Programmation génétique pour le jeu Tetris

Olivier Gies – MIR Méthodes Evolutionnistes – Mars 2004

étude de l'article

Playing Tetris Using Genetic Programming

Michael Yurovitsky

1 Introduction

La programmation génétique peut être vue comme une technique d'exploration de l'espace des programmes informatiques permettant de résoudre un problème donné. Cette technique est le centre des travaux de recherches du Pr. Koza [3], qui enseigne à l'université de Stanford. Elle s'inscrit dans le domaine des méthodes évolutionnistes dans la mesure où elle permet de trouver des solutions à un problème donné en utilisant les méthodes décrites par les théories évolutionnistes Darwiniennes.

Ce document est une étude critique des travaux réalisés par Michael Yurovitsky, décrits dans [1]. Il se propose d'y implémenter une méthode de programmation génétique appliquée au jeu Tetris, considéré comme une instance du problème plus général du « Bin Packing Problem » décrit plus loin. Après avoir rapidement rappelé les principes de la programmation génétique et détaillé les règles du jeu Tetris, on passe en revue la modélisation adoptée et les résultats obtenus par M. Yurovitsky. On s'attache ensuite à critiquer l'approche présentée et à proposer des pistes alternatives pour aborder le problème évoqué, avant de donner quelques résultats expérimentaux et conclure sur ce problème.

2 Programmation génétique

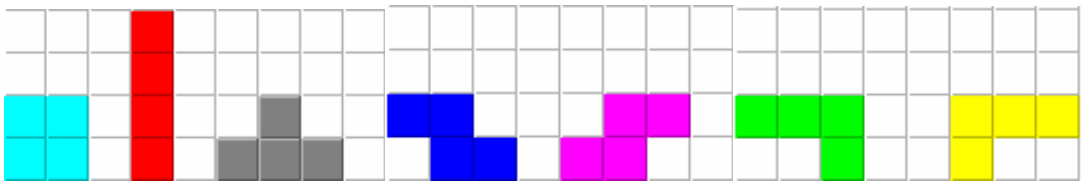
Les programmes conçus par programmation génétique sont représentés par des arbres composés d'éléments de base qui sont les terminaux (variables ou fonctions sans arguments) et les nœuds (fonctions avec arguments). L'exploration de cet espace est inspirée des modèles évolutionnistes darwiniens. Elle part d'une population initiale de programmes, généralement aléatoire, auxquels sont soumis le problème à résoudre. En fonction de leur *capacité* à résoudre ce problème de manière optimale, ils sont *sélectionnés*, puis soumis à un processus de *génération* d'une population fille, à partir des *méthodes évolutionnistes* naturelles.

Les arbres-programmes de la population initiale sont générés aléatoirement à partir de l'ensemble des composantes de bases que sont les nœuds et les terminaux. Cette population initiale est évaluée par une fonction dite de *fitness* qui évalue, la capacité du programme à résoudre le problème. Les individus ainsi évalués sont soumis à un processus de reproduction directement inspiré de la génétique. Trois opérateurs sont utilisés pour générer les individus de la génération suivante en fonction des *fitness* de la population parente : la *reproduction*, le *crossing-over*, et la *mutation*. La reproduction consiste à reprendre tel quel un programme de la population précédente. L'intérêt de cet opérateur est double : d'une part, il permet de conserver les bons programmes à travers les générations, d'autre part, il permet d'entretenir la diversité du *pool* génétique nécessaire à une évolution significative. Le crossing-over consiste à créer de nouveaux individus en échangeant des sous-arbres des individus parents. Enfin, la mutation consiste à générer un nouvel individu à partir d'un individu existant, en y modifiant aléatoirement un sous-arbre. Ce dernier opérateur n'est que très peu utilisé en programmation génétique. Il n'est par ailleurs pas dépendant de la *fitness* des individus, contrairement aux deux opérateurs précédents. La nouvelle population ainsi générée remplace la précédente, et soumise à un nouveau cycle d'évolution.

3 Tetris

Tetris est un jeu inventé par Alexei Pajitnov en 1982. Le principe du jeu est de placer des pièces de différentes formes dans un espace de jeu rectangulaire, haut de 20 lignes et large de 10 colonnes. Il existe de nombreuses variantes de Tetris, mais nous nous intéressons ici à la version standard.

Les pièces utilisées sont l'ensemble des combinaisons possibles de 4 blocs connexes, comme représenté ci-dessous (4 => tétra => tetris). Dans l'ordre, on identifie les pièces par « carré », « barre », « T », « Z », « S », « J » et « L », en raison de leur forme.



Le but du jeu est de placer autant de pièces que possible dans la surface de jeu. Lorsqu'une ligne de la zone de jeu est pleine, elle est retirée du jeu est tous les blocs au-dessus d'elle sont décalés vers le bas. Il est donc possible de jouer à Tetris indéfiniment en faisant régulièrement des lignes complètes. Le joueur est récompensé par un score qui dépend du nombre de blocs placés, accompagné d'un bonus pour chaque ligne complète. La partie s'arrête lorsqu'il n'est plus possible de placer de pièces dans la zone de jeu, ce qui arrive lorsqu'il n'est plus possible de faire de ligne et que les blocs s'accumulent pour atteindre le haut de la zone de jeu.

Un aspect important du jeu de Tetris est le fait que les pièces tombent à une vitesse régulière dans la zone de jeu, ce qui impose de choisir l'endroit où les faire tomber en un temps limité. De plus, cette vitesse augmente avec le temps passé à jouer, donnant ainsi de moins en moins de temps au joueur pour réfléchir. Par ailleurs, les pièces sont générées aléatoirement et il est impossible de prévoir dans quel ordre elles arrivent, la tâche se complique sensiblement la tâche (certaines versions de Tetris proposent cependant l'option de connaître au moins la pièce suivante). Enfin, il est possible faire tourner les pièces, ce qui complexifie les heuristiques possibles (seul le « carré » reste inchangé par rotation).

4 Synthèse de l'article

La programmation génétique est une technique permettant souvent d'obtenir des solutions optimales pour des problèmes où la complexité est un problème, notamment en termes d'explosion combinatoire. Dans son article [1], Yurovitsky utilise cette technique pour développer des programmes joueurs de Tetris. Ce jeu est présenté de manière détaillée plus loin dans ce document. Il est présenté comme une variante du problème de programmation linéaire classique mais très difficile du « Remplissage de Boîte » (« Bin Packing Problem »), défini comme suit par Kröger [2] : « Etant donné un ensemble *R* de *m* 'petits' rectangles différents et non-orientés ainsi qu'une 'grande' boîte *B*, l'objectif est de placer les éléments de *R* dans *B*. Le remplissage est orthogonal, i.e. chaque rectangle est placé tel que ses côtés soient parallèles aux bords de la boîte. Celle-ci est théoriquement de hauteur infinie et de largeur donnée *w*. Le but est de minimiser l'expansion verticale maximum du remplissage de *B*. » Le jeu de Tetris est une instance de ce problème, comme décrit dans la section qui suit. L'auteur en déduit qu'appliquer les méthodes de programmation génétique à Tetris est une bonne alternative aux méthodes linéaires généralement utilisées pour résoudre le problème décrit précédemment.

