UNIVERSITÉ DE SHERBROOKE

Apprentissage du blackjack par un joueur génétique

Projet – IFT630

présenté à
Gabriel Girard

Denis Gauthier (11 050 159)
Philippe Poulin (10 075 447)
Antoine Savage (10 096 479)

22/04/2013

Table des matières

Résumé du projet

Nous voulons créer un joueur artificiel qui pourra prendre de meilleures décisions qu'un joueur aléatoire au Blackjack. Le joueur joue une version simplifiée du Blackjack, dans laquelle on ne peut que prendre une carte ou rester avec la main courante, et où on ne peut avoir plus de cinq cartes en main. Le joueur prend ses décisions en fonction d'une politique construite à l'aide d'un algorithme génétique parallélisé.

D'abord, le joueur gagne si la valeur de sa main est un de plus que celle du croupier. Ainsi, en cas d'égalité, le croupier gagne toujours. De plus, nous ne considérons pas la différence entre une main « Blackjack » (un as et une autre carte) et une main de valeur 21 (trois cartes ou plus).

Un joueur est modélisé par une suite de calculs effectués sur l'état courant de la partie (une carte du croupier et les cartes du joueur) pour obtenir la décision du joueur (prendre une carte ou rester). En appliquant un algorithme génétique pour faire évoluer cette politique, nous espérons obtenir une politique plus avantageuse qu'un joueur aléatoire (qui perd en moyenne le deux tiers de ses parties). La valeur d'adaptabilité d'un certain joueur sera donnée par le pourcentage de parties gagnées contre le croupier, qui sera calculée en simulant un nombre arbitraire de parties. La parallélisation du calcul de cette valeur pour tous les joueurs d'une population nous permettra d'accélérer d'un facteur non négligeable la génération d'une nouvelle population.

Description de l'application

Nous avons utilisé le langage Java pour programmer notre algorithme génétique. L'application s'utilise en deux temps. D'abord, il est possible de générer un joueur avec l'algorithme de programmation génétique (generate_player.jar). Ensuite, il est possible de faire jouer le joueur généré contre le croupier pour un nombre de mains variable ou de jouer soi-même contre le croupier (play_blackjack.jar). Pour faciliter la configuration des paramètres, deux fichiers sont fournis (generate_player.bat et play_blackjack.bat).

Pour configurer la génération d'un joueur (generate_player.bat), les paramètres sont :

- Taille de la population
- Nombre maximum de générations
- Nombre de mains simulées pour calculer la valeur d'adaptabilité
- Critère de satisfaction (Nombre de fois consécutives où la différence de la valeur d'adaptabilité est de moins de 1%)
- Chance de reproduction (Le code génétique de deux parents est mélangé, sinon un parent est cloné)
- Chance de mutation

Pour jouer au blackjack (play_blackjack.bat), les paramètres sont :

- Le nombre de parties à simuler avec le meilleur joueur de la dernière génération
    - 0 activera le mode console, où un joueur humain peut jouer contre le croupier (20 parties)

Développement de l'application

Un algorithme génétique a été utilisé au départ pour générer une politique efficace. Nos politiques étaient représentées par une liste d'états possibles (les cartes en main et la carte visible du croupier), auxquelles était associée une décision (prendre une carte ou rester). Avec cette représentation, il était facile de générer des joueurs aléatoires en assignant une décision au hasard à chaque état. De façon générale, un joueur aléatoire avait un taux de victoires (notre valeur de fitness) d'environ 30%. Ainsi, nous générions une population de joueurs aléatoires dans le but de la faire évoluer et d'obtenir un joueur optimal. Cependant, nous n'avons pas réussi à obtenir un joueur visiblement plus efficace qu'un joueur aléatoire.

Nous expliquons ce résultat par le fait que les joueurs étaient représentés par un espace d'états beaucoup trop grand. Ainsi, les joueurs étaient très peu sensibles aux modifications induites par la reproduction ou la mutation. Puisque la population initiale était composée de joueurs aléatoires déjà efficaces, les fines améliorations n'étaient pas prises en compte lors de la génération d'une nouvelle population.

