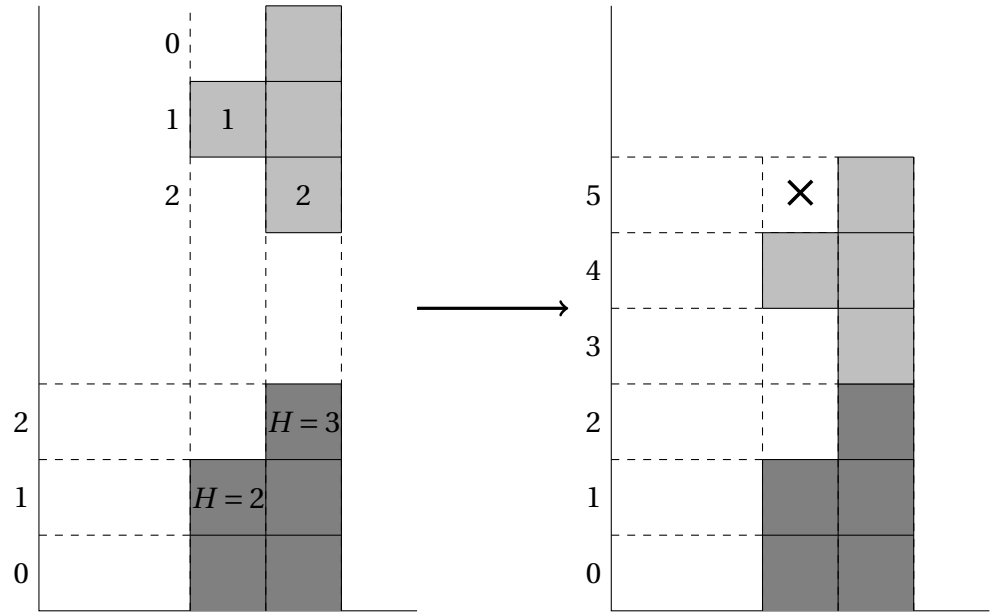
Partant de là, il est facile de déplacer un bloc dans une direction, avec la méthode `moveBlockInDirection(self, direction)` où direction vaut `'L'` pour gauche, `'R'` pour droite et `''` pour un simple déplacement vers le bas.

Pour les rotations, c'est le même principe sauf qu'avant de placer le bloc on lui fait subir la rotation désirée.

La méthode `dropBlock()` fait tomber directement le bloc en bas de sa colonne.
Au départ cela se faisait naïvement en bouclant tant que la pièce pouvait descendre d'une ligne mais cela constituait un gaspillage de ressources. Afin de gagner en performances, nous allons calculer directement l'indice de la ligne où placer le bloc (son coin supérieur gauche), la colonne étant déjà déterminée.
Pour se faire, on va utiliser :

- Les hauteurs des colonnes de la grille pour les colonnes occupées par la pièce
- Les lignes des cases les plus basses occupées par la pièce

La ligne où placer la pièce est alors la plus grande valeur de la somme de la hauteur de la colonne j et de la ligne la plus basse occupée par la pièce dans la colonne j, pour j parcourant les colonnes occupées par la pièce.



Sur cet exemple, $2+1=3$ et $3+2=5$ donc la pièce doit être placée sur la ligne 5.

Enfin, il est possible de placer directement le bloc dans une colonne et une rotation donnée en bas de la grille.
Pour cela on commence par tester si le bloc peut bien être placé dans la colonne et la rotation voulue et on le fait tomber.
Notons la méthode `getPossibleMoveDirect(self)` qui renvoie tous les tuples de la forme `(colonne, rotation)` pour lesquels on peut placer le bloc directement. Cette fonction sera très utile dans la suite pour calculer le meilleur coup.

4.4 Commandes de jeu

L'interface entre le moteur et les agents se fait grâce à un système de commandes qui sont interprétées dans la méthode `playCommand(self, command)`. Ce sont des chaînes de caractères que l'agent va envoyer au moteur. Ces commandes sont :

- `'L'` et `'R'` : Déplacement à gauche et à droite
- `'D'` : Lâche la pièce en bas (hard drop)
- `'N'` : Ne fait rien (déplacement d'une case vers le bas)
- `'H'` et `'T'` : Tourne la pièce dans le sens horaire ou trigonométrique
- `'P:j:r'` : Place directement la pièce dans la colonne `j` et la rotation `r`

4.5 Déroulement de la partie

Le déroulement du jeu se fait dans la méthode `run(self)`. L'algorithme est le suivant :

- Tant que le jeu tourne
  - Met un nouveau bloc en jeu
  - Tant que la pièce peut descendre et que le jeu tourne :
    - — Descendre la pièce d'un cran
    - — Met à jour le score sur un mouvement
    - — Met à jour l'affichage
    - — Récupère le prochain mouvement
    - — Joue ce mouvement
    - — Traite les lignes faites
    - — Met à jour le score sur les lignes
  - Fixe la grille
  - Teste la fin du jeu
- Retourne le score

Chapitre 4

LES AGENTS

Les agents sont à la base du projet dont le but est, justement, de les programmer.

1 Généralités

Un agent est essentiellement une classe qui a accès à un moteur de jeu (et donc à tous les paramètres de jeu) et qui implémente une méthode `getMove()` qui renvoie la commande du prochain coup à jouer.
C'est de leur responsabilité de lancer la partie.
Les fonctionnalités de base des agents sont implémentées dans la classe `Agent` dont les agents héritent tous. cette classe permet de :

- Récupérer les paramètres de la grille après qu'un coup ait été joué via la méthode `getMoveStats(self, move)`. Cette méthode est utilisée dans la méthode `allMoveStats(self)` qui remplit un dictionnaire dont les clefs sont les différents placements possibles et les valeurs un dictionnaire content les différentes statistiques de jeu (nombre de lignes créées, nombre de trous, ...)
- Créer une commande pour un placement direct via la méthode `commandFromMove(self, move)`.

2 Joueur humain en mode texte

Pour tester les différentes fonctionnalités du moteur, nous avons implémenté un agent, `AgentHuman`, qui se contente de recevoir les différentes commandes à jouer via l'entrée standard.
Grâce à cet agent nous avons pu tester le déplacement et la rotation des pièces, la suppression des lignes, ...

3 Agent aléatoire

Ensuite le premier agent automatique qui joue de manière aléatoire.
`AgentRandom1` joue des coups aléatoires de type "aller à gauche", "tourner la pièce", ... à la manière d'un joueur humain.
`AgentRandom2`, quant à lui, place directement les pièces dans des colonnes et des rotations aléatoires.

Évidemment ces deux agents sont catastrophiques en terme de performances mais ne demandent qu'à être améliorés (ces sont les "hello world" des agents).

4 Agent par filtrage

Le premier agent un tant soit peu efficace.
La stratégie utilisée est de filtrer la liste des coups jouables successivement selon plusieurs critères :

- D'abord il ne garde que les coups qui font le moins de trous
- Ensuite, parmi eux, il ne garde que ceux qui donnent une somme des hauteurs des colonnes de la structure minimale
- Puis, ceux qui minimisent le bumpiness
- Enfin ceux qui créent le plus de lignes

Cet agent joue plutôt bien pour une heuristique aussi simple mais il a tendance à créer des puits pour éviter de faire des trous.

5 Agent par évaluation des coups

Pour chaque coup jouable notons L le nombre de lignes, H la somme des hauteurs des colonnes, T le nombre de trous créés et B le bumpiness, après que le coup a été joué.
La qualité d'un coup peut être évaluée par une fonction

$$q(L,H,T,B) = a \times L - b \times H - c \times T - d \times B$$

où a, b, c et d sont des paramètres positifs, et nous pouvons supposer que le vecteur (a,b,c,d) est normé dans $\mathbb{R}^4$ (soit pour la norme $||\cdot||_1$, soit pour la norme $||\cdot||_2$).

Le coup à jouer est alors celui qui maximise cette fonction.

Tout le problème consiste donc à déterminer ces coefficients et c'est le but de l'optimisation par algorithme génétique.

Deuxième partie

Optimisation par algorithmes génétiques

Chapitre 5

LES ALGORITHMES GÉNÉTIQUES

Les algorithmes génétiques sont des algorithmes évolutionnistes initiés par John Holland dans les années 60 et popularisés par David Goldberg à la fin des années 80. Il s'agit de trouver une solution à un problème d'optimisation en faisant évoluer une population par analogie avec des concepts issus de la biologie comme le principe de sélection naturelle de Darwin, la recombinaison génétique et la mutation génétique.

1 Principe général

On considère un entier $n \in \mathbb{N}^*$ et une fonction continue $f : [0;1]^n \to \mathbb{R}_+$ qu'on cherche à maximiser. Comme $[0;1]^n$ est compact, on sait que f est bornée et atteint ses bornes. Ainsi, il existe au moins un élément $a \in [0;1]^n$ tel que, pour tout $x \in [0;1]^n$, $f(x) \leqslant f(a)$. Le but est ici de déterminer une valeur approchée de a et de $f(a)$.

On se donne un entier naturel non nul N et une population P_0 de N éléments choisis aléatoirement dans $[0;1]^n$ i.e. une partie aléatoire $P_0 = \{x_1, x_2, \ldots, x_N\} \subset [0;1]^n$. On choisit un codage des éléments de P_0 qui sera assimilé à son génome.

Pour tout entier $i \in \mathbb{N}$, à partir de la population P_i, qui constitue la génération i, on définit une nouvelle population P_{i+1}, qui constitue la génération $i+1$, de la manière suivante :

- on conserve éventuellement une partie A de cardinal $M \geqslant 0$ de la population P_i ;
- on choisit $N - M$ couples d'éléments (parent1, parent2) dans la population P_i ;
- chaque couple (parent1, parent2) engendre un ou deux « enfants » ;
- un « enfant » subit éventuellement une mutation génétique (autrement dit, une modification aléatoire de son codage) ;
- on réunit les M éléments de A et les $N - M$ meilleurs des nouveaux « enfants » créés pour former la nouvelle génération.

Le choix des éléments de A et des couples de « parents » va se faire de façon à maximiser f. En revanche, les mutations génétiques sont purement aléatoires et ont pour but d'éviter que le processus ne converge vers un élément qui ne réalise qu'un maximum local.

Nous examinons à présent en détails le déroulement de ces différentes étapes.

2 Codage

Les individus considérés ici sont des $n-$uplets de nombres réels appartenant à l'intervalle $[0;1]$. Dans la pratique, on ne travaille qu'avec des décimaux qu'on peut coder deux façons différentes.

2.1 Codage décimal

Dans ce codage, on utilise simplement la représentation par un flottant. Un élément de la population est donc un vecteur de n flottants compris entre 0 et 1. Ainsi, le génome est ce vecteur de n flottants et les n gènes sont les n flottants.

2.2 Codage binaire

On se donne un nombre entier naturel fixé B et on définit les éléments x_j de la population sous la forme de vecteurs de nombres décimaux codés en binaire sur B bits. Ainsi, si $B = 8$, on représentera chaque décimal composant le vecteur x_j sous la forme d'un tableau du type

$$[\varepsilon_0, \varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4, \varepsilon_5, \varepsilon_6, \varepsilon_7]$$

avec $\varepsilon_k \in \{0;1\}$ pour tout $k \in [\![0,7]\!]$. Précisément, le tableau précédent code le nombre décimal

$$\sum_{k=0}^{7} \frac{\varepsilon_k}{2^{k+1}}.$$

Ainsi, le tableau [0, 1, 0, 0, 1, 1, 0, 0] représente le décimal $\frac{1}{2^2} + \frac{1}{2^5} + \frac{1}{2^6} = 0{,}296875$.
Avec ce codage, le génome d'un individu est un vecteur de n tableaux binaires à B éléments et chacun de ces $n \times B$ bits est un gène de l'individu.

3 Filtrage

On peut décider, à chaque nouvelle génération, de conserver – ou non – une partie de la génération des meilleurs individus de la génération précédente i.e. ceux qui maximisent la fonction f : on parle alors de filtrage de la population.
Il existe plusieurs façon de « filtrer ». Une possibilité est de fixer une pourcentage de la génération précédente, représentant l'« élite », que l'on conserve. Dans ce cas où on conserve M éléments de la génération précédente, on ne doit créer que $N - M$ nouveaux éléments.
Une autre possibilité est de créer une génération nouvelle partielle (contenant donc moins de N individus), de réunir l'ancienne et la nouvelle génération et de ne conserver que les N meilleurs éléments.
On peut aussi décider de ne pas faire de filtrage et de renouveler entièrement la population en créant N nouveaux éléments.

4 Reproduction

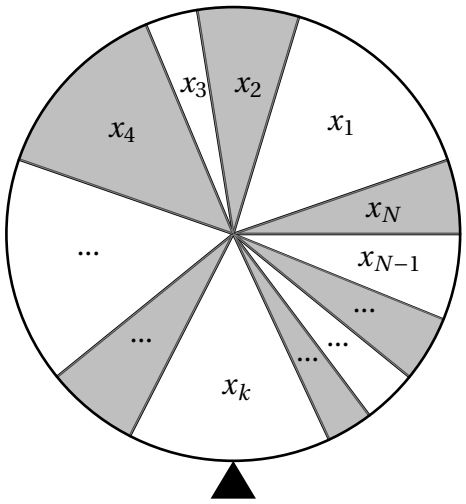
Pour créer les nouveaux éléments, on va sélectionner des couples de « parents » et simuler une reproduction en réalisant un « croisement » entre les génomes des parents.

4.1 Sélection des parents

Pour sélection les parents, on dispose de deux méthodes.