4.1 Simplifications

Plusieurs modifications sont apportées à la version standard de Tetris avant de lui appliquer les méthodes de programmation génétique. Elles sont décrites dans ce paragraphe.

4.1.1 Pas de chute

Le fait que la pièce tombe est totalement occulté. La position et l'orientation de la pièce sont ainsi déterminées avant que celle-ci ne soit placée. Cette simplification abolit les contraintes physiques du Tetris réel, notamment le fait qu'une pièce ne puisse être tournée si elle est coincée dans un « couloir » vertical. L'auteur argumente que de bonnes heuristiques ne devrait de toute façon pas mener à ce genre de situation dans le mesure où un couloir ne devrait apparaître que lorsque les pièces ont été « mal » empilées.

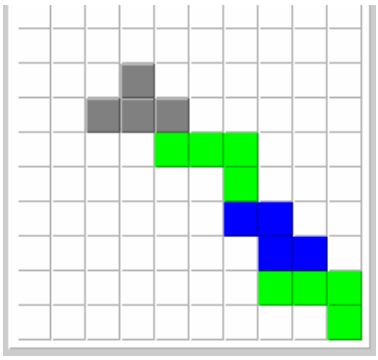
4.1.2 Nombre de pièces et taille de la zone de jeu

La zone de jeu utilisée dans ce projet est limitée à une taille de 10 lignes de haut et 8 colonnes de large. Cette simplification est uniquement faite pour des raisons de temps de calcul. L'auteur argumente que ceci ne devrait pas réellement influencer les bonnes stratégies, dans le mesure où on est en droit d'attendre d'une bonne stratégie qu'elle puisse compléter des lignes indépendamment de la largeur de la zone de jeu. Une taille plus petite permettrait ainsi de faire émerger plus rapidement de bonnes stratégies, car les mauvaises stratégies convergeraient plus rapidement vers des situations d'échec. Par ailleurs, la barre utilisée pour cette simulation est de longueur 3 et non pas 4. Ce choix n'est pas argumenté.

4.1.3 Vides inaccessibles

Dernière simplification importante, les espaces vides « cachés sous » des pièces déjà posées ne sont pas accessibles, bien qu'en réalité, il serait possible d'y mettre des pièces en passant par le côté (cf. figure). Cette simplification est motivée par une volonté de faciliter l'effort de programmation. Bien que cela semble une simplification significative par rapport à l'esprit du jeu de base, l'auteur argumente que cela ne devrait pas altérer la convergence vers de bonnes stratégies. La raison principale invoquée est qu'une stratégie menant à ce genre de situation n'a littéralement aucune chance d'être rattrapée plus tard. En d'autres termes, une

stratégie développant d'abord ce genre de remplissage est déjà si mauvaise que n'importe quelle heuristique sans « vides » est déjà meilleure.



4.2 Phase Préparatoire et définition du problème

Cette partie concerne la définition du problème en termes des méthodes de programmation génétique. En programmation génétique, il est futile d'exiger l'émergence de solutions optimales si les éléments de base pour les construire ne sont pas adaptés au problème. Il s'agit donc de définir un bon ensemble de terminaux et de fonctions pour construire les individus à évaluer. Par ailleurs, il faut définir la fonction de d'évaluation (*fitness*) qui attribuera un score à chaque individu en fonction de sa capacité à résoudre le problème. Enfin il faut établir les paramètres d'évolution génétique tels que le nombre de générations à faire évoluer, le nombre d'individus, et les paramètres de crossing-over et de mutation.

4.2.1 Fonctions et terminaux

4.2.1.1 Terminaux

L'auteur a utilisé différents ensembles de fonctions et terminaux au cours de son étude du problème. Il a d'abord envisagé de décrire les solutions du problème au moyen d'éléments permettant de récupérer des informations sur l'état de la zone de jeu, la pièce courante, puis de décider de la meilleure orientation à lui donner et du meilleur endroit où la placer. Pour récupérer les informations sur le contenu de la zone de jeu, il a défini les terminaux COL1 à COL8 et le terminal PIECE. COL# retourne la hauteur des blocs dans la colonne numérotée #, tandis que PIECE identifie la pièce en cours de traitement. Après avoir trouvé les terminaux COL# ineffectifs après plusieurs simulations, il a opté préférentiellement pour une fonction PROFILE à un argument, qui retourne la hauteur de blocs dans la colonne donnée en argument.

L'opérateur DEEPEST est défini comme un terminal identifiant la colonne la plus profonde. Ce choix est également motivé par des considérations de temps de calcul. L'auteur explique qu'il a d'abord tenté d'utiliser des ADFs (« *automatically defined function* ») à la place de ce terminal, mais que les résultats étaient particulièrement mauvais, et l'émergence de bonnes stratégies toujours plus longue.

4.2.1.2 Fonctions

Toutes les données du problème étant représentées par des entiers, deux fonctions ADD et SUB sont définies pour permettre au système de manipuler les opérations élémentaires sur les entiers. Elles opèrent bien entendu l'addition et la soustraction de 2 entiers.

L'auteur a ensuite défini deux fonctions de comparaison, fondées sur le principe que l'approche de beaucoup de joueurs humains consiste essentiellement à comparer différentes

valeurs de hauteurs de colonnes pour évaluer le meilleur site où placer la pièce. Ainsi IFEQ est une fonction à 4 arguments, qui compare les deux premiers, retourne le 3ème s'ils sont égaux et le 4ème sinon. De manière analogue, IFLT effectue la comparaison par valeur inférieure (IF Lower Than), toujours avec 4 arguments.

Pour chaque pièce, il est nécessaire de déterminer à la fois sa position et son orientation. Comme une fonction ne peut retourner qu'une seule valeur, le résultat du RPB (« result producing branch ») d'un individu retourne la colonne dans laquelle la pièce doit être placée. L'auteur définit donc une fonction SETROT qui tient à jour une variable globale du problème définissant l'orientation de la pièce. En conséquence, seule le dernier appel à SETROT aura un effet au cours de l'exécution d'un programme.

4.2.2 Cas d'étude

L'évolution a été effectuée sur une séquence test de 100 pièces à empiler dans la zone de jeu. La composition de cette séquence dépend des expériences testées. Par ailleurs, l'évaluation (fitness) d'un individu est calculée soit une fois que les 100 pièces ont été placées avec succès dans la zone de jeu, soit lorsque la pièce suivante ne peut pas être placée par la stratégie évaluée sans dépasser de la zone de jeu.