Nous avons donc choisi de modifier notre représentation et d'utiliser la programmation génétique pour obtenir un joueur efficace. Un joueur est donc représenté par une suite de calculs effectués à partir de la valeur de chaque carte de la main de celui-ci, et de la valeur de la carte du croupier. La population initiale est donc une suite de joueurs effectuant des calculs aléatoires sur ces valeurs; ils sont donc peu efficaces et très sensibles aux améliorations. En effet, le taux moyen de victoires d'une population aléatoire est d'environ 20%.

Pour notre situation, on considère qu'une simulation de 10 000 mains est suffisante pour calculer la fitness d'un joueur. Nous avons calculé que le calcul de fitness a une variation de moins de 5% avec cette valeur, que nous considérons acceptable.

Représentation d'un joueur avec la programmation génétique

Les programmes générés par notre application respectent une grammaire maison similaire au langage Lisp, mais énormément simplifiée. La grammaire est la suivante:

program -> expr
expr -> (expr0)
expr0 -> if_expr | op_expr | cte_expr | var_expr

if_expr -> expr expr expr
op_expr -> op expr expr
cte_expr -> -1000 | -999 | ... | -1 | 0 | 1 | ... | 999 | 1000
var_expr -> d | h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7 | h8 | h9 | h10 | h11

op -> (op0)
op0 -> + | - | *

Une if_expr évalue la première expr, et s'il s'agit d'une valeur inférieure à zéro, évalue la seconde expr et retourne cette valeur, sinon évalue la troisième expr et retourne cette valeur. Par exemple:

if_expr -> e0 e1 e2 est équivalent à :

```
if (e0 < 0) {
  return e1
} else {
  return e2;
}
```

Une op_expr évalue d'abord ses deux opérandes expr et retourne le résultat de op avec ces deux opérandes. Par exemple:
op_expr -> + e1 e2 est équivalent à:
Return e1 + e2

Une cte_expr retourne un litteral entier constant, entre deux bornes arbitraires (-1000 et 1000)
Une var_expr retourne la valeur de la variable correspondante. Par exemple:
d retourne la valeur de la carte du croupier
h0 retourne la premiere carte du joueur
h1 retourne la seconde carte du joueur
h2 retourne la troisieme carte du joueur

Puisque le nombre maximal de cartes pouvant être contenu dans une main de blackjack (un seul paquet de 52 cartes en jeu) est de 11 cartes sans dépasser 21. On prévoit donc la syntaxe pour pouvoir évaluer jusqu'à 12 variables de main et une variable de dealer.

Voici un exemple de programme (la politique du dealer):

```
((-)((+)(((-) ((+) (1) (h0)) (10)) ((+) (1) (h0)) (10))((+)(((-) ((+) (1) (h1)) (10)) ((+) (1) (h1)) (10))((+)(((-) ((+) (1) (h2)) (10)) ((+) (1) (h2)) (10))((+)(((-) ((+) (1) (h3)) (10)) ((+) (1) (h3)) (10))((+)(((-) ((+) (1) (h4)) (10)) ((+) (1) (h4)) (10))((+)(((-) ((+) (1) (h5)) (10)) ((+) (1) (h5)) (10))((+)(((-) ((+) (1) (h6)) (10)) ((+) (1) (h6)) (10))((+)(((-) ((+) (1) (h7)) (10)) ((+) (1) (h7)) (10))((+)(((-) ((+) (1) (h8)) (10)) ((+) (1) (h8)) (10))((+)(((-) ((+) (1) (h9)) (10)) ((+) (1) (h9)) (10))((+)(((-) ((+) (1) (h10)) (10)) ((+) (1) (h10)) (10))(((-) ((+) (1) (h11)) (10)) ((+) (1) (h11)) (10))))))))))))(17))
```

Il évalue d'abord la valeur de chaque carte de la main avec la ligne suivante:

```
(((-) ((+) (1) (h0)) (10)) ((+) (1) (h0)) (10))
```

Version expansée :

```
( [           // if_expr entre [ et ]
  ((-)
  ((+) (1) (h0)) // Un de plus que l'id de la carte (ACE = 0, 2 = 1, etc. absence de carte (-1) = 0)
  (10))        // Soustrait 10 au total. Si la valeur est inferieure a 10 (différence négative)
  ]
((+) (1) (h0))   // Retourner la valeur de la carte
(10)           // Retourner 10. JACK, QUEEN, KING et 10 retourneront dont tous une valeur de 10
)
```

Il somme ensuite la valeur de chacune des 12 cartes (les cartes absentes valent 0), et soustrait 17 à cette valeur. Si on retourne une valeur négative, cela correspond à prendre une autre carte. Si la valeur retournée est positive, cela correspond à rester. On a donc un joueur qui pige une carte si le total de la main est inférieur à 17 et reste si le total est égal ou supérieur à 17. C'est donc la politique du croupier.