4.1.1 Sélection proportionnelle à l'adaptation

On note $P = \{x_1, x_2, \ldots, x_N\}$ la population pour un certaine génération. On souhaite sélectionner les couples de parents parmi les individus ayant la meilleure adaptation i.e. les individus x ayant les plus grandes valeurs de $f(x)$. Pour cela, on définit sur P une probabilité $\mathbb{P}$ telle que, pour tout $j \in [\![1, N]\!]$, $\mathbb{P}(\{x_j\})$ soit proportionnelle à $f(x_j)$. Autrement dit, en posant $S = \sum_{j=1}^{N} f(x_j)$, on pose $\mathbb{P}(\{x_j\}) = \dfrac{f(x_j)}{S}$ pour tout $i \in [\![1, N]\!]$.
On choisit ensuite un nombre aléatoire $r \in [0;1]$ et on sélectionne dans la population le premier indice k tel que $\sum_{j=1}^{k} \mathbb{P}(\{x_j\}) \geqslant r$.
De façon imagée, cela revient à choisir x_k par la « méthode de la roulette ». On considère une roulette partagée en N secteurs angulaires correspondant chacun à un x_j et dont les aires sont respectivement proportionnelles aux $f(x_j)$. On fait alors tourner la roulette et on sélection le secteur (et donc le x_k) sur lequel la roulette s'arrête.



4.1.2 Sélection par tournoi

Une autre façon de choisir un couple de parents est de choisir au hasard une partie E de la population $P = \{x_1, x_2, \ldots, x_N\}$ et de conserver les deux meilleurs éléments de E i.e. les

deux éléments a et b de E tels que, pour tout $x \in S$, $f(a) \geqslant f(b) \geqslant f(x)$.
Cette méthode est appelée *sélection par tournoi* car cela revient à organiser un tournoi entre les éléments de E (le gagnant d'une partie entre deux éléments étant celui qui maximise f) et à conserver les deux vainqueurs du tournoi.

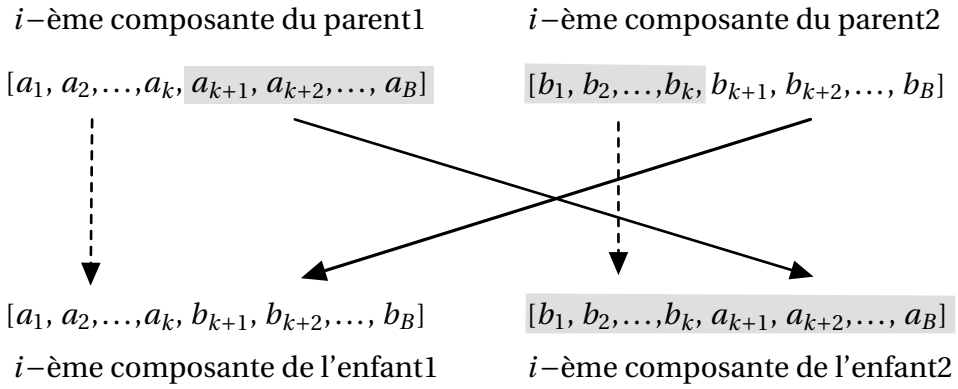
4.2 Reproduction

Une fois les deux parents choisis, on procède à la « reproduction » pour engendrer un ou deux enfants, selon le codage choisi.

4.2.1 Reproduction par enjambement

La reproduction par enjambement ou par croisement (*cross-over*) s'applique au codage binaire. Pour chaque entier i entre 1 et n, on construit la $i-$ème composante des vecteurs enfant1 et enfant2 de la manière suivante.
On suppose que le codage binaire de la $i-$ème composante du parent1 est $[a_1, a_2, \ldots, a_B]$ et le codage binaire de la $i-$ème composante du parent2 est $[b_1, b_2, \ldots, b_B]$. On choisit aléatoirement un rang k (propre à chaque i) entre 1 et B et alors

- le codage binaire de la $i-$ème composante du premier « enfant » est obtenu en prenant les k premiers bits du parent1 et les $B-k$ derniers bits du parent2 ;
- le codage binaire de la $i-$ème composante du second « enfant » est obtenu en prenant les k premiers bits du parent2 et les $B-k$ derniers bits du parent1.



4.2.2 Reproduction par combinaison linéaire

Pour le codage décimal, étant donné deux vecteurs parents p et q, on définit le vecteur enfant e de p et q par

$$e := f(p)p + f(q)q.$$

Autrement dit, pour tout entier i entre 1 et n, la $i-$ème coordonnée e_i de e est $e_i := f(p)p_i + f(q)q_i$ où p_i et q_i sont respectivement les $i-$èmes coordonnées des vecteurs p et q.

Dans le cas où on veut travailler avec des vecteurs normés, on divise de plus e par sa norme.

5 Mutation génétique

La mutation génétique apporte une modification aléatoire des « enfants » qui évite que le processus ne stagne autour d'un maximum local de la fonction f. Le procédé de mutation dépend du codage mais dans tous les cas, il a lieu avec une probabilité p fixée assez faible.

5.1 Mutation pour le codage binaire

Si le codage adopté est le codage binaire, le procédé de mutation consiste à changer de façon aléatoire la valeur d'un bit. Plus précisément, pour chaque bit de chaque composante d'un vecteur, on choisit aléatoirement un nombre aléatoire r entre 0 et 1 et si $r < p$, on change le bit b en $1 - b$.
Supposons, par exemple, que $B = 8$, $p = 0{,}1$ et que la première composante du vecteur soit [0, 0, 1, 0, 1, 1, 0, 0]. Si on obtient pour les valeurs de r successivement

$$0{,}72, \quad 0{,}33, \quad 0{,}06, \quad 0{,}51, \quad 0{,}08, \quad 0{,}18, \quad 0{,}58 \quad \text{et} \quad 0{,}06,$$

on va changer les bits 3, 5 et 8 de cette première composante, ce qui donne la nouvelle composante « mutée » [0, 0, $\underline{0}$, 0, $\underline{0}$, 1, 0, $\underline{1}$].

5.2 Mutation pour le codage décimal

Si le codage adopté est le codage décimal, on mute un vecteur en modifiant de façon aléatoire les composantes du vecteur. Plus précisément, on choisit un réel r entre 0 et 1. Si $r < p$, on choisit un réel δ dans un intervalle fixé $[-\eta;\eta]$ (où $\eta > 0$) et on modifie chaque composante v_i du vecteur v en $v_i + \delta$ à condition que $v_i + \delta$ reste dans l'intervalle $[0;1]$ (sinon, on ne change pas v_i).
Par exemple, supposons que $n = 4$, $p = 0{,}1$, $v = [0{,}12,\ 0{,}753,\ 0{,}179,\ 0{,}548]$ et $\eta = 0{,}3$. Si on obtient un réel $r < 0{,}1$, on choisit un nombre aléatoire dans $[-0{,}3;0{,}3]$, par exemple $\delta = -0{,}21$, et alors le vecteur muté est $v_{\text{muté}} = [0{,}12,\ 0{,}543,\ 0{,}179,\ 0{,}338]$.
Dans le cas où on veut travailler avec des vecteurs normés, on divise de plus $v_{\text{muté}}$ par sa norme.

6 Constitution de la nouvelle génération

La population de la nouvelle génération est finalement obtenue de la manière suivante :

- si on a choisi une politique de filtrage par élitisme, on a conservé M individus de la génération précédente. On leur rajoute les $N - M$ nouveaux individus créés en cas de codage décimal et on obtient la nouvelle génération de N individus. Si le

codage choisi est binaire, on a créé $2(N-M)$ nouveaux individus. On les ajoute au M individus de la génération précédente puis on ne conserve que les N meilleurs éléments parmi les $2(N-M)+M = 2N-M$ individus obtenus.

- si on a choisi un politique de filtrage par conservation des meilleurs, on créé une nouvelle population partielle contenant un nombre $M := \lfloor N \times t \rfloor$ d'individus où $t \in [0;1]$ est un pourcentage fixé. On réunit ensuite l'ancienne génération et la nouvelle génération partielle et on ne conserve que les N meilleurs éléments (au sens de la fonction f).
- si on n'a pas choisi une politique de filtrage par élitisme, on n'a conservé aucun individu de la génération précédente et le principe est le même que dans le premier point mais avec, cette fois, $M = 0$.

7 Itération du procédé

Partant d'une population initiale aléatoire, on itère le procédé sur plusieurs générations en conservant à chaque étape le taille de la population initiale. On arrête l'itération au bout d'un certain nombre de générations et on détermine, dans la population finale, un élément y qui maximise f. Si le nombre de générations est suffisant, $f(y)$ est une bonne approximation de $f(a) = \max\limits_{x\in[0;1]^n} f(x)$.

Chapitre 6

IMPLÉMENTATION POUR TÉTRIS

1 La classe AGOptimizer

L'algorithme génétique est implémenté par la classe `AGOptimizer`.
Celle-ci prend en paramètres notamment :

- `population_size` : la taille N de la population à chaque génération ;
- `nb_generations` : le nombre de générations créées ;
- `nb_bits` : le nombre B de bits pour le codage binaire ;
- `proba_mutation` : la probabilité p de mutation génétique
- `mutation_rate` : le taux de mutation η qui détermine l'intervalle $[-\eta,\eta]$ dans lequel est choisie la perturbation δ lors d'une mutation génétique pour le codage décimal ;
- `percentage_for_tournament` : le pourcentage d'éléments de la population qui participent lors de la sélection des parents par tournoi ;
- `percentage_new_offspring` = pourcentage t de la nouvelle génération qui est créée avant de réunir les enfants et les parents et de ne conserver que les meilleurs ;
- `elitism_percentage` : le pourcentage d'éléments de l'ancienne génération qui est conservé lorsqu'on effectue un filtrage par élitisme.

2 Fonction d'évaluation

Les algorithmes génétiques ont pour but de maximiser une fonction. Comment utiliser cela pour obtenir un agent jouant de manière efficace à Tétris ?
L'idée est de définir une fonction d'évaluation (*fitness* en anglais) qui est d'autant plus grande que l'agent joue bien. Ainsi, maximiser f revient à déterminer un agent qui joue le mieux possible.

La fonction d'évaluation choisie ici est le score moyen par partie obtenue en faisant jouer l'agent selon la fonction d'évaluation de la qualité d'un coup définie dans le paragraphe **5** du chapitre 3 (p. 24) :

$$q(L,H,T,B) = a \times L - b \times H - c \times T - d \times B.$$

Notre but est de déterminer les paramètres a, b, c et d dans $[0;1]$ afin que l'agent choisisse le meilleur coup à jouer i.e. le coup qui maximise le score.
Ainsi, f est la fonction qui associe au vecteur $(a,b,c,d) \in [0;1]^4$ le score moyen par partie lorsque l'agent joue par évaluation des coups selon la fonction q. Cette fonction f est implémentée par la méthode `fitness(self, vector)`.

3 Codage

Le codage du vecteur (a,b,c,d) est effectué en binaire ou en décimal selon le procédé décrit dans le chapitre 4.
Les fonctions `binToFloat` et `binVectorToFloat` servent à convertir l'écriture binaire en flottant pour l'évaluation de la fonction f.
La fonction `normalize` normalise un vecteur en le divisant par sa norme (s'il est non nul).

4 Initilisation de la population

La méthode `initPopulation(self)` détermine la population initiale aléatoire en fonction du codage choisi en utilisant la méthode utilitaire `randomBinaryVector(self)` ou `randomFloatVector(self)`.

5 Reproduction

5.1 Sélection des parents

La sélection des parents se fait à l'aide de la méthode `wheelSelection(self)` pour une sélection par « roulette » ou de la méthode `tournamentSelection(self)` pour une sélection par tournoi.

5.2 Croisement

Le croisement permettant d'engendrer les enfants se fait à l'aide de la méthode `crossover(self, parent1, parent2)` par une simple concaténation de tableaux dans le cas du codage binaire et en utilisant la fonction utilitaire `linearCombination` pour réaliser la combinaison linéaire dans le cas du codage décimal.

6 Mutation

La mutation génétique d'un individu se fait via la méthode `mutate(self, individu)` qui appelle, selon que le codage est binaire ou décimal, l'une des méthodes `mutateBinVector(self, bin_vector)` ou `mutateFloatVector(self, vector)`.

7 Création de la nouvelle génération

La méthode `generateNewOffspring(self)` détermine le nombre de nouveaux individus à créer et effectue cette création et la méthode `makeNewGeneration(self)` détermine la nouvelle génération en fonction du filtrage choisi en faisant appel à l'une des méthodes `keepOnlyElite(self)` ou `deleteWorst(self)`.

8 Résultats et limites

Les résultats obtenus sont plutôt bons mais aléatoires (voir, notamment, les écarts-types obtenus dans les tests du chapitre 10). Un agent optimisé par algorithme génétique peut jouer plusieurs centaines de milliers de coup sur une partie mais perdre après seulement quelques centaines sur une autre.
La raison de ces résultats fluctuant réside dans le fait que la fonction f est en fait aléatoire. En faisant tourner l'algorithme génétique plusieurs fois, on obtient donc des quadruplets (a,b,c,d) très différents d'une fois sur l'autre. À chaque fois, l'agent obtenu va être efficace dans les situations qu'il a rencontrées (ou proches de celle-ci) mais peut devenir très mauvais dans une suite de configurations mal connues.
De plus, deux configurations différentes peuvent conduire aux mêmes valeurs de L, H, T et B sans qu'il soit évident qu'il faille jouer de la même façon dans ces deux configurations.

Ainsi, l'utilisation de la fonction f a un côté trop aléatoire et trop simplificateur. Pour prendre en compte toute la complexité et la multitude des grilles de jeu possibles, il va falloir avoir recours à une technique beaucoup plus avancée : l'apprentissage profond (*reinforcement learning*) et les réseaux de neurones.