4.2.3 Fitness

L'évaluation d'un individu est calculée en soustrayant le score obtenu par l'individu au maximum théorique de points possibles avec la séquence de pièces proposée. Cette normalisation est choisie pour faire en sorte que les meilleurs individus ont les *fitness* les plus faibles. Le maximum de points théorique est obtenu par la formule suivante :

$$Max = \sum_i N_i * A_i + 100 * \left\lfloor \left(\sum_i N_i * A_i \right) \%8 \right\rfloor \qquad (1)$$

Dans cette formule, *i* représente le type d'une pièce, N_i est le nombre d'occurrence de ce type de pièce dans la séquence et A_i le nombre de blocs qui la composent. Le maximum théorique est donc le nombre de blocs individuels (« carrés ») disponibles dans la séquence de test auquel s'ajoute le nombre théorique de lignes possibles de 8 blocs avec le nombre de blocs disponibles dans la séquence. De manière analogue, la fitness *brute* d'un individu est donnée par :

$$f = A + 100 * R \qquad (2)$$

Où *A* est le nombre de blocs effectivement placés et *R* le nombre de lignes pleines constituées au cours de l'exécution. L'auteur souligne que même une stratégie optimale peut ne pas avoir une fitness nulle. En effet, il n'est pas toujours possible d'assembler autant de lignes que le maximum théorique de lignes proposé dans la formule (1). A noter que l'idée et de favoriser les stratégies qui composent des lignes, car c'est d'une part une façon optimale d'empiler les blocs et d'autre part une source de bonus dans le vrai Tetris.

4.2.3.1 Paramètres de l'évolution

La bibliothèque utilisée pour implémenter le problème est le système de programmation génétique DGPC, développé par David Andre à Stanford. Elle définit 4 paramètres pour les opérations génétiques :

- taux de copie = 10% : proportion d'individus directement copiés d'une génération à l'autre
- taux de feuilles pour le crossing-over = 10% : proportion sur toutes les feuilles (terminaux) utilisée pour le crossing-over, i.e. 10% des individus de la nouvelle génération seront obtenus par crossing-over sur les feuilles
- taux de nœuds pour le crossing-over = 79% : proportion sur tous les nœuds (fonctions) utilisée pour le crossing-over
- taux de mutation = 1% : pourcentage d'individus obtenus par mutation

Par ailleurs, l'auteur a utilisé un nombre de générations fixé de 100 pour toutes les évolutions. Pour raisons de temps de calcul, les tailles de population varient de 500 à 2000 individus selon les tests effectués. Enfin, aucun individu n'a été conservé d'un test à l'autre.

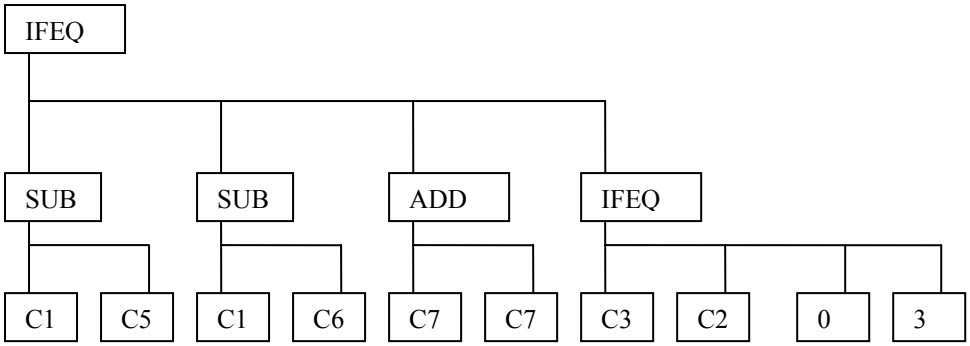
4.3 Résultats

4.3.1 Tetris à 2 pièces

Dans les premières simulations, L'approche développée est une version très simplifiée du problème : faire évoluer des méthodes de jeu utilisant uniquement 2 pièces distinctes, à savoir le *carré* et la *barre* (cf. §3).

4.3.1.1 Séquence périodique

La séquence de pièces utilisée ici est périodique du type { 1 carré, 4 barres, 1 carré, 4 barres, …}. Tous les tests ont généré des solutions très rapidement dans ce contexte, dont certains même dans la population aléatoire initiale. Voici l'une des solutions obtenues dans ce cadre :



Cet arbre correspond à la S-expression suivante :
IFEQ (SUB C1 C5) (SUB C1 C6) (ADD C7 C7) (IFEQ C3 C2 0 3)

Les terminaux C# sont des abréviations des terminaux COL# utilisés pour les premières expériences. Cette solution est une solution parfaite. Voyons ci-dessous comment elle s'exécute :

- 1ère pièce : *carré*

  La zone de jeu étant vide, C# = 10 (profondeur maximale) pour toutes les colonnes. Donc (SUB C1 C5) = (SUB C1 C6), et c'est (ADD C7 C7) = 20 qui est évalué pour la position. Elle est recadrée à l'intérieur de la zone de jeu et le *carré* est placé tout à fait à droite.

- 2ème pièce : *barre*

  C6 vaut maintenant 8, donc (SUB C1 C5) != (SUB C1 C6). On passe dans le 2ème IFEQ, qui compare C3 et C2. Ces derniers étant égaux, la position choisie et le 0 : la barre va tout à fait à gauche
- 3ème pièce : *barre*

  On a toujours (SUB C1 C5) != (SUB C1 C6) car C5 != C6. De même, on passe dans le 2ème IFEQ, qui compare C3 et C2. Maintenant, C2 != C3 à cause de la barre précédente, donc la position choisie est le 3 : la barre s'insère dans les 3 espaces vides et complète ainsi une ligne.
- 4ème et 5ème pièces: *barre, barre*

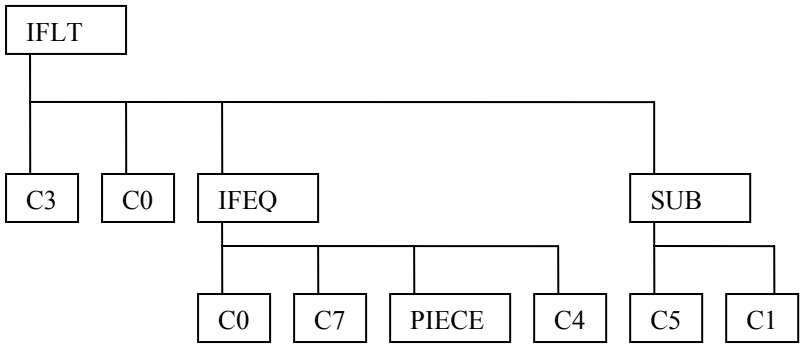
  On se retrouve dans la même situation que pour la deuxième pièce, avec cette légère différence que C6 et C7 valent maintenant 9 et non plus 8, mais cela ne change pas le parcours de l'arbre. Les deux barres sont donc placées comme les précédentes et complètent une nouvelle ligne

Après un cycle {carré, barre, barre, barre, barre}, la zone de jeu est de nouveau vide. On est donc de retour au point de départ. Cette solution est parfaitement adaptée aux conditions du problème, à la fois à la périodicité des cycles et à la nature des pièces de la séquence.

