Algorithme génétique pour Pierre-Feuille-Ciseaux

Stéphane Graham-Lengrand
Département d'Informatique de l'X
stephane.lengrand@polytechnique.edu

12 février 2014

Difficulté : (*)

1 Introduction

Le jeu de Pierre-Feuille-Ciseaux peut sembler être un jeu de hasard. Effectivement, si les deux joueurs décident de choisir aléatoirement leurs actions avec une loi uniforme, le résultat est effectivement aléatoire, et c'est une bonne stratégie défensive.

Malheureusment, il est très difficile pour un humain de décider de ses coups avec un tirage uniformément aléatoire, entre autres à cause de ses biais psychologiques, la gestion de sa frustration, la volonté de casser les mauvaises séries, etc.

Il est donc tentant pour son adversaire d'essayer de déceler ses motifs de jeu pour prendre un léger avantage qui payera sur un grand nombre de tours. [1]

Ce projet vise à développer un programme pour cela, à base d'un algorithme génétique. Le projet sera développé en Java.

2 Mise en place d'une architecture de test

La première tâche du projet consiste à définir
- une interface Java pour modéliser (la stratégie d')un joueur
- un système de test, qui pourra initialiser deux joueurs qui implémentent l'interface ci-dessus, puis les faire jouer l'un contre l'autre un grand nombre de fois, comptabilisera les points, affichera les résultats...

On pourra représenter l'ensemble des coups $\{P, F, C\}$ (pour Pierre, Feuille, Ciseaux) par l'ensemble d'entiers $\{0, 1, 2\}$, et calculer le résultat de chaque tour à base de calculs modulo 3.

La deuxième tâche consiste à écrire un certain nombre de classes qui implémentent l'interface ci-dessus, dont
- une classe qui, pour chaque coup, demande à l'utilisateur quoi jouer [2]
- une classe qui tire au hasard ses coups, selon une loi de probablilté passée en paramètre
- une classe qui répète un cycle de coups donné comme paramètre
- d'autres classes implémentant des stratégies de test

Testez vos stratégies en les faisant jouer l'une contre l'autre.

1. Vous trouverez sur le web les nombreux sites webs des ligues et des championnats de Pierre-Feuille-Ciseaux, comme ceux de World RPS ou de la USARPS league.
2. Dans un premier temps au moins, une interface en ligne de commande suffira.

3 Représentation d'une stratégie, d'une population, etc

Les principes de ce projet sont dans l'article [ANC00] dsiponible sur le web.

On considère pour l'instant l'ensemble des stratégies qui déterminent le prochain coup à jouer en fonction des m derniers coups de l'adversaire. On commencera par $m = 3$.

Il s'agit donc d'une fonction de $\{P, F, C\}^3$ dans $\{P, F, C\}$; autrement dit, cette fonction doit indiquer une lettre P, F, ou C pour chacun des $3 \times 3 \times 3 = 27$ inputs possibles. On la représentera par une chaîne de charactères (P, F, ou C) de longueur 27, ou plus directement un tableau de taille 27 contenant des 0, 1, ou 2. Par ailleurs, pour que nos stratégies déterminent quel coup jouer dans les 3 premiers tours (pour lesquels "les 3 derniers coups de l'adversaire" n'a pas de sens), on rajoutera 3 cases au tableau (maintenant de taille 30) où l'on stockera les coups supposés de l'adversaires aux tours -1, -2 et -3.

Chaque stratégie va être notée en fonction de ses performances contre un adversaire fixé. Sur n tours de jeux on va noter combien de fois elle gagne et perd.

Ecrivez une classe pour représenter une population de stratégies à chacune desquelles est attribué un score (nombre entier) ; on pourra par exemple utiliser une `HashMap`.

Ecrivez également un programme d'*évaluation de population* qui, étant donnés
- un adversaire A,
- une stratégie particulière S,
- et une population de stratégies P (toutes initialisées à 0 dans la `HashMap`),

fait jouer A contre S sur n tours de jeu, mais calcule aussi à chaque tour, pour chaque stratégie dans P, le coup que celle-ci aurait joué si elle était à la place de S. Si celle-ci aurait gagné, son compteur augmente de 1, si elle aurait perdu, il baisse de 1, sinon il ne change pas.