Troisième partie

Optimisation par reinforcement learning

Chapitre 7

PROCESSUS DE DÉCISION MARKOVIEN

1 Modélisation

On considère un système formé d'un agent $\mathscr{G}$ et d'un environnement $\mathscr{E}$ sur lequel $\mathscr{G}$ agit. L'ensemble des états possibles de $\mathscr{E}$ est un ensemble fini $\mathscr{S}$ et l'ensemble des actions possibles pour l'agent est un ensemble fini $\mathscr{A}$. Ces actions peuvent être différentes selon les états : on notera, pour tout $s \in \mathscr{S}$, $\mathscr{A}(s)$ l'ensemble des actions effectivement possibles pour l'agent lorsque le système est à l'état s. On a donc $\mathscr{A} = \bigcup_{s\in\mathscr{S}} \mathscr{A}(s)$.
L'environnement est également muni de deux fonctions :

1) une *fonction de transition probabiliste* $T : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to [0,1]$ telles que, pour tout $(s,a,s') \in \mathscr{S} \times \mathscr{A} \times \mathscr{S}$, $T(s,a,s')$ représente la probabilité que l'environnent passe de l'état s à l'état s' lorsque l'agent effectue l'action a. Autrement dit, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$, $s' \mapsto T(s,a,s')$ est une probabilité sur $\mathscr{S}$. Notons que cette fonction T est constante au cours du temps.

2) une *fonction de récompense* $R : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$: pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$, $R(s,a)$ représente la récompense (qui peut être positive ou négative) attribuée à l'agent lorsqu'il effectue l'action a alors que l'environnement est dans l'état s. On remarquera que comme $\mathscr{S} \times \mathscr{A}$ est fini, la fonction R est bornée. Comme la fonction T, la fonction R est constante au cours du temps.

La quadruplet $(\mathscr{S},\mathscr{A},T,R)$ forme ce qu'on appelle un *processus de décision markovien.*

Remarque. — Nous nous plaçons ici dans le cas où la fonction R ne dépend que de l'état présent s et de l'action choisie a i.e. $R(s,a)$ est indépendant du nouvel état s'. Dans certain processus de décision markovien, il est nécessaire de considérer une fonction récompense $R : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$ telle que, pour tout $(s,a,s') \in \mathscr{S} \times \mathscr{A} \times \mathscr{S}$, $R(s,a,s')$ représente la récompense reçue par l'agent lorsque l'environnement passe de l'état s à l'état s' à la suite de l'action a.

Initialement, l'environnement $\mathscr{E}$ se trouve dans un certain état $s_0 \in \mathscr{S}$. L'agent effectue une action $a_0 \in \mathscr{A}(s_0)$ et l'environnement passe dans un certain état s_1 avec une probabilité $T(s_0,a_0,s_1)$, puis une action $a_1 \in \mathscr{A}(s_1)$ et l'environnement passe dans l'état s_2 avec une probabilité $T(s_1,a_1,s_2)$, etc.
On obtient ainsi une suite de variables aléatoires d'états $(S_n) \in \mathscr{S}^{\mathbb{N}}$ et une suite de variables aléatoires d'actions $(A_n) \in \mathscr{A}^{\mathbb{N}}$.

1.1 Politiques

Une politique déterministe π est une fonction $\pi : \mathscr{S} \to \mathscr{A}$ qui à chaque état $s \in \mathscr{S}$ associe une action $\pi(s) \in \mathscr{A}$.
Une politique probabiliste π est une fonction $\pi : \mathscr{S} \times \mathscr{A} \to [0,1]$ telle que, pour tout $s \in \mathscr{S}$, la fonction $a \mapsto \pi(s,a)$ est une probabilité sur $\mathscr{A}$. Dans la suite, on note $\pi(a|s)$ (plutôt que $\pi(s,a)$) l'image de (s,a) par cette fonction π.
On ne considère ici que des politiques stationnaires i.e. qui n'évoluent pas au cours du temps : elles sont identiques à chaque instant n.

Si π est une politique déterministe, on dit que l'agent suit la politique π si, pour tout $n \in \mathbb{N}$, $A_n = \pi(S_n)$. Si π est une politique probabiliste, on dit que l'agent suit la politique π si, pour tout $n \in \mathbb{N}$, l'agent choisi aléatoirement l'action A_n à partir de l'état S_n de telle sorte que, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$, la probabilité que $A_n = a$ sachant que $S_n = s$ est égale à $\pi(a|s)$. Autrement dit, l'ensemble $\mathscr{S} \times \mathscr{A}$ est muni d'une probabilité $\mathbb{P}_\pi$ telle que, pour tout $n \in \mathbb{N}$, $\mathbb{P}_\pi(A_n = a \mid S_n = s) = \pi(a|s)$.
Notons qu'une politique déterministe π peut toujours être vue comme une politique probabiliste $\tilde{\pi}$ en posant, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$, $\tilde{\pi}(a|s) = 1$ si $a = \pi(s)$ et $\tilde{\pi}(a|s) = 0$ sinon. Dans la suite, on identifiera π et $\tilde{\pi}$ et on supposera, sauf mention du contraire, que les politiques sont probabilistes.
Si l'agent suit une politique π alors, pour tout $(s,s') \in \mathscr{S}^2$, la probabilité de $(S_{n+1} = s')$ sachant $(S_n = s)$ est indépendante de n car

$$\mathbb{P}_\pi(S_{n+1} = s' \mid S_n = s) = \sum_{a \in \mathscr{A}} T(s,a,s') \mathbb{P}_\pi(A_n = a \mid S_n = s) = \sum_{a \in \mathscr{A}} T(s,a,s') \pi(a|s). \tag{7.1}$$

Ainsi, dans ce cas, la suite des variables aléatoires (S_n) définit une chaîne de Markov sur l'espace d'états $\mathscr{S}$ dont les probabilités de transition sont données (7.1) pour tout couple $(s,s') \in \mathscr{S}^2$.

1.2 Récompenses pondérées

Supposons que l'agent suive une politique π. Alors, pour tout $n \in \mathbb{N}$, la récompense à l'instant $n+1$ est $R_{n+1} := R(S_n, A_n)$: il s'agit donc de la récompense reçue par l'agent pour avoir choisi l'action A_n alors que l'environnement est dans l'état S_n.
Dès lors, pour $(n,m) \in \mathbb{N}^2$ tel que $n < m$, la récompense totale obtenue entre les instants $n+1$ et m est $\sum_{k=n+1}^{m} R_k$. Lorsque m tend vers $+\infty$, cette somme n'a pas de raison de converger. On introduit alors un réel $\gamma \in [0;1]$ appelé *facteur d'actualisation* et on définit la récompense totale pondérée à partir de l'instant $n+1$ par

$$G_n = \sum_{k=n+1}^{+\infty} \gamma^{n+1-k} R_k = \sum_{k=0}^{+\infty} \gamma^k R_{n+1+k}. \tag{7.2}$$

Cette série est bien convergente car la suite de variables aléatoires (R_n) est bornée (puisque la fonction R est bornée) donc $\gamma^k R_{n+1+k} = O(\gamma^k)$ qui est le terme général d'une série géométrique convergente car $0 \leqslant \gamma < 1$.

On va chercher s'il existe une politique qui maximise, pour tout $n \in \mathbb{N}$, l'espérance de G_n sachant qu'à l'instant n, $S_n = s$ et $A_n = a$ pour un certain $(s,a) \in \mathscr{S} \times \mathscr{A}$. Puisque cette espérance dépend de la politique π et des états initiaux s et a, on la notera

$$Q_\pi(s,a) := \mathbb{E}_\pi [G_n \mid (S_n, A_n) = (s,a)] \tag{7.3}$$

On remarquera que la notation $Q_\pi(s,a)$ ne fait pas intervenir n ce qui est légitime puisque la loi conditionnelle de G_n sachant $(S_n, A_n) = (s,a)$ ne dépend en fait que de T, R et π et pas de n.

2 Équations de Bellman pour les fonctions de valeur

2.1 Fonctions de valeur

On fixe une politique π.

La fonction Q_π définie par (7.3) est appelée *fonction de valeur état-action.* On définit également sur $\mathscr{S}$ la *fonction de valeur état* V_π par

$$\forall s \in \mathscr{S}, \quad V_\pi(s) = \mathbb{E}_\pi[G_n \mid S_n = s]$$

Comme dans le cas de la fonction Q_π, cette fonction ne dépend pas de n mais uniquement de s, T, R et π.

Les deux fonctions de valeur V_π et Q_π sont liées par la relation suivante :

$$\forall s \in \mathscr{S}, \quad V_\pi(s) = \sum_{a \in \mathscr{A}} \pi(a|s) Q_\pi(s,a). \tag{7.4}$$

En effet, si $s \in \mathscr{S}$,

$$\begin{aligned} V_\pi(s) = \mathbb{E}_\pi[G_n \mid S_n = s] &= \sum_{a \in \mathscr{A}} \mathbb{P}_\pi(A_n = a \mid S_n = s) \mathbb{E}_\pi[G_n \mid (S_n = s) \cap (A_n = a)] \\ &= \sum_{a \in \mathscr{A}} \pi(a|s) \mathbb{E}_\pi[G_n \mid (S_n, A_n) = (s,a)] \\ &= \sum_{a \in \mathscr{A}} \pi(a|s) Q_\pi(s,a). \end{aligned}$$

2.2 Équations de Bellman

Considérons une politique π et revenons à la récompense pondérée définie par (7.3). Pour tout $n \in \mathbb{N}$,

$$G_n = \sum_{k=0}^{+\infty} \gamma^k R_{n+1+k} = R_{n+1} + \sum_{k=1}^{+\infty} \gamma^k R_{n+1+k} = R_{n+1} + \gamma \sum_{k=1}^{+\infty} \gamma^{k-1} R_{n+1+k} = R_{n+1} + \gamma \sum_{k=0}^{+\infty} \gamma^k R_{n+2+k}$$

i.e.

$$G_n = R_{n+1} + \gamma G_{n+1}.$$

Dès lors, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$,

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_{n+1} + \gamma G_{n+1} | (S_n, A_n) = (s,a)].$$

Par linéarité de l'espérance conditionnelle, il s'ensuit que

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_{n+1} | (S_n, A_n) = (s,a)] + \gamma \mathbb{E}_\pi[G_{n+1} | (S_n, A_n) = (s,a)].$$

Comme $R_{n+1} = R(S_n, A_n)$, si $(S_n, A_n) = (s,a)$ alors R_{n+1} est constante égale à $R(a,b)$ donc $\mathbb{E}_\pi[R_{n+1} | (S_n, A_n) = (s,a)] = R(s,a)$. De plus,

$$\mathbb{E}_\pi[G_{n+1} | (S_n, A_n) = (s,a)] = \sum_{s' \in \mathscr{S}} T(s,a,s') \mathbb{E}_\pi[G_{n+1} | ((S_n, A_n) = (s,a)) \cap (S_{n+1} = s')]$$

Or, la loi de G_{n+1} ne dépend que de S_{n+1} et de la politique π donc

$$\mathbb{E}_\pi[G_{n+1} | (S_n, A_n) = (s,a)] = \sum_{s' \in \mathscr{S}} T(s,a,s') \mathbb{E}_\pi[G_{n+1} | S_{n+1} = s']$$

et donc, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$,

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathscr{S}} T(s,a,s') V_\pi(s') \tag{7.5}$$

À l'aide de (7.4), on en déduit que, pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$,

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathscr{S}} T(s,a,s') \sum_{a' \in \mathscr{A}} \pi(a'|s') Q_\pi(s',a') \tag{7.6}$$

Les égalités (7.5) et (7.6) sont connues sous le nom d'*équations de Bellman* pour la fonction Q_π.

2.3 Politiques optimales

Soit Π l'ensemble des politiques sur $\mathscr{S}$. On définit sur Π un ordre partiel $\leqslant$ par :

$$\pi \leqslant \pi' \text{ si, pour tout } s \in \mathscr{S},\ V_\pi(s) \leqslant V_{\pi'}(s)$$

On dit qu'une politique π_* est optimale si, pour tout $\pi \in \Pi$, $\pi \leqslant \pi_*$. Autrement dit, π_* est optimale si, pour toute politique π et tout état s, $V_\pi(s) \leqslant V_{\pi_*}(s)$

Il existe toujours au moins une politique optimale mais celle-ci n'est en général pas unique. Dans toute la suite, on admet cette existence et on note π_* une politique optimale.

Par définition, $V_{\pi_*} = \max_{\pi \in \Pi} V_\pi$ et, de plus, grâce à l'équation de Bellman (7.5), $Q_{\pi_*} = \max_{\pi \in \Pi} Q_\pi$. Ainsi, V_{π_*} et Q_{π_*} ne dépendent pas de la politique maximale choisie. On les notera V_* et Q_*. Par définition, on a donc

$$V_* = \max_{\pi \in \Pi} V_\pi \qquad \text{et} \qquad Q_* = \max_{\pi \in \Pi} Q_\pi.$$

Sachant qu'il existe une politique optimale, on peut montrer qu'il existe un politique optimale déterministe. En effet, considérons la politique $\pi_{\max}$ définie de la manière suivante : pour tout $s \in \mathscr{S}$, on choisit un élément $a_s \in \mathscr{A}$ tel que $Q_*(s,a_s) = \max_{\alpha\in\mathscr{A}} Q_*(s,\alpha)$ (ce qui est possible par $\mathscr{A}$ est fini) et on pose

$$\forall (s,a) \in \mathscr{S}\times\mathscr{A},\ \pi_{\max}(a|s) = \begin{cases} 1 & \text{si } a = a_s \\ 0 & \text{sinon} \end{cases}.$$

Alors, d'après (7.4), pour tout $s \in \mathscr{S}$,

$$V_*(s) = \sum_{a\in\mathscr{A}} \pi_*(a|s)Q_*(s,a) \leqslant \sum_{a\in\mathscr{A}} \pi_*(a|s)Q_*(s,a_s) = Q_*(s,a_s)\underbrace{\sum_{a\in\mathscr{A}} \pi_*(a|s)}_{=1} = Q_*(s,a_s)$$

et

$$V_{\pi_{\max}}(s) = \sum_{a\in\mathscr{A}} \pi_{\max}(a|s)Q_{\pi_{\max}}(s,a) = Q_{\pi_{\max}}(s,a_s)$$

Or, la politique $\pi_{\max}$ consiste à choisir, quand l'environnement se trouve à l'état s, l'action a_s qui conduit à $Q_*(s,a_s)$ donc $Q_{\pi_{\max}}(s,a_s) = Q_*(s,a_s)$ et ainsi

$$V_*(s) \leqslant Q_*(s,a_s) = Q_{\pi_{\max}}(s,a_s) = V_{\pi_{\max}}(s)$$

Par maximalité de V_*, on en déduit que $V_{\pi_{\max}} = V_*$ donc $\pi_{\max}$ est bien une politique déterministe optimale.