A titre indicatif, une deuxième solution est présentée ci-dessous. Nous ne détaillerons pas le déroulement de cette solution, mais l'intérêt est de montrer que le terminal PIECE est utilisé de manière plutôt inattendue, en raisons de la modélisation informatique du système. L'auteur précise qu'en réalité, le terminal PIECE est quasiment toujours utilisé de cette manière, ce qui rend l'interprétation directe des S-expression très difficile.

S-expression de l'arbre ci-dessous :

IFLT C3 C0 (IFEQ C0 C7 PIECE C4) (SUB C5 C1)



4.3.1.2 Séquence aléatoire en proportion 4

La seconde série de tests effectuée dans le cadre du Tetris à 2 pièces diffère de la précédente dans le faite que la séquence de pièces n'est plus périodique mais aléatoire, tout en conservant un ratio $N_{barre} / N_{carré}$ sensiblement égal à 4. Comme précédemment, toutes les expériences ont abouti à des stratégies réussissant à placer les 100 pièces dans la zone de jeu, bien que cela prenne un plus grand nombre de générations.

L'auteur observe alors dans ces conditions une évolution similaire dans toutes les expériences. Le programme génétique commence par créer une population d'individus

aléatoires dont certains arrivent à compléter quelques lignes. Au cours des générations, la fitness du meilleur individu évolue par à-coups de 100 points lorsqu'il arrive à compléter une ligne supplémentaire par rapport à la génération précédente, ou beaucoup plus si la combinaison de 2 stratégies précédentes s'avère très performante. Par exemple, dans l'une des évolutions, la fitness du meilleur individu pour la génération 0 était de 4061 et ne parvenait qu'à faire 2 lignes complètes. A la génération 3, le meilleur individu avait une fitness de 2978 et parvenait à compléter 12 lignes avant d'échouer. La solution était atteinte à la 11ème génération, avec une fitness de 100, et 39 lignes complétées pour un maximum théorique de 40 lignes.

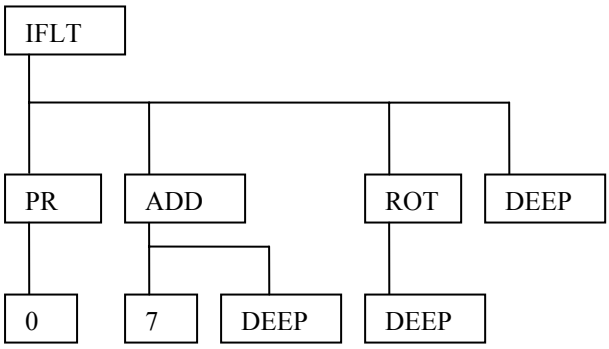
4.3.1.3 Séquence aléatoire en proportions équitables

Dans un premier temps, l'auteur reprend les conditions des tests précédents en changeant la proportion des pièces de 4 à 1 (répartition uniforme). Dans ces conditions, les évolutions successives ne donnent pas des très bonnes stratégies, la meilleure stratégie moyenne ne complétant que 10 lignes avant d'échouer.

Pour permettre de développer des stratégies efficaces, l'auteur introduit alors la fonction à 1 argument ROTATION qui retourne simplement son argument mais effectue également une rotation de la pièce de 90 degrés. Par ailleurs, les terminaux COL# sont remplacés par la fonction à 1 argument PROFILE qui retourne la profondeur de la colonne en argument, et le terminal DEEPEST est ajouté.

Dans ces conditions, le programme génétique donne de très bons résultats. Plus de 70% des évolutions aboutissent à une solution parfaite, i.e. qui réussit à placer les 100 pièces. Ci-dessous, l'arbre de la plus courte stratégie empilant les 100 pièces et complétant 39 lignes sur 43 théoriques.

S-expression : IFLT (PROFILE 0) (ADD 7 DEEPEST) (ROTATE DEEPEST) DEEPEST



L'auteur ne décrit pas le déroulement de ce programme. C'est regrettable car il est impossible de le faire sans connaître l'implémentation de DEEPEST. En effet, si toutes les colonnes sont au même niveau, on ne sait pas quelle colonne est choisie par DEEPEST, ce qui conditionne fortement la condition du IFLT à cause du second argument (ADD 7 DEEPEST).

4.3.2 Tetris à 3 et 4 pièces

Dans ces expériences, l'auteur rajoute tout d'abord la pièce T aux deux précédentes. Différentes séries de tests sont exécutées sur des populations de 500, 1000 et 2000 individus, avec et sans ADFs, mais aucune ne permet d'obtenir des solutions intéressantes.

Ces résultats ont amenés l'auteur à modifier la structure du programme pour construire des individus multi-RPB. Chaque individu contient un arbre définissant une stratégie pour une pièce donnée. Le terminal PIECE est donc supprimé. Ce changement dans la structure des programmes a permis d'obtenir de meilleurs résultats sans atteindre ceux des expériences à 2 pièces : seuls deux individus arrivent à placer les 100 pièces.

La quatrième pièce ajoutée est le Z. Comme précédemment, chaque individu contient 4 arbres, définissant une stratégie différente pour chaque pièce. Les résultats obtenus sont moins bons que ceux du Tetris à 3 pièces. L'auteur avance que cela peut être du au trop petit nombre de générations. Il s'appuie sur le fait que la seule solution réussissant à placer les 100 pièces est obtenue à la 93ème génération.

4.3.3 Synthèse des résultats

Le tableau ci-dessous présente un récapitulatif chiffré des résultats obtenus par l'auteur.

On y observe que les 2 modifications importantes qui ont contribué à améliorer les performances sont d'une part l'ajout de la fonction ROTATION, et d'autre part l'utilisation de différents RPBs pour chaque pièce (rendant superflue l'utilisation du terminal PIECE).