A la fin des n tours, on peut alors regarder le score de chaque stratégie.

4 Opérations génétiques sur les stratégies

Vient alors la phase de transformation de la population de stratégies. Pour faire simple, la nouvelle population aura la même taille que l'ancienne.

Une proportion de l'ancienne population (les x% qui ont fait les meilleurs scores) est sélectionnée et reste dans la nouvelle population. Les autres stratégies sont remplacées par de nouvelles, obtenues à partir de la population sélectionnée, par deux mécanismes : le *croisement* et la *mutation*.

Pour le croisement, on produit une stratégie dont le code (le tableau de 30 cases) reprend une partie du code d'une stratégie sélectionnée, et une partie du code d'une autre stratégie sélectionnée (une manière simple de le faire est de prendre les p premières cases chez l'une, et les $30 - p$ autres cases chez l'autre, avec p tiré au hasard).

Pour la mutation, on produit une stratégie en changeant au hasard la valeur dans certaines cases du tableau, avec une faible probabilité.

La proportion de stratégies sélectionnées, le nombre de croisement, de mutation, etc seront des paramètres.

Ecrivez un programme pour transformer une population évaluée, selon ces mécanismes et ces paramètres (la nouvelle population doit avoir tous ses scores à 0).

5 Tests

Ecrivez un programme qui effectue en boucle l'évaluation d'une population et sa transformation.

A chaque nouvelle génération, il reste à choisir quelle stratégie va effectivement jouer contre l'adversaire lors de la phase d'évaluation, et ce choix est critique, puisqu'il peut influencer la stratégie de l'adversaire. On pourra prendre par exemple la meilleure stratégie de la population courante, mais aussi tester d'autres choix.

Testez la convergence éventuelle de la meilleure stratégie, ou à défaut, la convergence éventuelle de son score.

Après un nombre de générations passé à votre programme comme paramètre, la meilleur stratégie est sélectionnée pour combattre l'adversaire (rappelons qu'il s'agit d'un adversaire fixé depuis le début). Pour chacune de vos stratégies de test écrites dans la section 2, obtenez cette stratégie et jouez le match. Faites une analyse de vos résultats dans le rapport de projet.

6 Améliorations

Comme on l'a dit, la stratégie qui tire aléatoirement ses coups selon une loi uniforme est la plus efficace sur le plan défensif, car l'adversaire (qu'il soit humain, qu'il soit la stratégie obtenue par votre algorithme, etc) ne peut rien anticiper. La stratégie selectionnée par l'algorithme génétique ci-dessus n'étant pas cette stratégie, il est possible qu'elle soit vulnérable, si l'adversaire cherche à son tour à reconnaître vos motifs de jeu.

On peut alors embarquer dans nos stratégies une information, le "facteur de caprice", que l'on peut coder sur quelques cases supplémentaires au tableau (toujours contenant des 0, 1 ou 2). Il s'agit de définir une probabilité avec laquelle la stratégie joue, non pas le coup déterminé par la valeur contenue dans l'une de ses 27 premières cases, mais un coup tiré au hasard. C'est ce qui est proposé dans l'article [ANC00].

Implémentez ce raffinement et testez-le ; faites également jouer des stratégies avec facteur de caprice contre des stratégies sans facteur de caprice.

Maintenant que les mécanismes de l'algorithme génétique sont mis en place, on peut aussi varier les inputs de nos stratégies : au lieu de considérer les 3 derniers coups de l'adversaire, on peut en prendre plus. On peut aussi se dire que ce qui détermine le prochain coup de l'adversaire, c'est moins ses n derniers coups que les n derniers résultats (il a gagné / on a gagné / il y a eu égalité), ou une combinaison des deux.

Expérimentez ces alternatives, en prenant garde bien sûr à maintenir des temps de calcul raisonnables.

Références

[ANC00] F. Ali, Z. Nakao, and Y.-W. Chen. Playing the rock-paper-scissors game with a genetic algorithm. In A. Zalzala, editor, *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 741–745. IEEE Computer Society Press, 2000. http ://www.ia.urjc.es/grupo/docencia/ia4/material/articulo.pdf. 3, 6