2.4 Équations d'optimalité de Bellman

Comme $V_* = V_{\pi_{\max}} = Q_{\pi_{\max}}(s,a_s)$ et $Q_{\pi_{\max}}(s,a_s) = Q_*(s,a_s) = \max_{\alpha\in\mathscr{A}} Q_*(s,\alpha)$, on obtient l'équation d'optimalité de Bellman permettant d'exprimer V_* en fonction de Q_* :

$$\forall s \in \mathscr{S}, \qquad V_*(s) = \max_{\alpha\in\mathscr{A}} Q_*(s,\alpha) \tag{7.7}$$

Or, l'équation de Bellman (7.5) permet d'exprimer Q_* en fonction de V_* :

$$\forall (s,a) \in \mathscr{S}\times\mathscr{A}, \quad Q_*(s,a) = R(s,a) + \gamma \sum_{s'\in\mathscr{S}} T(s,a,s')V_*(s') \tag{7.8}$$

On en déduit que

$$\forall (s,a) \in \mathscr{S}\times\mathscr{A}, \quad Q_*(s,a) = R(s,a) + \gamma \sum_{s'\in\mathscr{S}} T(s,a,s') \max_{\alpha\in\mathscr{A}} Q_*(s',\alpha).$$

Sachant que, pour tout $(a,s) \in \mathscr{A}\times\mathscr{S}$, $\sum_{s'\in\mathscr{S}} T(s,a,s') = 1$ puisque $s' \mapsto T(s,a,s')$ est une probabilité sur $\mathscr{S}$, on conclut que

$$\forall (s,a) \in \mathscr{S}\times\mathscr{A}, \quad Q_*(s,a) = \sum_{s'\in\mathscr{S}} T(s,a,s')\left[R(s,a) + \gamma \max_{\alpha\in\mathscr{A}} Q_*(s',\alpha)\right] \tag{7.9}$$

L'égalité (7.9) est *l'équation d'optimalité de Bellman pour* Q_*.

Chapitre 8

Q-LEARNING PAR ITÉRATIONS DES FONCTIONS DE VALEUR

1 Principe général

On souhaite déterminer (ou au moins approcher) une politique π_* optimale i.e. on souhaite déterminer une fonction de valeur état-action aussi proche que possible de Q_*. Pour cela, on considère une fonction Q quelconque sur $\mathscr{S} \times \mathscr{A}$, par exemple la fonction nulle. Notons p le nombre d'états et q le nombre d'actions et énumérons les ensembles $\mathscr{S}$ et $\mathscr{A}$:

$$\mathscr{S} = \{s_1, s_2, ..., s_p\} \qquad \text{et} \qquad \mathscr{A} = \{a_1, a_2, ..., a_q\}.$$

Se donner une fonction Q revient à se donner une matrice $\mathtt{Q}$ de taille $p \times q$ telle que, pour tout $(i,j) \in [\![1,p]\!] \times [\![1,q]\!]$, $Q(s_i, a_j) = \mathtt{Q}_{i,j}$. Ainsi, si initialement Q est le fonction nulle alors $\mathtt{Q}$ la matrice nulle $O_{p,q}$.
On va alors modifier de façon itérative la matrice $\mathtt{Q}$ de la manière suivante : si l'environnement se trouve dans l'état s_k, on choisit l'action a_ℓ qui maximise Q i.e. on choisit l'indice ℓ tel que $\mathtt{Q}_{k,\ell}$ soit le plus grand élément de la ligne k de $\mathtt{Q}$ (si plusieurs valeurs de ℓ sont possibles, on en prend une quelconque, par exemple, la plus petite).
On effectue alors l'action a_ℓ et on note $r_{k,\ell} = R(s_k, a_\ell)$ la récompense obtenue et s_m l'état de l'environnement à l'issue de cette action.
On détermine ensuite le plus grand élément e_{max} dans la ligne m de $\mathtt{Q}$ i.e. $e_{max} = \max\limits_{\alpha \in \mathscr{A}} Q(s_m, \alpha)$ et on calcule $r + \gamma e_{max}$ i.e. $R(s_k, a_\ell) + \gamma \max\limits_{\alpha \in \mathscr{A}} Q(s_m, \alpha)$.
On introduit un réel $\alpha \in]0\,;1[$ appelé *facteur d'apprentissage* (qui pourra évoluer au cours de temps), et on va mettre à jour la matrice $\mathtt{Q}$ en remplaçant l'élément d'indice (k,ℓ) par

$$(1-\alpha)\mathtt{Q}_{k,\ell} + \alpha(r + \gamma e_{max}).$$

Autrement dit, on fait la substitution

$$\mathtt{Q}_{k,\ell} \leftarrow (1-\alpha)\mathtt{Q}_{k,\ell} + \alpha(r_{k,\ell} + \gamma \max_{j \in [\![1,q]\!]} \mathtt{Q}_{m,j})$$

soit, en termes de fonction,

$$Q(s_k, a_\ell) \leftarrow (1-\alpha)Q(s_k, a_\ell) + \alpha(R(s_k, a_\ell) + \gamma \max_{\alpha \in \mathscr{A}} Q(s_m, \alpha)).$$

À ce niveau, deux problèmes se posent :

1) si on prend initialement la matrice nulle, il n'y aura pas de choix pertinent pour démarrer car tous les coefficients seront égaux.

   On pourrait imaginer remplacer la matrice nulle par une matrice aléatoire quelconque. Cependant, se poserait alors un second problème

2) Si on réitère le procédé précédent, l'agent agira de façon uniquement déterministe à partir des données de la matrice `Q`. De façon plus concrète, il n'évoluera qu'en fonction de son expérience et n'essayera pas d'actions aléatoires nouvelles qui pourraient donner de meilleurs résultats. De plus, il y a un risque dans ce cas que l'agent se contente de récompense « minimale » et stagne dans une sorte de maximum local plutôt que de chercher à maximiser Q sur l'ensemble des états et des actions possibles.

2 Compromis exploitation/exploration

Pour palier ce problème, on va introduire un facteur permettant de trouver un compromis entre l'exploitation de l'expérience accumulée à travers l'actualisation de `Q` et l'exploration de nouvelles actions non déterminées par la matrice `Q`.
Pour se faire, on se donne un réel $\varepsilon \in [0\,;1]$, appelé *facteur d'exploration* et destiné à évoluer au cours du temps et, à chaque instant n, au lieu de choisir systématiquement l'action a_ℓ qui maximise la ligne k de la matrice `Q`, on choisit cette action avec une probabilité $1-\varepsilon$ et on choisit une action aléatoire avec une probabilité ε.
À l'instant 0, on prend $\varepsilon = 1$ i.e. la première action est choisie de façon totalement aléatoire (ce qui compense le fait que la matrice `Q` est initialisée à $O_{p,q}$) et, ensuite, on va faire progressivement diminuer ε pour tenir compte du fait que plus le temps avance, plus l'agent à des données à exploiter issue de son expérience passée et moins il a intérêt à explorer de nouvelles actions. On peut, par exemple, utiliser une décroissance exponentielle de la forme

$$\varepsilon_N = \varepsilon_{\min} + (\varepsilon_{\max} - \varepsilon_{\min})\mathrm{e}^{-\tau N}$$

où ε_N est le facteur d'exploration au temps N, $\varepsilon_{\min}$, $\varepsilon_{\max}$ et τ sont des paramètres représentant respectivement le facteur d'exploration minimum, le facteur d'exploration maximum et le taux de décroissance. En prenant $\varepsilon_{\max} = 1$, on aura en particulier, $\varepsilon_0 = 1$.

3 Algorithme

Pour simplifier l'écriture, les états seront notés état 1, état 2, ..., état p au lieu de s_1, s_2, ..., s_p et les actions seront notées action 1, action 2, ..., action q au lieu de a_1, a_2, ..., a_q.
De plus, l'état 1 est considérée comme l'état initial de tout épisode et l'état p comme l'état final de tout épisode.

```
Paramètres: facteur_actualisation, facteur_apprentissage, taux_decroissance,
              facteur_exploration_min, facteur_exploration_max
temps = 0
Initialiser la matrice Q à la matrice nulle de taille pxq
k = 1
Tant que k!= p :
     facteur_exploration = facteur_exploration_min +
            (facteur_exploration_max - facteur_exploration_min)*
                        exp(-taux_decroissance*temps)
     Choisir un nombre réel x aléatoire entre 0 et 1
     Si x < facteur_exploration :
          Choisir une action l au hasard
     Sinon :
          Déterminer l'indice l d'un élément maximal de la ligne k de Q
     Effectuer l'action l
     Déterminer la récompense r associée à k et l
     Déterminer le nouvel état k'
     Déterminer le plus grand élément e_max de la ligne k' de Q
     Q[k,l] = (1 - facteur_apprentissage)*Q[k,l] +
           facteur_apprentissage*(r + facteur_actualisation*e_max)
     k = k'
     temps = temps + 1
```

4 Implémentation pour Tétris

4.1 Limitations

Pour Tétris, les états sont a priori toutes les configurations possibles de la grille (y compris le tétramino en cours) soit $2^{10\times 22} \approx 1{,}7 \cdot 10^{66}$ états. Ceci est donc totalement impossible à gérer (et même à stocker) sur une machine.

Dans un premier temps, on peut dissocier le tétramino en cours des pièces accumulées en base de la grille et ne considérer que la hauteur de chacune des 10 colonnes (sans se soucier des trous) mais cela laisse encore 7×10^{22} états possibles.
Bdolah & Livnat [1] ont proposé (sur une version simplifiée à 6 colonnes et en utilisant 5 pièces comportant au plus 2×2 blocs) la simplification suivante :

- on ne considère que les écarts entre les hauteurs des colonnes successives ;
- on borne ces écarts par la longueur de la plus grande pièce (c'est-à-dire tout écart supérieur à 4 (resp. inférieur à -4) est remplacé par 4 (resp. -4)).

Ainsi, avec leur version simplifiée, ils avaient 5 écarts possibles et ces écarts prenaient leurs valeurs dans $\{-2,-1,0,1,2\}$ (car la plus grande pièce était un carré de côté 2) ce qui leur donnait $5^5 = 3125$ états possibles (pour les seules colonnes).

En adaptant cette approche à la version standard de Tétris, cela donne $7 \times 9^9 \approx 2{,}7 \cdot 10^9$ états possibles pour l'environnement. Cela parait difficilement gérable...

En conséquence, nous avons décidé de n'implémenter l'algorithme précédent non pas sur l'ensemble des états mais seulement sur un échantillon aléatoire de couples état-action obtenus lors d'une phase d'entraînement (en s'inspirant de la notion de mémoire de reprise – voir le chapitre suivant).
Ainsi, on fait fonctionner l'algorithme sur un nombre déterminé d'épisodes et on crée une $Q-$table contenant les couples état-action rencontrés. Ensuite, lors de la phase de jeu, l'agent joue selon la $Q-$table s'il est dans une configuration qui y figure et de manière aléatoire sinon.

Cette limitation ne peut être efficace avec le jeu de Tétris usuel. Nous ne l'avons fait fonctionner qu'avec un jeu d'un seul domino (au lieu des 7 tétraminos) sur une grille de taille modeste.

4.2 Les classes QRLOptimizer et TetrisEnv

4.2.1 La classe QRLOptimizer

L'optimisation par reinforcement learning est implémenté par la classe `QRLOptimizer`.
Celle-ci prend en paramètres notamment :

- `width` : largeur de la grille ;
- `height` : hauteur de la grille ;
- `alpha` : facteur d'apprentissage α ;
- `gamma` : facteur d'actualisation γ ;
- `epsilon_min` : valeur minimale du facteur d'exploration $\varepsilon_{\min}$;
- `epsilon_max` : valeur maximale du facteur d'exploration $\varepsilon_{\max}$;
- `epsilon_delta` : valeur de décrémentation du facteur d'exploration τ ;
- `max_episodes` : nombre maximal d'épisodes lors de l'entraînement ;
- `max_blocks` : nombre maximal de blocs lors d'un épisode.

4.2.2 La classe TetrisEnv

Pour l'implémentation du Q-Learning, la classe `QRLOptimizer` utilise un environnement dédié à travers la classe `TetrisEnv`.
Cet environnement définit notamment l'ensemble des actions

```
'L', 'R', 'D', 'H', 'T', 'N'
```

au paragraphe **4.4** du chapitre 3 (p. 20). Il permet également de récupérer l'état de la grille grâce à la méthode `getStateCode(self)`. Cette dernière fait elle-même appel à la méthode `encodeToInt(self)` de la classe `Board` qui renvoie une représentation de la grille sous forme d'un nombre entier binaire.

La classe `TetrisEnv` définit également la fonction récompense (`reward`) : nous avons choisi pour celle-ci le carré du nombre de lignes supprimées à la suite de l'action effectuée.