Conditions de test	Meilleur individu			Nombre de tests	Meilleure fitness moyenne	Nombre moyen de lignes	% des tests plaçant les 100 pièces	Nombre moyen de pièces placées	Temps d'exécution moyen (min.)
	Fitness	Lignes	Pièces						
2 pièces, séquence **périodique** {1 carré, 4 barres }	0	40	100	8	0	40	100	100	2
2 pièces, séquence **aléatoire** en proportion 1 : 4	100	39	100	12	280	38	100	100	15
2 pièces, séquence **aléatoire** en proportion **1 : 1**	698	33	91	56	3096	10	0	35	20
2 pièces, séquence **aléatoire** en proportion **1 : 1**, avec **ROTATION**	0	42	100	65	645	37	71	96	28
3 pièces, séquence aléatoire en proportion 1 : 1 : 1	2638	20	61	8	3289	15	0	49	151
3 pièces, séquence aléatoire en proportion 1 : 1 : 1 **avec 3 RPBs**	300	42	100	30	2736	20	7	57	64
4 pièces, séquence aléatoire en proportion 1 : 1 : 1 : 1 avec **4 RPBs**	400	42	100	24	3240	15	4	48	74

5 Approche personnelle

Dans ce paragraphe, on se propose de critiquer la modélisation du problème telle que décrite dans l'article de M. Yurovitsky. On se propose dans un premier temps de faire une critique objective du travail présenté, avant d'aborder des pistes éventuelles pour faire évoluer ses travaux, puis de présenter quelques résultats expérimentaux obtenus en testant ces nouvelles approches.

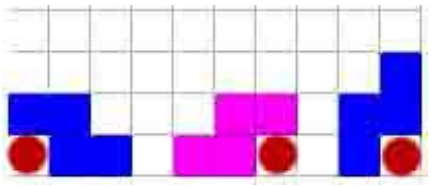
5.1 Critique du modèle

5.1.1 Simplifications du problème

La première simplification introduite par l'auteur semble justifiée. En effet, ne pas prendre en compte la chute des pièces ne pose pas de problème en soi pour trouver la meilleure position où placer la pièce. Du point de vue du jeu, le principe de la chute des pièces se justifie principalement par la difficulté proposée au joueur. L'objectif étant de faire découvrir des stratégies de jeu optimales par un ordinateur, on est en droit de supposer que le temps de décision n'est pas une contrainte pertinente. D'un autre côté, comme expliqué dans l'article, le chute des pièces peut être une contrainte pour leur positionnement dans la mesure où une pièce bloquée dans un couloir vertical ne pourra pas être tournée pour être placée dans une position potentiellement optimale. L'argument que de bonnes stratégies ne devraient pas mener à ce genre de situation est justifié à l'exception d'une situation particulière qui est décrite plus loin, dans la critique de la 3ème simplification.

Considérant que l'objectif des heuristiques développées dans ce travail est de maintenir un niveau relativement bas dans la zone de jeu, les simplifications de dimensions semblent également justifiées. En effet, une hauteur de 10 cases permet – avec une barre de 3 blocs – au minimum de placer 3 pièces l'une sur l'autre sans dépasser. Ne retenir que 8 colonnes de largeur est une simplification déjà plus discutable, dans la mesure où les formes particulières des pièces influe sur la facilité à constituer des lignes en fonction de leur largeur. La combinaison traditionnelle {*carré, barre horizontale, barre horizontale*} pour créer une ligne dans le jeu normal (2+4+4=10 blocs) n'est pas affectée car la barre est réduite à 3 blocs (2+3+3=8 blocs). Cependant, réduire la largeur à 8 blocs introduit une facilité de jeu lorsqu'on utilise les pièces *T*, *L* et *J*. En effet, chacune de ces pièces présente une orientation qui peut la faire passer pour une barre horizontale de 3 blocs de largeur. Ainsi, les conbinaisons « simples » {*carré, L, L*},{*carré, J, J*},{*carré, T, T*} peuvent également constituer des lignes de 8 blocs. Ceci facilite le problème dans la mesure où dans un Tetris à 10 colonnes, c'est précisément le genre de situation recherchée et difficile à atteindre avec des pièces de 3 blocs de largeur. En d'autres termes, seule la combinaison {*carré, barre horizontale, barre horizontale*} permet de compléter une ligne en 3 pièces dans le vrai Tetris, tandis que 4 combinaisons le permettent une fois la largeur réduite à 8 blocs. Les résultats obtenus doivent donc être relativisés au regard de cet argument, principalement les résultats obtenus pour le Tetris à 3 pièces (*carré*, *barre*, *T*).

Enfin, la simplification principale de ne pas permettre de placer la pièce sous les « arches », comme présenté dans le §4.1.3 n'est, elle aussi, que partiellement justifiée. Il est vrai que l'on est en droit d'attendre de bonnes stratégies qu'elles convergent rapidement vers des solutions complétant les lignes en allant « droit au but ». Ceci étant dit, il est fréquent dans une partie classique de Tetris d'être confronté à un cas limite qui est celui d'un vide d'un seul bloc. En effet, si la première pièce de la partie est un *Z* ou un *S*, il est inévitable de créer un vide comme montré ci-dessous (points rouges).



Dans une partie réelle, n'importe laquelle des pièces *barre*, *J*, *L* et *T* permet de se sortir rapidement de ce genre de situation, particulièrement en prenant en compte la chute des pièces, et en attendant que la pièce en question soit à la bonne hauteur et dans la bonne orientation. Ceci n'est pas possible dans cette modélisation. Cependant, ce genre de situation est plus rare qu'il n'y paraît car elle n'apparaît que lorsqu'il n'y a pas de « créneau » pour placer le *S* ou le *Z*. A noter qu'une fois placée, d'autres *S* peuvent être empilé sur le premier *S*, et de même pour les *Z*. A titre indicatif, on peut majorer le préjudice causé par cette modification en imaginant la pire des situations : sur une série de 100 pièces à proportions égales pour tous les types, où cette situation apparaît pour chaque *S* ou *Z* ; celles-ci étant approximativement au nombre de 2*(100/7) = 28,… ceci correspond au nombre maximum de lignes « perdues » à cause de cette simplification. Bien que très rudimentaire et peu réaliste, ce calcul rend indirectement compte d'un avis partagé par nombre de joueurs : les pièces *S* et *Z* sont les plus antipathiques. Ce sont souvent celles qui amènent à des empilements de stockage sur les côtés de la zone de jeu, qui peuvent ensuite entraîner l'échec d'une partie. On peut imaginer que ce sont aussi celles qui vont rendre plus difficile l'émergence d'heuristiques très performantes.

On peut éventuellement considérer que l'apport produit par la seconde simplification est compensé par les pertes que pourrait générer la 3ème simplification, une évaluation quantitative restant à être établie.

5.1.2 Choix des terminaux et des fonctions

Comme expliqué en §4.3.1.1, la modélisation informatique du problème est fondée sur l'emploi d'entiers pour représenter les différentes données (identification d'une pièce, profondeur d'une colonne, numéro de colonne, orientation). Le système de programmation génétique employé ne distingue donc pas ces différents types de données et peut employer l'un pour l'autre sans scrupule. L'auteur note d'ailleurs à fort juste titre que cela rend littéralement impossible l'interprétation par un lecteur humain des S-expressions obtenues, qu'elles soient de mauvaises ou de bonnes stratégies. Bien que l'emploi de SETROT et l'utilisation de RPB permettant d'éviter le terminal PIECE règlent partiellement ce problème, on voit toujours des solutions mélangeant hauteur de blocs et numéro de colonne, ce qui est totalement impossible à interpréter, notamment dans le cas de comparaisons (IFEQ, IFLT). Une alternative proposant un typage des données est proposée plus loin dans ce document.