On effectue les mutations dans les arbres syntaxiques de la façon suivante:

Chaque nœud de l'arbre (un nœud est une expr) a une certaine probabilité de simplement transmettre la mutation à un de ses enfants. Les if_expr et les op_expr peuvent donc transmettre la mutation à l'un de leurs 3 ou 2 sous-expressions, respectivement. Les var_expr et les cte_expr n'ont pas d'enfant et appliquent donc simplement la mutation à eux-mêmes inconditionnellement.

Si un nœud ne transmet pas la mutation à un sous-nœud, il peut se modifier de la façon suivante selon son type:

if_expr : Intervertir deux des trois sous-expressions
op_expr : 1) Intervertir les deux sous-expressions OU BIEN 2) propager la mutation à l'operateur (op)

var_expr: devient une autre des variables, aléatoirement. Par exemple (h0) -> (d), (h2) -> (h3) et même (h0) -> (h0)
cte_expr: devient une autre des valeurs constantes. Par exemple (1000) -> (263), (-45) -> (354) et même (3) -> (3)
op : devient un autre operateur. Par exemple, (+) -> (*), (-) -> (+) et même (+) -> (+)

On effectue les croisements en sélectionnant un nœud N d'un des deux parents, et l'on fournit ce nœud à l'autre parent qui produit un nouvel arbre syntaxique dans lequel un nœud aléatoire a été remplacé par N.

Résultats (IA)

Taille de la population	Chance de reproduction	Chance de mutation	Pire fitness de la population*	Meilleure fitness de la population*	Fitness moyenne de la population*
10	0.5	0.1	0.296	0.394	0.373
30	0.5	0.1	0.376	0.394	0.384
50	0.5	0.1	0.373	0.392	0.385
100	0.5	0.1	0.369	0.399	0.383
50	0.5	0.05	0.373	0.398	0.383
50	0.5	0.1	0.374	0.393	0.384
50	0.5	0.2	0.371	0.395	0.382
50	0.5	0.7	0.371	0.395	0.382
50	0.9	0.1	0.374	0.394	0.383
50	0.1	0.01	0.374	0.392	0.383
50	0.1	0.01	0.374	0.396	0.383

* : Après 100 générations, avec 10000 mains simulées

Après nos tests, nous obtenons dans tous les cas un joueur optimal avec un taux de victoires d'environ 39%. La variation des paramètres a semblé faire varier la vitesse de convergence vers cette valeur, sans nous permettre d'obtenir un meilleur joueur.

Selon une étude de l'université de Galway, en Irlande[1], un joueur artificiel de blackjack (utilisant la même version simplifiée que notre projet) peut obtenir un taux de victoires entre 30% et 44%, selon la stratégie utilisée (la meilleure étant la stratégie de Hoyle, qui tient compte de la carte du croupier et des as en main, suivie de près par le réseau de neurones artificiel des chercheurs). Notre joueur s'inscrit donc bel et bien à l'intérieur des limites obtenues par cette étude. Il serait intéressant de développer un réseau de neurones utilisant des techniques d'évolution génétique pour voir s'il est possible de dépasser la limite de 39% obtenue ici.

[1] Dara Curran and Colm O'Riordan, Evolving Blackjack Strategies Using Cultural Learning in Multi-Agent Systems, Dept. of Information Technology, National University of Ireland, Galway, http://ww2.it.nuigalway.ie/cirg/localpubs/CurranICANNGA2005.pdf (9 avril 2013)

Résultats (programmation parallèle)

En théorie, notre programme arrive aux résultats désirés sans avoir besoin de parallélisme. Cependant, un grand nombre d'opérations identiques s'effectuent séquentiellement, ce qui alourdit beaucoup le temps de virement du programme. Deux options de parallélisation s'offrent à nous : paralléliser la simulation de chacune des mains, puisqu'elles sont totalement indépendantes, ou paralléliser le calcul d'adaptabilité de chacun des joueurs (la simulation de toutes les mains d'un joueur).