4.2.3 Apprentissage

L'apprentissage se fait en jouant un certain nombre de parties et en remplissant une $Q-$table qui prend la forme d'un dictionnaire dont les clés sont les états et les valeurs associées sont les valeurs de la fonction Q pour les différentes actions possibles.
Au cours des parties successives d'apprentissage, soit l'état est déjà présent et alors il est mis à jour grâce à l'équation de Bellamn soit il ne figure pas encore dans le dictionnaire et, dans ce cas, on le rajoute en initialisant les coefficients des différentes actions à 0.
La méthode `learn(self)` prend en charge la réalisation de cet apprentissage en mettant en œuvre le principe du compromis exploitation/exploration et en mettant à jour la $Q-$table via la méthode `update(self, s, a)` qui implémente l'équation de Bellman.
Une fois l'apprentissage terminé, la méthode `printQIndexes(self)` permet de visualiser la fréquence des cellules apparues lors de l'entraînement à l'aide d'une coloration en niveau de gris sur la grille.

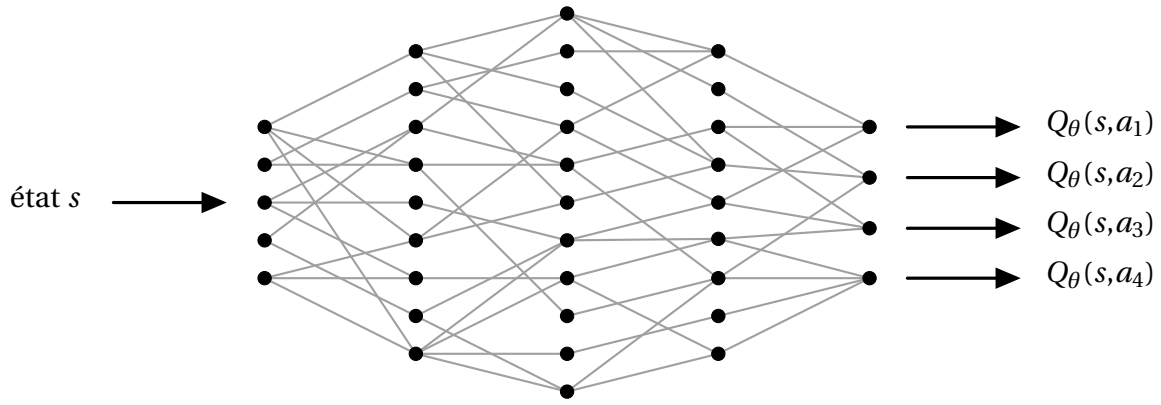
4.2.4 Phase de jeu

La méthode `play(self)` prend en charge la phase de jeu à l'aide de la $Q-$table. Si l'état rencontré lors de la phase de jeu est apparu lors de la phase d'entraînement alors l'agent joue selon la $Q-$table et, sinon, il joue de façon aléatoire.

Chapitre 9

VERS LE DEEP-Q-LEARNING

Dans l'algorithme par itération des fonctions de valeur, on construit une suite d'approximations successives (Q_k) qui converge vers Q_*. Concrètement, les Q_k sont implémentées sous la forme de matrices $p \times q$ où p est le nombre d'états et q le nombre d'actions. Lorsque p devient très grand (comme c'est le cas pour Tétris, même en simplifiant à l'extrême l'ensemble des états), cela devient à la fois ingérable du point de vue de la mémoire et inefficace du point de vue du temps de calcul.
Pour contourner ce problème, il est possible d'utiliser un réseau de neurones pour calculer des approximations de Q_*. Plus précisément, on peut construire un réseau de neurones pondéré par un ensemble de poids θ qui prend en entrée l'état s de l'environnement et qui calcule en sortie une valeur approchée de $Q_*(s,a)$ pour chaque action a possible. Cette valeur dépend de s, de a et de θ : nous la noterons $Q_\theta(s,a)$.
Schématiquement, si on encode l'état s sur 5 neurones en entrée et s'il y a 4 actions possibles pour chaque état, on obtient un réseau du type suivant.



Partant d'un réseau muni d'un ensemble de poids θ aléatoire, le but va être d'entraîner ce réseau de façon à modifier θ de telle sorte que, pour tout couple état-action (s,a), l'écart entre la valeur $Q_\theta(s,a)$ calculée par le réseau et la valeur $Q_*(s,a)$ optimale soit le plus petit possible.
Lorsqu'on utilise un réseau de neurones pour faire, par exemple, de la reconnaissance de caractère, l'entraînement est simple à définir : on fournit au réseau une série d'images dont on sait, à l'avance, ce qu'elles représentent et on entraîne le réseau sur ces images. On sait alors explicitement calculer l'erreur commise entre la sortie du réseau et la valeur exacte attendue. Dans le cas des processus de décision markovien, on ne connaît pas la valeur de Q_* et, dès lors, on ne sait pas calculer, de façon exacte, l'erreur commise. Pour

contourner ce problème, on va chercher à minimiser cette erreur en utilisant l'équation d'optimalité de Bellman sur un stock d'expérience déjà réalisée.

1 Fonction d'erreur

Rappelons que, pour tout couple état-action $(s,a) \in \mathscr{S} \times \mathscr{A}$, $R(s,a)$ désigne la récompense obtenue après avoir effectué l'action a alors que l'environnement était dans l'état s. De plus, l'environnement passe alors dans un état s' avec une probabilité $T(s,a,s')$.
La fonction Q_* est caractérisée par l'équation de Bellman (7.9) :

$$\forall (s,a) \in \mathscr{S} \times \mathscr{A}, \quad Q_*(s,a) = \sum_{s' \in \mathscr{S}} T(s,a,s') \left[R(s,a) + \gamma \max_{\alpha \in \mathscr{A}} Q_*(s',\alpha) \right]$$

Autrement dit, étant donné un couple état-action (s,a), Q_* est l'unique fonction de valeur état-action Q qui minimise la fonction $E_{(s,a)}$ définie par

$$E_{(s,a)}(Q) = \left[Q(s,a) - \sum_{s' \in \mathscr{S}} T(s,a,s') \left[R(s,a) + \gamma \max_{\alpha \in \mathscr{A}} Q(s',\alpha) \right] \right]^2 \tag{9.1}$$

puisque Q_* est l'unique fonction de valeurs état-action telle que $E_{(s,a)}(Q_*) = 0$.
La fonction $E_{(s,a)}$ est appelée *fonction d'erreur* associée au couple (s,a). Elle sert à estimer si une fonction de valeur état-action Q est plus ou moins proche de Q_*. Plus $E_{(s,a)}$ est proche de 0, plus Q sera considérée comme proche de Q_*.
On va donc entraîner le réseau à minimiser $E_{(s,a)}$ pour tout $(s,a) \in \mathscr{S} \times \mathscr{A}$. Pour se faire, on va utiliser un échantillon aléatoire de valeurs issues d'un stock d'expériences conservées en mémoire.

2 Entraînement par mémoire de reprise

On fixe un entier $M > 0$ et entier $M_e \in [\![1,M]\!]$ (dans la pratique $M \gg M_e$).
On fait agir le réseau de neurones sur l'environnement (essentiellement comme dans le cas du Q-learning par itération décrit au paragraphe précédent – notamment en conservant le principe de compromis exploitation/exploration) et on stocke en mémoire les M dernières valeurs du quadruplet $(s, a, R(a,s), s')$. Ce stock est appelé la *mémoire de reprise* (*replay memory* en anglais).
Pour mettre à jour le réseau de neurones, on choisit un échantillon aléatoire de M_e éléments dans la mémoire de reprise. Pour chaque élément $(s_j, a_j, R(s_j,a_j), s'_j)$ de cet échantillon, on calcule, pour chaque action $a \in \mathscr{A}$, $Q_\theta(s_j,a)$ à l'aide du réseau de neurones puis on détermine $\max_{\alpha \in \mathscr{A}} Q_\theta(s_j,\alpha)$ et on en déduit la valeur de

$$y_j := R(s_j,a_j) + \gamma \max_{\alpha \in \mathscr{A}} Q_\theta(s'_j,\alpha).$$

On travaille ici avec un nouvel état s'_j déterminé donc, dans la somme de l'égalité (9.1), on considère que toutes les probabilités $T(s,a,s')$ sont nulles sauf pour $s' = s'_j$ pour lequel elle vaut 1.
On utilise enfin la méthode de descente de gradient pour mettre à jour le réseau de telle sorte à minimiser la fonction d'erreur

$$E_{(s_j,a_j)}(Q_\theta) = \left[Q_\theta(s_j,a_j) - y_j\right]^2$$

sur le lot de M_e valeurs de l'échantillon.

3 Stabilisation par duplication du réseau

Lors du calcul de l'erreur, on utilise le réseau de neurones à deux reprises : une première fois pour calculer $Q_\theta(s_j,a_j)$ et une seconde fois pour calculer y_j. Entre ces deux calculs, le réseau n'a pas évolué alors qu'on cherche plutôt à tirer partie des « progrès » du réseau pour estimer au mieux l'écart entre y_i et $Q_\theta(s_j,a_j)$. En utilisant le même réseau – ou même des réseaux trop proches (dans la suite des évolutions) – on constate une certaine instabilité du processus.
Pour palier ce problème, on introduit un réseau secondaire, initialement identique au réseau principal et dont les poids restent figés pendant un intervalle de temps t assez long puis sont régulièrement mis à jour, à intervalle régulier t, à l'aide des poids du réseau principal. Lors du calcul de $E_{(s_j,a_j)}(Q_\theta)$ vu précédemment, on utilisera en fait le réseau principal pour calculer $Q_\theta(s_j,a_j)$ mais on utilisera le réseau secondaire pour le calcul de y_j. Ainsi, si on note θ_0, θ_1, ..., θ_k, ... la suite des ensembles des poids du réseau de neurones principal aux instants 0, 1, ..., k, ..., on calcule la fonction d'erreur $E_{(s_j,a_j)}(Q_\theta)$ par la formule

$$E_{(s_j,a_j)}(Q_{\theta_{mt+k}}) = \left[Q_{\theta_{mt+k}}(s_j,a_j) - \left(R(s_j,a_j) + \gamma \max_{\alpha\in\mathscr{A}} Q_{\theta_{mt}}(s'_j,\alpha)\right)\right]^2$$

pour tout $m \in \mathbb{N}$ et tout $k \in [\![0,t-1]\!]$.

4 Algorithme

On aboutit à l'algorithme de la page suivante où l'on note :

- `etat_initial` l'état initial de l'environnement (début d'une partie) ;
- `etat_final` l'état final de l'environnement (fin d'une partie) ;
- `Res_P` le réseau principal ;
- `Res_S` le réseau secondaire ;
- `Res_P(s)_max` (resp. `Res_S(s)_max`) la plus grande valeur en sortie du réseau principal (resp. secondaire) lorsqu'on passe en entrée l'état `s` ;
- `Res_P(s,a)` (resp. `Res_S(s,a)`) la valeur obtenue pour l'action `a` en sortie du réseau principal (resp. secondaire) lorsqu'on passe en entrée l'état `s`.

```
Paramètres: facteur_actualisation, facteur_apprentissage
            facteur_exploration_min, facteur_exploration_max,
            taille_memoire, taille_echantillon,
            periode_actualisation_Res_S, nb_episodes_max
### constitution d'une mémoire de reprise aléatoire ###
Initialiser à 0 la liste memoire_de_reprise de taille taille_memoire
t = 0
Tant que t < taille_memoire :
   s = etat_initial
   Tant que s != etat_final :
      Effectuer une action a au hasard
      Déterminer la récompense r et le nouvel état s'
      memoire_de_reprise[t] = [s,a,r,s']
      t = t+1
### début de l'entrainement ###
Initialiser Res_P avec des poids aléatoires
Initialiser à 0 les poids de Res_S
temps_t = 0
delta_exploration = facteur_exploration_max - facteur_exploration_min
Pour k allant de 1 à nb_episodes_max :
   s = etat_initial
   Tant que s != etat_final :
      Si temps_t est un multiple de periode_actualisation_Res_S :
        Res_S = Res_P
      facteur_exploration = facteur_exploration_min +
              delta_exploration*exp(-taux_decroissance*temps_t)
      Choisir un nombre réel x aléatoire entre 0 et 1
      Si x < facteur_exploration :
        Choisir une action a au hasard
      Sinon :
        Déterminer une action a telle que Res_P(s,a) == Res_P(s)_max
      Effectuer l'action a et déterminer la récompense r et le nouvel état s'
      Pour j allant de 0 à taille_memoire-2 :
        memoire_de_reprise[j] = memoire_de_reprise[j+1]
      memoire_de_reprise[taille_memoire-1]=[s,a,r,s']
      Initialiser à 0 une liste y de longueur taille_echantillon
      Selectionner aléatoirement un échantillon de taille_echantillon
           éléments de memoire_de_reprise sous forme d'une liste E
      Pour j allant de 0 à taille_echantillon-1 :
        Si E[j][0] == etat_final :
          y[j] = E[j][2]
        sinon :
          y[j] = E[j][2] + facteur_actualisation * Res_S(E[j][3])_max
      Mettre à jour Res_P par descente de gradient pour la fonction d'erreur
    (Res_P(E[j][0],E[j][1])-y[j])^2 sur le lot de valeurs contenues dans y
    s = s'
    temps_t = temps_t + 1
```

Quatrième partie

Résultats et conclusion

Chapitre 10

RÉSULTATS

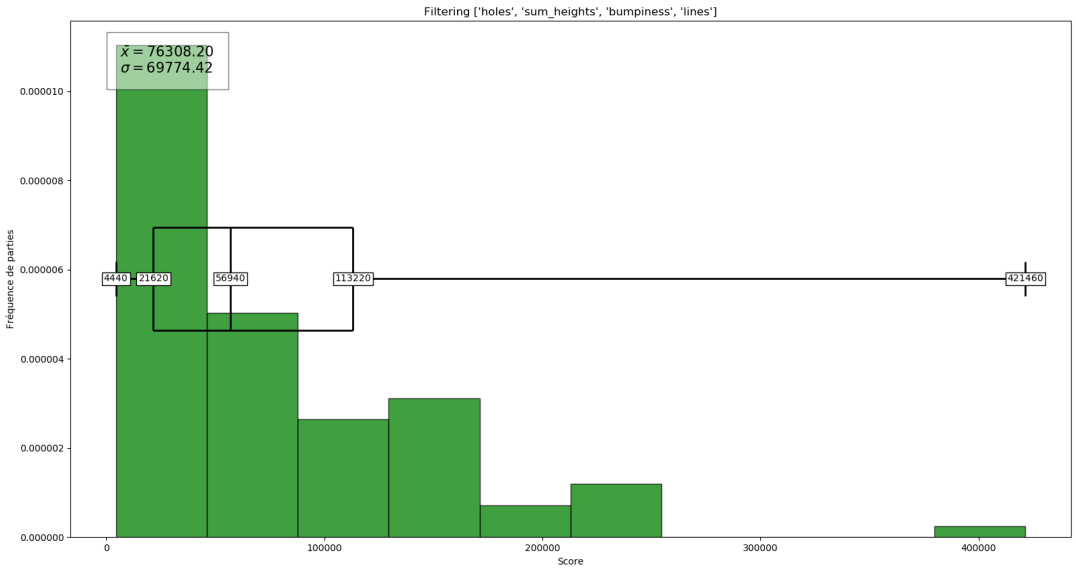
Les résultats suivants ont été obtenus à l'aide du module `stats.py`.
La figure donne :

- L'histogramme de répartition des scores
  (on rappelle que : 1 ligne → 40, 2 lignes→ 100, 3 lignes → 300 et 4 lignes → 1200)
- La moyenne et l'écart-type
- Le diagramme en boîte des résultats (minimum, Q_1, médiane, Q_3 et maximum)

Sauf indication contraire, on ne limite pas le nombre de blocs joués (ce qui fait que les tests ont été particulièrement longs à effectuer) et on teste sur 100 parties.