Un autre point discutable de l'approche proposée est l'utilisation du terminal SETROT. En effet, contrairement aux autres fonctions et terminaux, dont l'action se limite à sonder la zone de jeu et effectuer des comparaisons entre diverses données, SETROT a une action directe sur l'orientation de la pièce. Comme expliqué par l'auteur, seul le dernier appel à SETROT est pris en compte, ce qui implique que l'orientation finale de la pièce dépend notamment de l'ordre d'évaluation dans l'arbre d'un individu.

Enfin, on constate un problème apparemment inévitable dans l'approche de la programmation génétique pour les opérateurs de comparaison IFEQ et IFLT, à savoir l'utilisation de conditions tautologiques. Cela ne pose cependant pas de réel problème dans la mesure où cela ne fait que surcharger l'arbre sans introduire de préjudice pour la recherche de solutions.

5.2 Apports

Ce chapitre présente des approches qui étendent le travail proposé dans l'article étudié. En plus d'étendre la modélisation proposée au Tetris standard, on propose des améliorations potentielles par typage des données et extension des ensembles de fonctions et de terminaux. Enfin quelques résultats implémentant ces améliorations sont également rapportés.

5.2.1 Extension au vrai Tetris

L'approche immédiate retenue après étude de ces travaux a été d'étendre la méthode proposée au Tetris complet à 7 pièces, avec les dimensions standard de 20 lignes et 10 colonnes (cf. résultats en§5.3).

Par ailleurs, un apport non négligeable serait de permettre le placement de pièces dans les zones « couvertes » comme expliqué dans le paragraphe précédent. Cette approche n'a pas été implémentée. Elle entraîne cependant une réflexion sur l'utilité de prendre en compte la chute des pièces et ses conséquences sur la modélisation du problème. En effet, les individus obtenus génétiquement devant pouvoir faire face à des situations locales (pièce donnée dans un état de jeu donné), la prise en compte de la chute implique également une localisation temporelle de l'action. Autrement dit, l'idée serait d'implémenter, par exemple, une fonction d'attente d'un tour avant de tester le voisinage pour trouver une solution. Comme pour SETROT, une telle fonction aurait un impact direct sur l'état du système au moment de l'évaluation, qui serait donc biaisée car dépendante de l'ordre d'évaluation de l'arbre. Ceci dit, de nombreuses approches en programmation génétique se fondent sur une modalité séquentielle, comme par exemple la célèbre antenne de la NASA conçue par programmation génétique en faisant progresser une entité virtuelle dans un espace 3D avant d'utiliser sa trajectoire comme modèle d'antenne. Ainsi, un effort de réflexion et de modélisation pourrait être apporté à ce niveau là.

5.2.2 Typage des données

L'idée de la programmation génétique étant d'explorer un espace de programmes a priori infini, il semblerait judicieux d'imposer des contraintes de types sur les données manipulées. En plus de résoudre, d'une part, le problème de l'interprétation par un esprit humain des S-expression et des arbres obtenus, cela contraindrait l'espace de recherche des heuristiques possibles et pourrait permettre l'émergence de solutions pertinentes. Le fait d'empêcher l'utilisation « barbare » de fonctions *n*-aires sur des types de données différentes ne pose aucun problème ici car les concepts de hauteur de blocs et de numéro de colonne sont totalement indépendants. Ce type de contraintes sur le typage des données, appelé « Strong Typing » (« typage fort ») est notamment décrit dans [5].

Le principe du typage des éléments de base des arbres est d'imposer une « sémantique » sur les objets manipulés. Ainsi, des types peuvent être imposés aux valeurs des terminaux comme aux arguments des fonctions et à leur valeur de sortie. Les arbres sont alors constitués en tenant compte de ces contraintes : un terminal de type A ou une fonction retournant ce type ne peut dans ce cas pas être argument d'une fonction qui exige des arguments de type B. Dans notre problème, l'avantage est par exemple de pouvoir modéliser les hauteurs de blocs des colonnes comme un certain type et les identifiants (numéros) de colonne comme un autre. On verrait ainsi disparaître les comparaisons « barbares » entre ces deux types de données explicitement indépendants, ainsi que le placement d'une pièce dans un numéro de colonne obtenu à partir d'une hauteur de blocs. Une implémentation utilisant ce système de typage est décrite plus loin dans ce document.

5.2.3 Terminaux et fonctions

Une motivation essentielle du choix des terminaux et des fonctions dans le travail étudié est l'analogie avec l'approche d'un joueur humain. Partant de ce même principe d'analogie avec le raisonnement humain, plusieurs approches viennent à l'esprit.

5.2.3.1 Connaissance de la pièce

L'approche actuelle implémente une identification interne des pièces selon une certaine valeurs. Même en utilisant plusieurs RPBs (cf. plus loin), la connaissance de la pièce se limite à son identifiant. Ceci implique que les heuristiques cherchant le bon site où placer une pièce déterminent, en retour, la forme de la pièce *implicitement*. Autrement dit, c'est en trouvant les bons sites qu'il en déduit la forme qu'a une pièce. Tetris est un jeu très accessible au joueur novice, et l'on peut expliquer cela par une raison simple : placer une pièce au bon endroit relève du jeu de cubes d'un enfant. L'approche immédiate semble en effet de comparer tout simplement la forme de la pièce au « paysage » de la zone de jeu. De cette manière, il pourrait être intéressant d'introduire un ensemble de fonctions ou de terminaux permettant d'obtenir des informations sur la composition de la pièce.

5.2.3.2 Détection de « trous »

Si le joueur humain cherche effectivement à placer les pièces de manière optimale pour créer des lignes, il n'a pas toujours « la bonne pièce au bon moment ». En d'autres termes, il n'y a pas toujours de bonne position pour placer la pièce courante. Ceci implique qu'un joueur cherche à ne pas créer des situations de préjudice, en évitant par exemple de créer des « trous ». Cette recherche est implicitement faite dans la recherche du site le plus adapté à la forme de la pièce, mais à l'instar du terminal DEEPEST, qui aide le programme à trouver des heuristiques optimales en cherchant directement la colonne la plus profonde, on pourrait imaginer une fonction VOIDS, prenant en argument un numéro de colonne et retournant le nombre de trous présents dans cette colonne. En effet, une heuristique fréquente chez les joueurs de Tetris est de ne pas accumuler de pièces au-dessus des trous déjà présents, afin d'éliminer les blocs bouchant les trous et de constituer des lignes une fois ces trous libérés.