Dans le premier cas, il y aura autant de fils d'exécution qu'il y a de mains à simuler. Par exemple, une population de 10 joueurs avec 100 mains simulées par joueur générera 1000 fils d'exécution. On peut donc s'attendre à un gain très faible lorsque le nombre de joueurs et de mains simulées restent relativement bas (sous l'ordre de 100 fils d'exécution), mais un ralentissement considérable dès que l'on veut utiliser le programme à plus grande échelle.

Dans le deuxième cas, un fil d'exécution sera lancé par joueur. En utilisant l'exemple précédent, une population de 10 joueurs avec 100 mains simulées par joueur générera 10 fils d'exécution. Cette version sera probablement plus appropriée pour une utilisation à grande échelle, puisque pour obtenir une valeur d'adaptabilité avec un faible taux d'erreur, il est nécessaire de simuler un très grand nombre de mains (au moins 10 000 mains par joueur pour un taux d'erreur de moins de 5% en moyenne).

Nous avons comparé le temps de virement d'une version séquentielle du programme avec les deux versions parallèles mentionnées ci-haut, afin de connaître le gain possible et les limites de la parallélisation (parallélisation triviale dans le cas présent) dans le tableau ci-dessous. N.B. : les tests ont été effectués sur une machine à 8 cœurs, et 10 générations sont générées pour obtenir le temps de virement. De plus, N/A signifie que le test n'a pas été effectué puisque le temps requis était trop long, et donc que le test n'était plus pertinent.

Taille de la population	Nombre de mains simulées par joueur	Élément parallélisé	Temps de virement* (millisecondes)	Pourcentage de gain sur le temps séquentiel
10	10	Aucun	62	1
		Mains	120	0.51
		Joueurs	46	1.35
10	100	Aucun	300	1
		Mains	805	0.37
		Joueurs	183	1.64
10	1000	Aucun	775	1
		Mains	6031	0.13
		Joueurs	1149	0.67
100	10	Aucun	222	1
		Mains	702	0.32
		Joueurs	161	1.38
100	100	Aucun	528	1
		Mains	5479	0.10
		Joueurs	524	1.01
100	1000	Aucun	4676	1
		Mains	56936	0.08
		Joueurs	4590	1.02
1000	1000	Aucun	52667	1
		Mains	N/A	N/A
		Joueurs	50384	1.05
2	10000	Aucun	2212	1
		Mains	N/A	N/A
		Joueurs	1754	1.26
4	10000	Aucun	1367	1
		Mains	N/A	N/A
		Joueurs	1292	1.06
8	10000	Aucun	3197	1
		Mains	N/A	N/A
		Joueurs	3948	0.81
16	10000	Aucun	7410	1
		Mains	N/A	N/A
		Joueurs	8120	0.91

Tableau 1 – Comparaison de différents types de parallélisation

Le premier constat est que la parallélisation des mains n'apporte pratiquement aucun avantage, tel que prévu. Cela s'explique par le fait qu'un nombre élevé de fils d'exécution qui font un calcul simple et rapide et qui s'exécutent sur une machine avec un nombre de cœurs beaucoup plus petit que le nombre de fils eux-mêmes nécessite beaucoup de gestion. En effet, le temps requis pour gérer tous les fils (par exemple, lors des changements de contexte) est beaucoup plus grand que le temps requis par un fil pour faire son calcul. Dans notre cas, ce temps est même plus grand que le temps requis pour faire le calcul séquentiellement.

Le second constat est que la parallélisation des joueurs apporte un gain le plus important lorsque le nombre de joueurs est relativement faible. Un nombre plus élevé de joueurs n'apporte pratiquement aucun gain. Effectivement, avoir plus de fils d'exécution que de cœurs pose le même problème que la parallélisation des mains, mais dans une moindre mesure.

Enfin, pour profiter au maximum de la parallélisation, il faudrait pouvoir utiliser autant de fils d'exécution qu'il y a de cœurs, indépendamment du nombre de joueurs ou de mains simulées. Ainsi, le processeur pourrait être utilisé autant que possible, tant que la communication reste à son minimum. Par exemple, nous pourrions désigner un fil maître qui compile les résultats, et chaque thread se ferait assigner un certain nombre de mains à simuler pour un joueur particulier, retournerait le résultat au maître et demanderait plus de travail.