1 Agents par filtrage

Agent par filtrage en filtrant, dans l'ordre, les trous, la somme des hauteurs, le bumpiness et le nombre de lignes :



2 Algorithmes génétiques

Voici différents résultats d'optimisation par algorithmes génétiques en variant les différents paramètres.
Afin d'obtenir des résultats comparables et de ne pas avoir de temps de calculs trop longs, nous avons fixé les paramètres suivants dans toutes les optimisations :

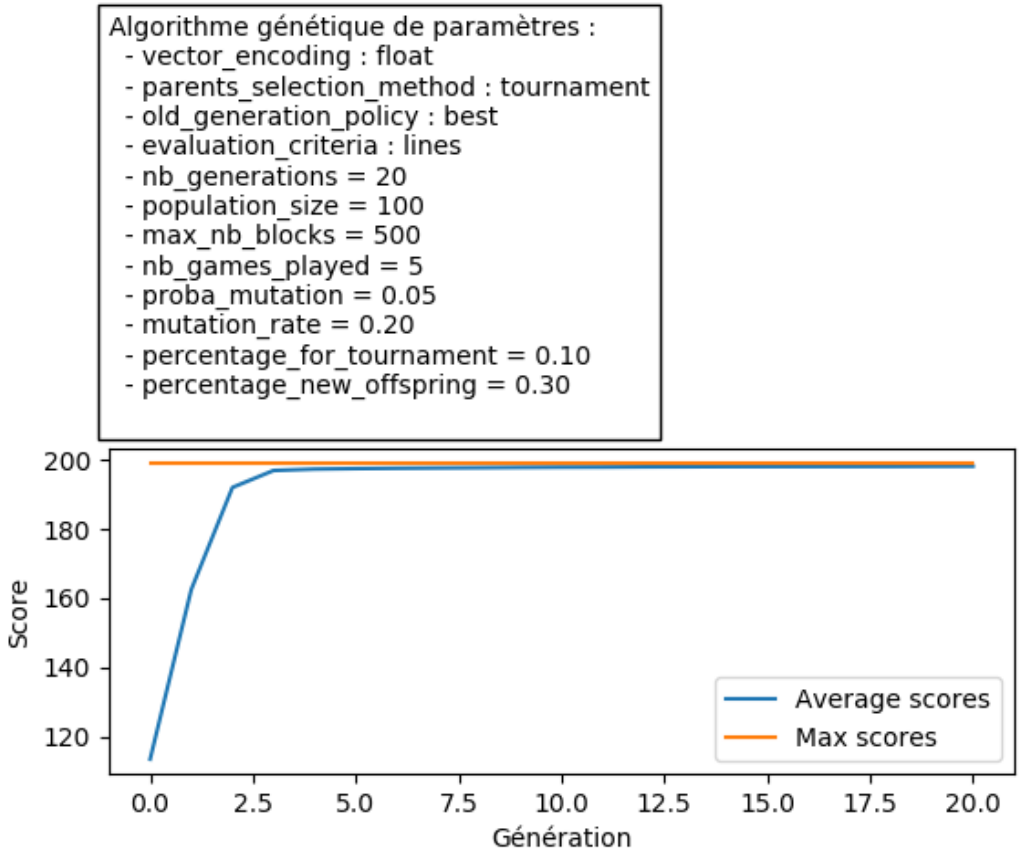
- Taille de la population : 100
- Nombre de générations : 20
- Nombre maximum de blocs joués : 500
- Nombre de parties pour l'évaluation : 5

2.1 Exemple 1

Paramètres :

- `vector_encoding : float`
- `parents_selection_method : tournament`
- `old_generation_policy : best`
- `evaluation_criteria : lines`
- `proba_mutation = 0.05`
- `mutation_rate = 0.20`
- `percentage_for_tournament = 0.10`
- `percentage_new_offspring = 0.30`

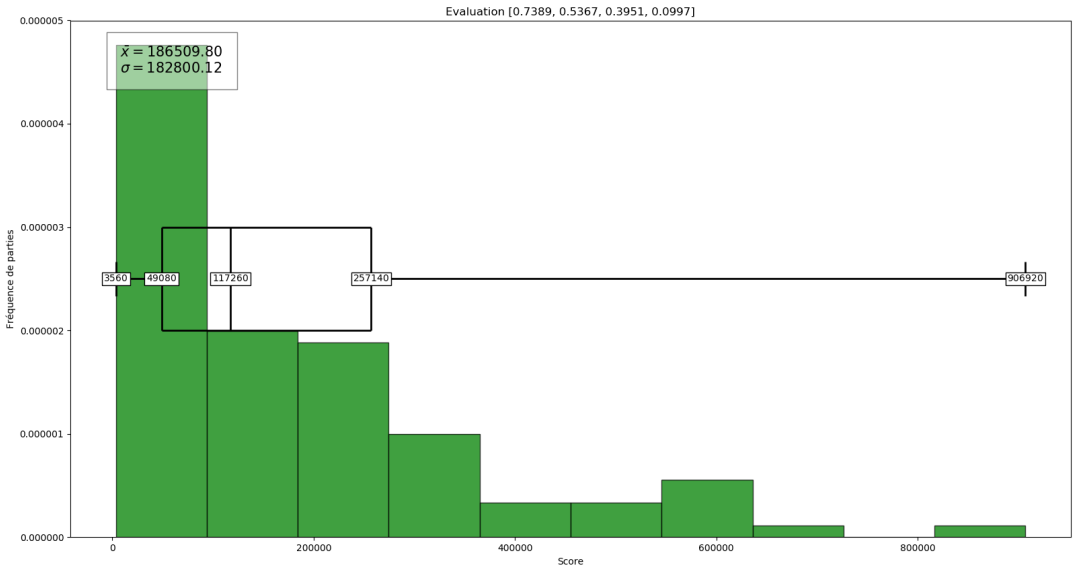
L'optimisation a duré à peu près 5 heures. Voici l'évolution de la population :



On voit que si la population a une tendance moyenne à fortement augmenter, le meilleur individu stagne. En fait ça a été le même durant toutes les générations.
En effet, l'évaluation des agents se faisant sur le nombre de lignes, il semble qu'un maximum de 200 lignes ne puisse pas vraiment être dépassé. Toutes les optimisations avec une évaluation sur les lignes ont donné à peu près le même type de résultat. C'est pourquoi dans les exemples suivants nous prendrons une évaluation sur les scores.

Le meilleur individu a pour coefficients : `[0.7389, 0.5367, 0.3951, 0.0997]`.

Regardons ses statistiques sur 100 parties :

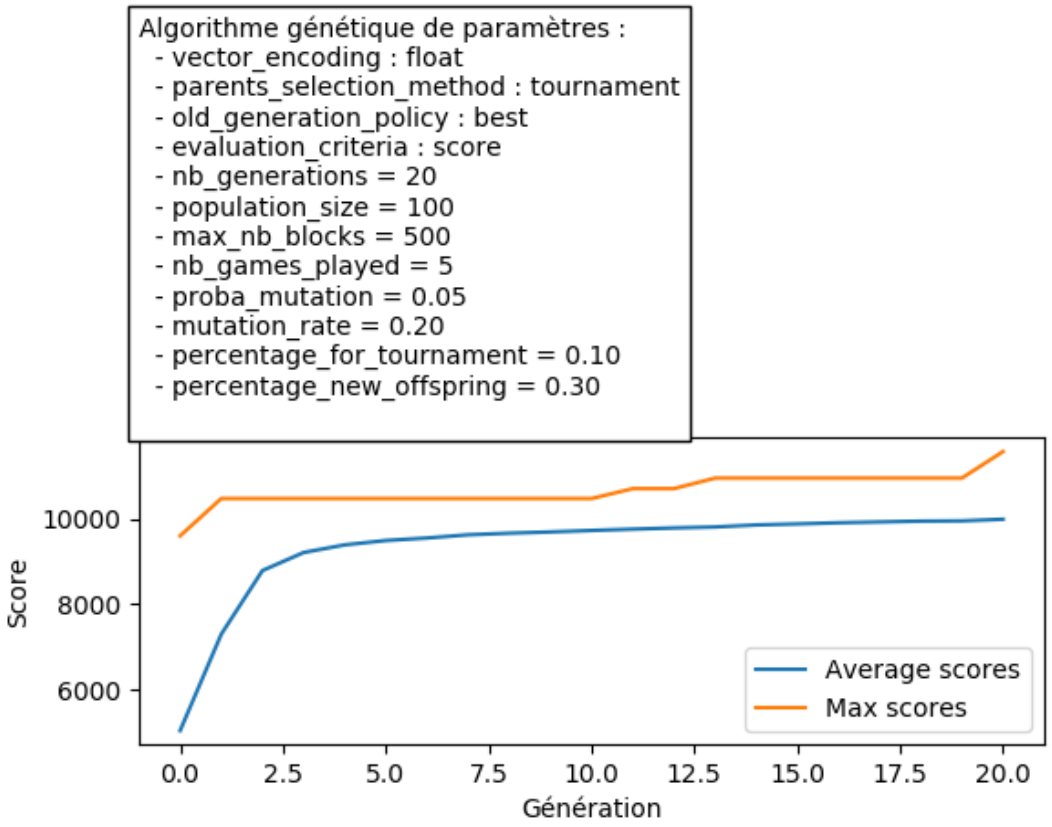


2.2 Exemple 2

Paramètres :

- `vector_encoding : float`
- `parents_selection_method : tournament`
- `old_generation_policy : best`
- `evaluation_criteria : scores`
- `proba_mutation = 0.05`
- `mutation_rate = 0.20`
- `percentage_for_tournament = 0.10`
- `percentage_new_offspring = 0.30`

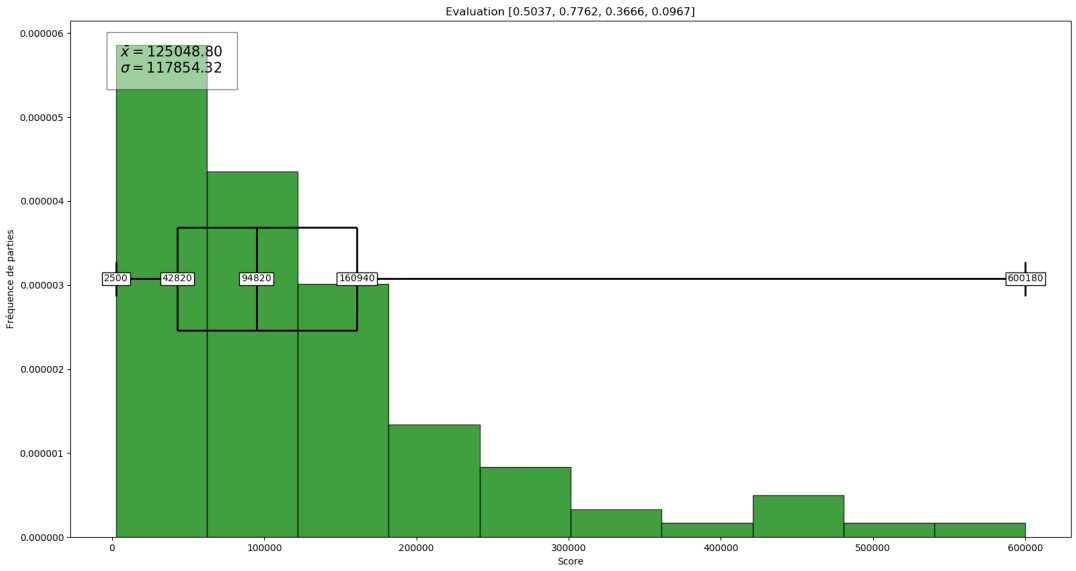
L'optimisation a duré à peu près 3 heures. Voici l'évolution de la population :



Ici la population a une tendance moyenne à fortement augmenter, et le meilleur individu augmente également.

Le meilleur individu a pour coefficients : `[0.5037, 0.7762, 0.3666, 0.0967]`.