5.2.3.3 Opérateurs booléens

L'éventuel typage des données peut être combiné à un ensemble de fonctions plus adaptées au problème de la recherche du site optimal pour placer une pièce. Ainsi, si l'intuition de l'auteur d'utiliser les comparateurs IFEQ et IFLT est judicieuse, pourquoi ne pas pousser plus loin et écrire un ensemble d'opérateurs booléen complet ? Un tel ensemble comprendrait les classiques opérateurs binaires AND et OR, ainsi que l'opérateur unaire NOT. De plus, les comparateurs seraient remplacés par les opérateurs classiques de comparaison, comme par exemple EQUAL ou LOWERTHAN, prenant en argument deux données de même type et retournant un booléen. L'opérateur IF serait alors différencié et prendrait 3 arguments : un booléen conditionnel et deux branches de résultats selon la valeur de condition.

5.2.3.4 Intervalles par type de donnée

La modélisation d'une unique variable entière aléatoire pour introduire des valeurs de hauteurs, de numéro de colonnes ou de calcul sur ces grandeurs semble très limitée. En effet, ces différentes grandeurs pouvant prendre des valeurs dans des intervalles variés, il convient de leur adapter des valeurs aléatoires prises dans ces intervalles. L'introduction du typage de donnée décrite au paragraphe précédent peut permettre la création de nouveaux terminaux remplaçant le R de l'article étudié. Par exemple, l'utilisation d'un RCOL, RHEIGHT et RORIENT, retournant chacun des valeurs aléatoires prises dans les intervalles de numéro de

colonnes (1 à 10) de hauteur (1 à 20) et d'orientation (1 à 4) permettrait de cadrer les valeurs arguments des fonctions concernées.

5.2.4 Multi-RPBs et ADFs

Une évolution majeure dans le travail proposé est l'introduction de RPB différents pour chaque pièce. Cette approche permet de distinguer la stratégie à adopter en fonction de la pièce présente. Elle a l'avantage d'obliger le programme à considérer chaque pièce explicitement. Cela permet notamment de ne plus utiliser le terminal PIECE, qui, d'une part, introduit d'autant plus de bruit dans les programmes que les données ne sont pas typées, et d'autre part ne garantit pas la distinction de stratégie selon la pièce courante. Dans les travaux présentés plus loin, tous les systèmes utilisent un RPB distinct pour chaque pièce.

Une autre piste potentielle serait d'implémenter une RPB pour déterminer l'orientation de la pièce. Cette piste semble cependant relativement ardue, car orientation et position ne sont pas indépendantes pour le choix optimal du placement d'une pièce.

Les ADFs (« automatically defined functions ») pourraient également être un plus pour la modélisation de ce problème. Comme expliqué par l'auteur, cependant, l'émergence de bonnes stratégies semble plus longue avec l'utilisation de ce type de fonctions. Par ailleurs, définir ce genre de fonctions demande de paramétrer son nombre d'arguments, et éventuellement leurs types selon que le système et typé ou non. En d'autres termes, cela suppose d'avoir une petite idée du genre de fonction qui pourrait être utile au problème, ce qui n'est de prime abord pas évident dans le problème traité ici.

5.3 Implémentation et résultats

L'implémentation du programme génétique est relativement sommaire et présente à des fins d'illustration. Elle utilise ECJ,

5.3.1 ECJ : Evolutionnary Computing in Java

Le système utilisé pour l'implémentation de la programmation génétique est ECJ [4] Cette bibliothèque écrite en Java est très complète, et permet d'implémenter de nombreux problèmes par algorithmes génétiques, programmation génétique ou encore stratégies d'évolution type (*mu,lambda*) et (*mu+lambda*). A titre indicatif, il est très bien documenté et disponible à l'adresse indiquée en référence. Les détails techniques de ce système ne sont pas détaillés ici.

5.3.2 Résultats obtenus avec les ensembles de l'article

Dans un premier temps, les expériences menées suivent les conditions décrites dans l'article. Par manque de temps, le nombre de tests par expérience est cependant relativement réduit, ce qui peut expliquer la divergence avec les résultats de l'article étudié. Les résultats obtenus pour plusieurs séries de tests sont décrits dans le tableau ci-dessous :

Conditions de test	Meilleur individu	
	Fitness	Lignes
2 pièces	0	40
3 pièces	8500	34
4 pièces	13750	25
5 pièces	15250	19
6 pièces	14750	21
7 pièces	17500	10

Les divergences avec les résultats obtenus par Yurovitsky pour les expériences à 3 et 4 pièces peuvent s'expliquer par plusieurs causes. Tout d'abord, on a utilisé ici la taille normale de la zone de jeu du Tetris. Comme expliqué en §5.1.1, cela peut fortement influencer l'existence de solutions « faciles » à découvrir. Par ailleurs, le nombre de tests pour chacune des expériences ne dépasse pas 10, si bien qu'il est difficile d'évaluer les résultats moyens.

A titre indicatif, voici ci-dessous les 7 meilleurs RPBs obtenus pour l'une des expériences à 7 pièces, sous leur forme de S-expression :

```
Tree 0: // carré
 (ifLower (ifLower (ifLower 8 3 9 6) (add
     deepest deepest) (setrot 4) (ifLower (setrot
     deepest) (add deepest deepest) 3 (ifEqual
     6 5 7 0))) (ifEqual (sub 3 0) (sub 4 7) (add
     (ifLower 4 0 5 0) (add 1 deepest)) (setrot
     1)) (add (ifEqual deepest 1 1 deepest) 0)
     (add deepest deepest))
Tree 1: // barre
 (ifEqual (profile (add 6 2)) (sub (setrot
     (ifLower 7 7 1 3)) (ifEqual (sub 8 (ifEqual
     6 3 8 1)) (sub 2 8) (sub 8 9) (profile 0)))
     (setrot (setrot (ifLower 7 7 1 3))) (setrot
     (setrot 1)))
Tree 2: // T
 (ifLower 9 3 deepest deepest)
Tree 3: // Z
 (ifLower (ifLower 8 8 4 (ifLower 9 3 deepest
     deepest)) (ifEqual (sub 3 0) deepest (ifEqual
     deepest 1 1 deepest) (setrot 1)) (add (sub
     8 9) (sub 8 9)) (ifLower (ifLower 8 8 4 deepest)
     (ifEqual (sub 3 0) (sub 4 7) (profile 4)
         (setrot 1)) (add (sub 8 9) (sub 4 3)) (add
     (ifLower 4 0 5 0) (add 1 deepest))))
Tree 4: // S
 6
Tree 5: // J
 (ifLower deepest 9 (ifLower (ifLower 8 (ifLower
     (ifLower 2 5 6 6) deepest (sub 0 0) (profile
     5)) 4 (ifLower 9 3 deepest deepest)) (ifEqual
     (add 9 1) (ifLower 9 3 deepest deepest) (add
     (ifLower 4 0 5 (add 9 1)) 9) (setrot 1))
     deepest (add (ifLower 4 0 5 0) (add 1 deepest)))
     deepest)
Tree 6: // L
 (ifEqual (add (ifEqual deepest 1 1 deepest)
     deepest) 1 1 deepest)
```