Regardons ses statistiques sur 100 parties :

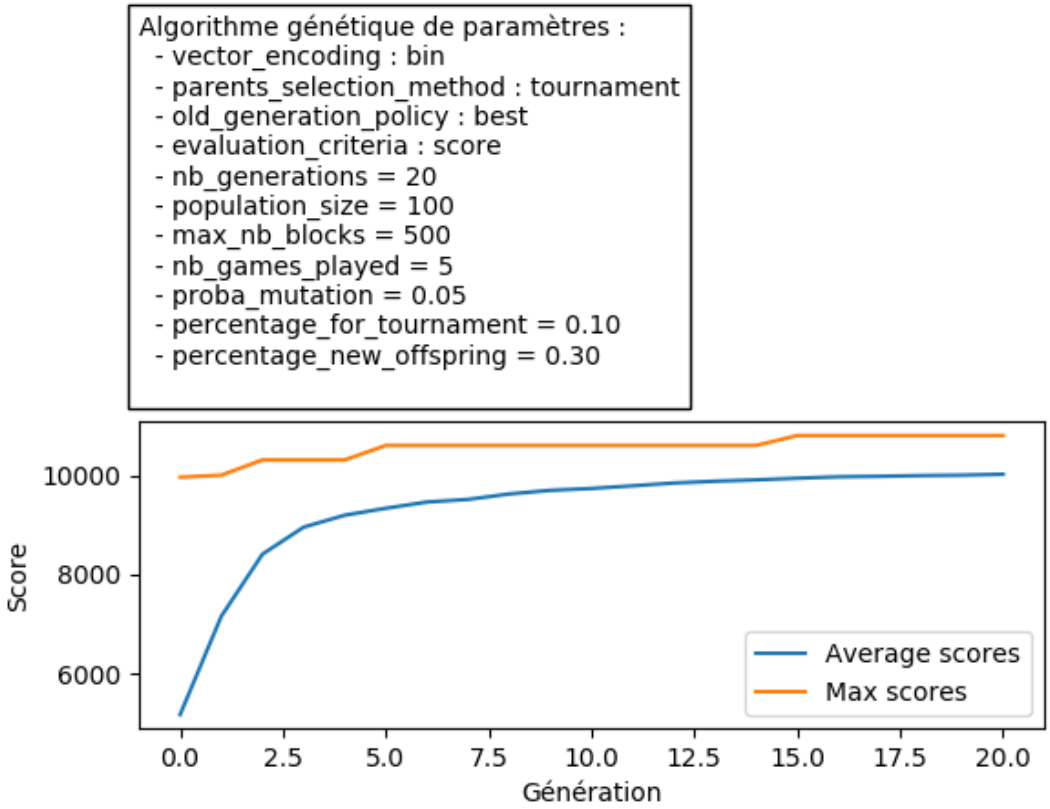


2.3 Exemple 3

Dans cet exemple nous avons simplement changé l'encodage des individus :
Paramètres :

- `vector_encoding : bin`
- `parents_selection_method : tournament`
- `old_generation_policy : best`
- `evaluation_criteria : scores`
- `proba_mutation = 0.05`
- `percentage_for_tournament = 0.10`
- `percentage_new_offspring = 0.30`

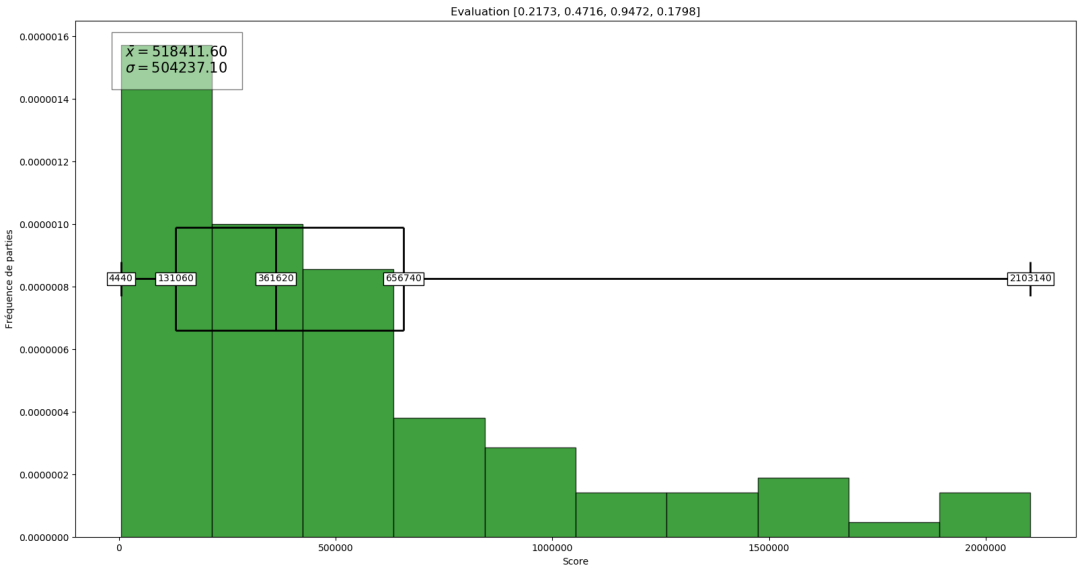
L'optimisation a duré à peu près 3 heures. Voici l'évolution de la population :



On obtient le même profil que précédemment en changeant l'encodage.

Le meilleur individu a pour coefficients : `[0.2173, 0.4716, 0.9472, 0.1798]`.

Regardons ses statistiques sur 100 parties :

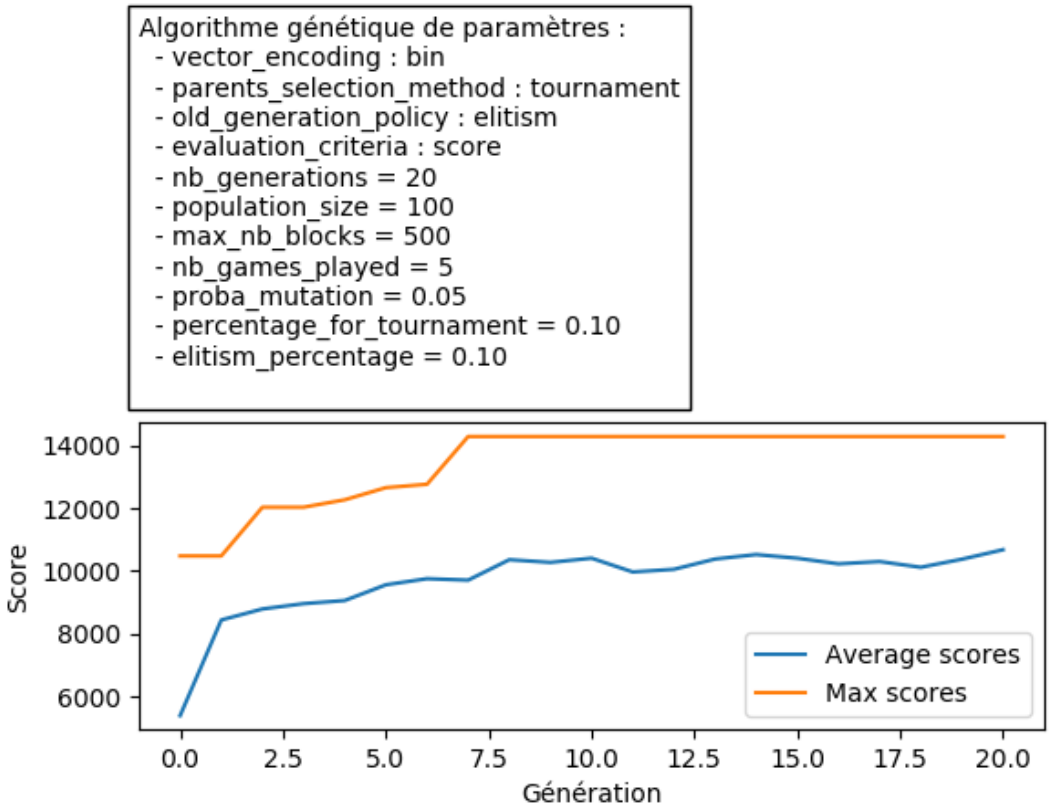


2.4 Exemple 4

Dans cet exemple nous avons changé le mode de conservation des anciens individus :
Paramètres :

- `vector_encoding : bin`
- `parents_selection_method : tournament`
- `old_generation_policy : elitism`
- `evaluation_criteria : scores`
- `proba_mutation = 0.05`
- `percentage_for_tournament = 0.10`
- `elitism_percentage = 0.10`

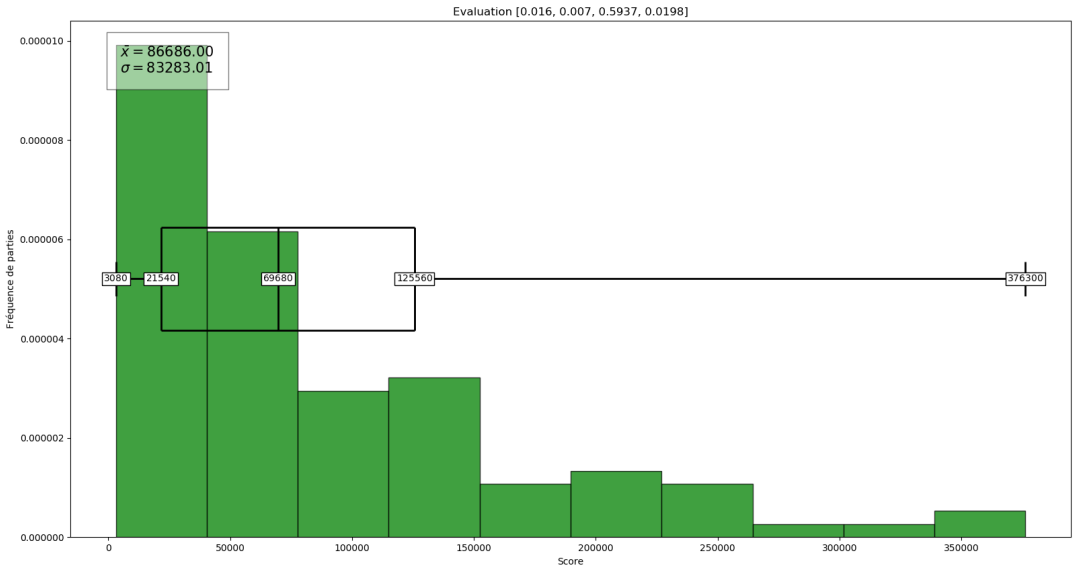
L'optimisation a duré à peu près 12 heures ce qui est considérablement plus long. Voici l'évolution de la population :



On n'a plus ce phénomène de population moyenne qui se stabilise sur le maximum. En revanche, on constate une nette évolution du meilleur individu.

Le meilleur individu a pour coefficients : `[0.0.016, 0.007, 0.5937, 0.0198]`.

Regardons ses statistiques sur 100 parties :

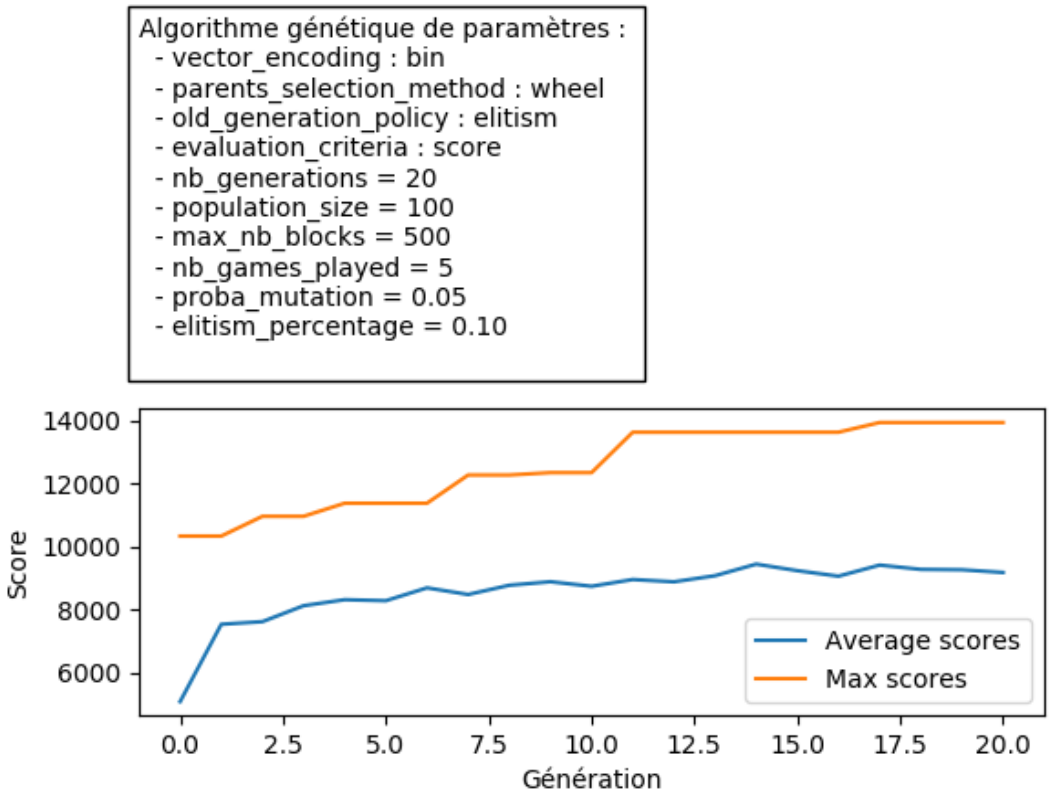


2.5 Exemple 5

Ici, nous avons changé le mode de sélection des individus :
Paramètres :

- `vector_encoding : bin`
- `parents_selection_method : wheel`
- `old_generation_policy : elitism`
- `evaluation_criteria : scores`
- `proba_mutation = 0.05`
- `percentage_for_tournament = 0.10`
- `elitism_percentage = 0.10`

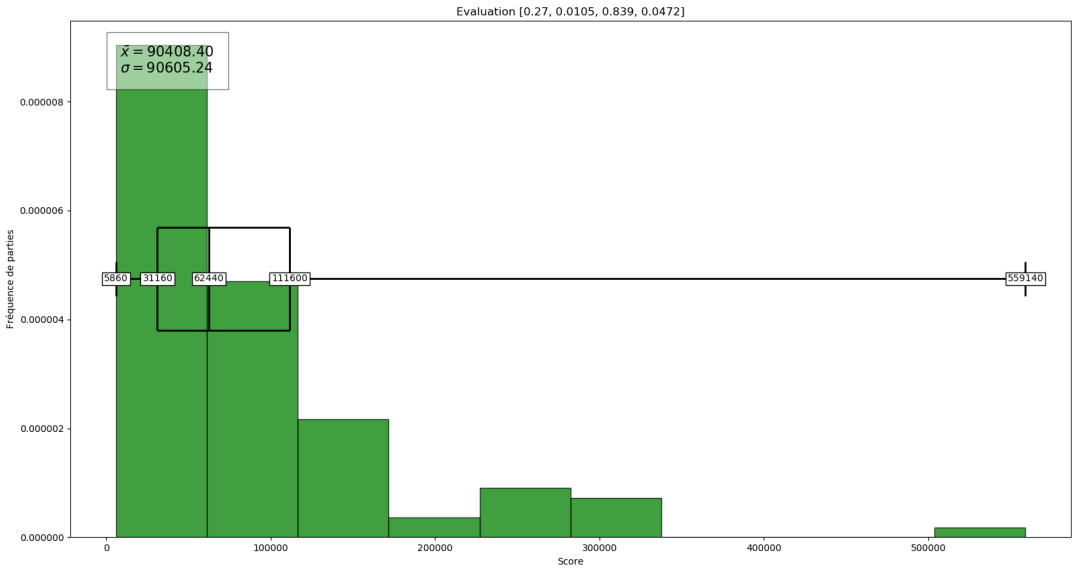
L'optimisation a encore duré à peu près 12 heures. Voici l'évolution de la population :



On observe le même profil que précédemment.

Le meilleur individu a pour coefficients : `[0.27, 0.0105, 0.839, 0.0472]`.

Regardons ses statistiques sur 100 parties :

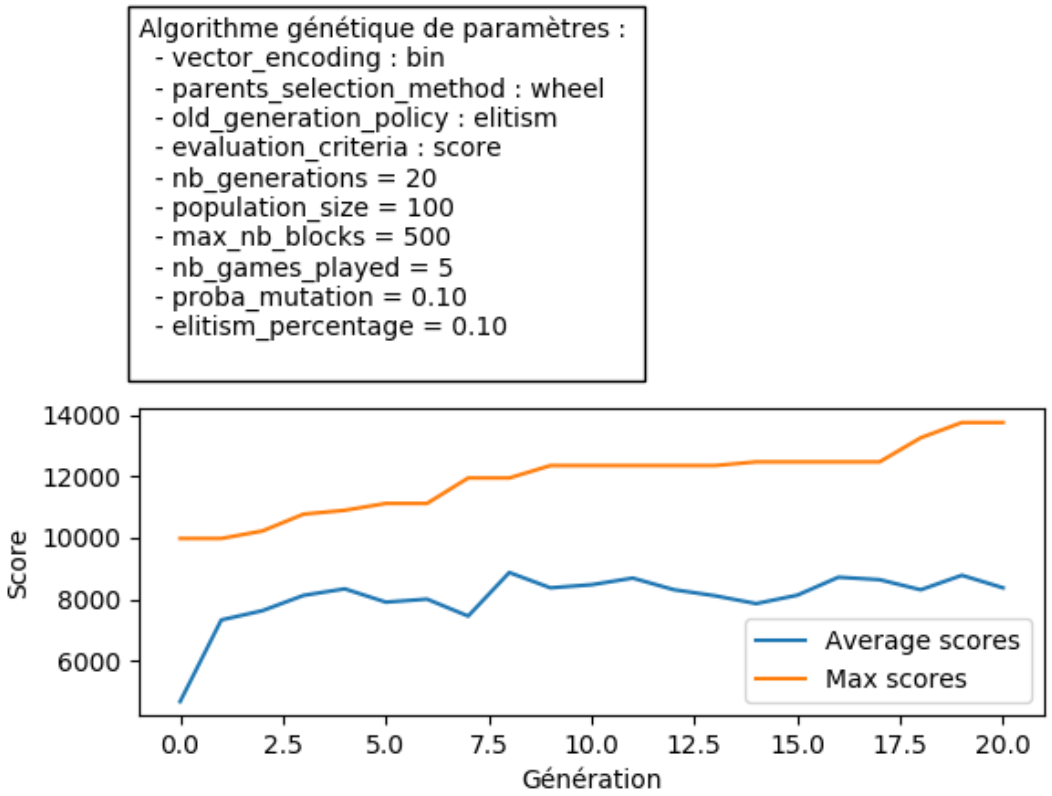


2.6 Exemple 5

Enfin nous avons ajouté de la mutation :
Paramètres :

- `vector_encoding : bin`
- `parents_selection_method : wheel`
- `old_generation_policy : elitism`
- `evaluation_criteria : scores`
- `proba_mutation = 0.1`
- `percentage_for_tournament = 0.10`
- `elitism_percentage = 0.10`

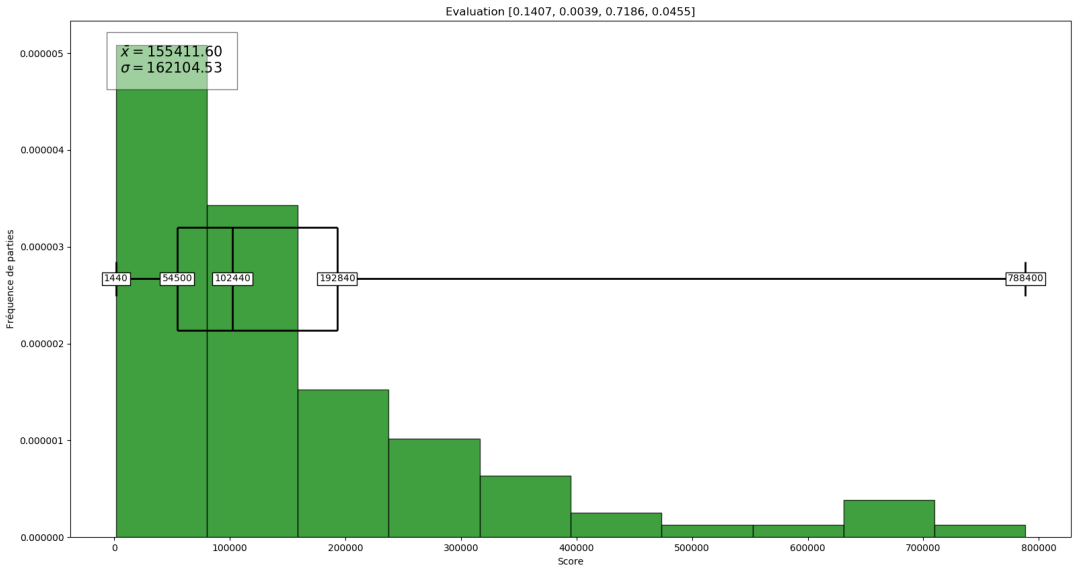
L'optimisation a encore duré à peu près 12 heures. Voici l'évolution de la population :



Le meilleur individu augmente sensiblement. En revanche, la population moyenne semble dégénérer.

Le meilleur individu a pour coefficients : `[0.1407, 0.0039, 0.7186, 0.0455]`.

Regardons ses statistiques sur 100 parties :

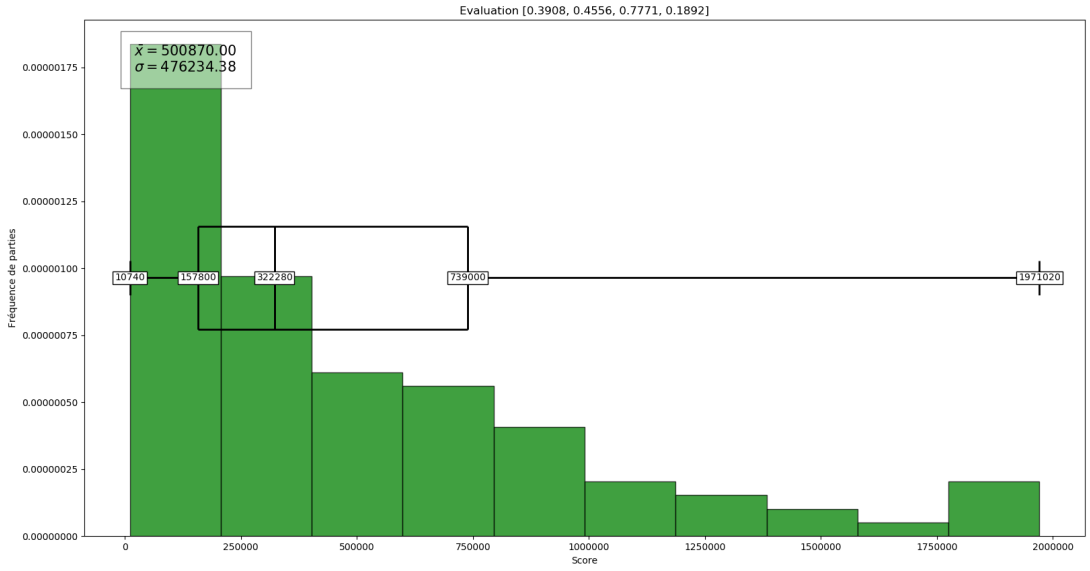


2.7 Dernier exemple

Cette dernière optimisation a tourné pendant 3 semaines et nous avons dû l'interrompre après seulement 9 générations.
Paramètres :

- `population_size : 1000`
- `nb_games_played : 100`
- `vector_encoding : float`
- `parents_selection_method : tournament`
- `old_generation_policy : best`
- `evaluation_criteria : scores`
- `proba_mutation = 0.05`
- `mutation_rate = 0.20`
- `percentage_for_tournament = 0.10`
- `percentage_new_offspring = 0.30`

Voici le résultats sur 100 parties :



3 Optimisation par Q-Learning

Dans la mesure où on ne joue qu'avec des dominos, sur de petites grilles, il n'est pas possible de comparer avec les optimisations précédentes.

Néanmoins, cette optimisation est capable de trouver une stratégie pour jouer sans fin sur une grille de 8×8 en prenant 50000 épisodes d'apprentissage.

Chapitre 11

CONCLUSION

Après plusieurs centaines d'heures de codage et de réflexion, 3000 lignes de code et quelques nuits blanches, nous avons obtenu des résultats tout à fait satisfaisants. Les agents sont plaisants à voir jouer et dépassent les capacités d'un joueur humain.

Le fait d'implémenter plusieurs types d'agents ainsi que la nécessité d'avoir un moteur optimisé au maximum a été très stimulant.

Il ne nous manque que l'implémentation du deep Q-Learning qui était peut-être un peu ambitieux vu le temps dont nous disposions. Nous avons quand même pu faire du simple Q-Learning et cela nous a permis de nous documenter et de comprendre le reinforcement learning.

Il serait possible de faire évoluer le projet sur les points suivants :

- Bien sûr, en implémentant le DQN
- En essayant de paralléliser les calculs dans les algorithmes génétiques
- En rajoutant une interface graphique, ce que nous avons délibérément choisi de ne pas faire ici

Étant tous les deux enseignants, le côté pédagogique est important et ce projet, même s'il dépasse largement le niveau attendu d'un élève de lycée, pourrait servir de support pour travailler les points suivants :

- Réflexion sur le codage d'une structure de données (pièces et grille)
- Organisation d'un projet sous forme de modules et programmation objet
- Algorithmique sur les tableaux (détermination des statistiques de la grille)
- Passage de fonctions en paramètres (callback)
- Gestion de l'affichage d'une grille
- Mise en place de stratégies simples pour les agents (par exemple le filtrage)

To be continued...

BIBLIOGRAPHIE

[1] Y. BDOLAH ET D. LIVNAT – Reinforcement learning playing tetris, 2000. Disponible à l'adresse http://www.math.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html

[2] J. BRZUSTOWSKI – *Can You Win at TETRIS?* T. N.p., 1992. Web. 21 Jan. 2019. Retrospective Theses and Dissertations, 1919-2007. Disponible à l'adresse https://open.library.ubc.ca/collections/ubctheses/831/items/1.0079748

[3] J. GREAVES – Understanding RL : The Bellman Equations, Post de blog daté du 9 novembre 2017. Disponible à l'adresse https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/

[4] S. KISTEMAKER – Cross-Entropy Method for Reinforcement Learning, Thesis for Bachelor Artificial Intelligence, 2008. Disponible à l'adresse https://staff.fnwi.uva.nl/b.bredeweg/pdf/BSc/20072008/Kistemaker.pdf

[5] B. LOGUIDICE ET M. BARTON – *VINTAGE GAMES. An Insider Look at the History of Grand Theft Auto, Super Mario, and the Most Influential Games of All Time*, Elsevier, Focal Press, Amsterdam, 2009, p. 291–301.

[6] D. SHEFF – *GAME OVER. Press Start to Continue*, GamePress, CyberActive Publishing In., Wilton, 1999, p. 292–348.

[7] B. SCHERRER, M. GHAVAMZADEH, V. GABILLON, B. LESNER ET M. GEIST – Approximate Modified Policy Iteration and its Application to the Game of Tetris, *Journal of Machine Learning Research* **16** (2015), pp. 1629-1676

[8] M. STEVENS ET S. PRADHAN – Playing Tetris with Deep Reinforcement Learning. Report for Stanford course CS231n : Convolutional Neural Networks for Visual Recognition, 2016. Disponible à l'adresse http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

[9] R. S. SUTTON ET A.G. BARTO – *Reinforcement Learning : An introduction*, 2nd ed., MIT Press, Cambridge, MA, 2018.