5.3.3 Résultats obtenus avec les ensembles de l'article et le typage des données

Dans cette série de tests, on impose un typage aux données manipulées. Ceci est une fonctionnalité permise par le package ECJ. On définit 3 types qui sont *nil*, *columnId* et *heightValue*, désignant respectivement les grandeurs génériques, les indices de colonne et les hauteurs de blocs. On définit également un « ensemble de types » appelé *any*. Un ensemble de types permet de regrouper des types sous une même appellation, ce qui permet par exemple de définir des fonctions génériques acceptant tous les types (comparateurs, etc.) tout en ayant des fonctions spécifiques. Dans ce cadre, on type les fonctions comme décrit dans le tableau ci-dessous :

Fonctions	Entrées	Sortie
ADD, SUB	(any, any)	any
IFEQ, IFLT	(heightVal, heightVal, columnId, columnId)	columnId
PROFILE	columnId	heightValue
ROTATE	any	any
C#	/	columnId
DEEPEST	/	columnId
RANDOM	/	any

Ce typage permet principalement d'éviter que des décisions ne soient prises en comparant des grandeurs sémantiquement différentes, à savoir des indices de colonne et des hauteurs de blocs.

Les résultats obtenus après typage sont les suivants :

Conditions de test	Meilleur individu	
	Fitness	Lignes
2 pièces, avec typage	0	40
3 pièces, avec typage	8750	29
4 pièces, avec typage	16500	10
5 pièces, avec typage	18750	5
6 pièces, avec typage	18500	6
7 pièces, avec typage	18500	5

Ils sont bien en deçà des résultats obtenus sans typage, ce qui laisse à penser que le typage n'apporte rien. On peut interpréter ce résultat de la manière suivante : les heuristiques trouvées sans typage permettant plus de mélange de valeurs, qu'elles aient un sens à être mélangées ou non. Le typage impose une contrainte qui limite ce « bruit » plus qu'il ne le canalise vers des solutions optimales. La question du typage pose par conséquent la question de la modélisation. Ci-dessous, on présente quelques résultats obtenus avec des ensembles de terminaux et de fonctions augmentés.

5.3.4 Résultats obtenus avec des terminaux et fonctions booléennes

Dans ces expériences, on introduit de nouveaux terminaux et de nouvelles fonctions afin de permettre plus de modularités dans la création des blocs conditionnels. Ainsi, les fonctions

IFEQ et IFLT sont remplacées par la fonction IF(bool, any, any) et les opérateurs EQUAL(any, any) et LOWER(any, any). Par ailleurs, on introduit les opérateurs booléens NOT(bool), AND(bool, bool) et OR(bool, bool). On espère de cette façon permettre une plus grande modularité dans les comparaisons, telle qu'une analyse de la zone de jeu plus précise soit possible.

Les résultats obtenus avec ces éléments sont récapitulés ci-dessous :

Conditions de test	Meilleur individu	
	Fitness	Lignes
2 pièces, avec typage et fonctions booléennes	3000	38
3 pièces, avec typage et fonctions booléennes	10500	26
4 pièces, avec typage et fonctions booléennes	17000	8
5 pièces, avec typage et fonctions booléennes	18750	5
6 pièces, avec typage et fonctions booléennes	18750	5
7 pièces, avec typage et fonctions booléennes	18750	5

Comme précédemment, les résultats obtenus sont très moyens. La modularité offerte par de nouvelles fonctions ne change rien, voir empire, les résultats déjà obtenus avec le typage de données. Etant conceptuellement pertinents, le manque de résultats de ces deux approches laisse à penser qu'il faut revoir de fond en comble les ensembles de fonctions et terminaux.

Une autre lacune de ces méthodes est peut-être le trop petit nombre de générations ou encore de la taille des arbres. En effet, étant donnée une pièce, le nombre de comparaison optimal à effectuer pour trouver un bon – en dur – est déjà conséquent. Les limitations sur les proportions des arbres peuvent par conséquent être une des limites du système.

Quoiqu'il en soit, la méthode proposée dans [1] semble marcher correctement, bien qu'encore loin des performances d'un joueur humain.

6 Conclusions

Ce document a présenté les travaux de recherche décrits dans [1], concernant la découverte d'heuristiques pour le jeu Tetris à l'aide des méthodes de programmation génétique. On a vu que les résultats obtenus étaient encourageants, bien que beaucoup de simplifications aient été faites dans la modélisation du problème. Après avoir critiqué cette approche à plusieurs niveaux, quelques pistes ont été présentées, dont certaines implémentées et testées.

Les approches proposées vont dans deux sens : le typage de données impose des contraintes sur l'organisation des nœuds dans les arbres, tandis que les fonctions booléennes proposent une plus grande modularité dans la création de blocs conditionnels. Aucune de ces deux approches ne donnent de résultats significativement bons.

Les résultats obtenus ne sont cependant pas à la hauteur des espérances que pouvaient susciter le travail de M. Yurovitsky. On est en droit de penser que les simplifications du problème y sont pour bonne partie, notamment en termes de réduction de la zone de jeu et du nombre de pièces impliquées.

Une approche négligée dans ce document est l'utilisation d'ADFs (fonctions), d'ADLs (boucles) et d'ADRs (récursions). Il conviendrait d'approfondir la pertinence de tels extensions dans le problème étudié ici.

L'approche d'un joueur humain lorsqu'il joue à Tetris utilise, par ailleurs, des informations sur la forme de la pièce à placer, ce dont ne disposent pas les programmes génétiques implémentés ici. Une autre piste de travail serait donc de rendre possible un sondage de la morphologie de la pièce en parallèle du sondage du paysage de la zone de jeu. De même, la possibilité de détecter des trous déjà présents dans les colonnes traitées pourrait s'avérer un atout déterminant pour l'optimisation d'heuristiques de jeu.

Références

[1] Yurovitsky M. 1995. Playing Tetris Using Genetic Programming. *Genetic Algorithms and Genetic Programming at Stanford 1995*,pp. 309-319, John R. Koza, 1995, Stanford Bookstore

[2] Kröger B & al. 1992. *Massive Parallel Genetic Packing*. IOS Press, G.L. Reijins and J. Luo, Editors

[3] Koza J. 1994. *Genetic Programming*. The MIT Press

[4] ECJ 11 (2004). Site officiel http://cs.gmu.edu/~eclab/projects/ecj/ (visité le 30 Mars 2004)

[5] Montana D. Strongly-Typed Genetic Programming. *Evolutionary Computation* 3(2), pp. 199-230