IA - rapport

Gwendal PRAT Clément THION

Présentation du problème

L'objectif de cette matière est de comprendre et manipuler les bases et concepts d'une intelligence avancée. Pour cela, nous avons créé des algorithmes permettant de "jouer" à différents jeux simples, et dont le but est de gagner contre un adversaire Humain ou Virtuel.

On appelle "intelligence avancée" un algorithme de calcul, auquel on attribue une mémoire dans laquelle on va pouvoir stocker des règles et informations qui pourront être mises à jour par la suite au fur et à mesure de l'utilisation de l'algorithme. Cet algorithme doit donc être capable d'apprendre et de se rappeler les actions lui permettant de remplir la tâche demandée.

Description de notre approche du problème

Nous savons qu'il existe différentes approches pour faire face à un problème :

- Approche informatique : résolution d'un problème grâce à la mise en place d'un algorithme.

- Approche données ou statistiques : on n'a pas accès au problème mais seulement à des données. L'algorithme créé utilise des outils statistiques tels que la régression, la classification, etc … L'idée ici est de prédire de nouvelles données.
- Approche évolutionnaire : se base sur la théorie de l'évolution ; l'agent est performant s'il peut survivre et se reproduire.
- Approche par agent.

Nous avons choisi de prendre une approche par agent. Dans cette approche, nous disposons d'un environnement particulier (un jeu), c'est-à-dire que les règles sont stables au cours du temps (on ne change pas les règles du jeu). Le jeu pour lequel a été pensé notre agent est le morpion. L'environnement conciste alors en un damier carré qui pourra être de 3, 5 ou 7 case de côté ; le nombre de pierres à aligner sera de 3, 4 ou 5. Le damier pourra aussi être de forme plan ou torique.

De plus, il est possible de disposer de plusieurs agents (les joueurs) en interaction, mais nous avons choisi de nous limiter à un seul.

Les caractéristiques principales du système « intelligent » produit

On rappelle qu'un système intelligent doit être capable d'apprendre et de se souvenir : il doit donc posséder une mémoire. Il nécessite d'autre part au moins l'une des caractéristiques suivantes.

La rapididité d'exécution

La rapidité d'exécution d'un agent désigne sa vitesse de calcul. Un agent qui met 3 semaines à jouer un coup au jeu du morpion n'est a priori pas très intéressant. Pire, dans des jeux où le temps est une variable de victoire ou de défaite, c'est indispensable. Dans un jeu type RPG par exemple, si l'agent met trop de temps à calculer ses décisions, il va se prendre un coup d'épée… Autre exemple pour un jeu de gestion de ressource dans le style simulation OSCAR : si l'agent met trop de temps à déterminer qu'une case contient de la nouriture et qu'il faut s'y déplacer, il meurt.

Pour augmenter la rapiditer d'un Agent, on doit limiter et optimiser au maximum les calculs. Si la rapidité est la caractéristique qui nous importe le plus, on pourra envisager de sacrifier certains calculs, au prix d'une baisse de l'efficacité de l'agent.

En particulier, une très bonne manière d'optimiser la rapidité d'exécution est d'utiliser une mémoire pour sauvegarder le résultat des calculs afin d'éviter d'avoir à les refaire dans le future quand on se retrouve dans un état déjà rencontré dans le passé.

Pour le cas du morpion, c'est un jeu très très simple avec relativement peu d'états possibles, la quantité de calculs à réaliser est donc moindre face à un jeu plus complexe, par exemple du type échecs. D'autant qu'aujourd'hui les machines que l'on utilise ont en général une vitesse de calcule d'au moins un milliard et demi de calcul par seconde ; même sachant que windows nous en mobilise un bon milliard, cela nous en laisse quand même pas mal pour le morpion.

L'efficacité du travail réalisé

L'efficacité d'un agent désigne sa capacité à produire aussi surement que possible un résultat convenant le mieux possible à l'objectif qui lui est fixé, gagner dans le cas d'un jeu.
On juge que la vitesse ne fait pas tout, mais que gagner le plus de fois possible peut être un aspect important. En effet, agir de manière spontanée et donner un point à son adversaire n'est pas la meilleure option. Dans le cas du Morpion, il est défavorable de ne pas remplir un alignement, ou bien de ne pas empêcher son adversaire de remplir un alignement.

De plus, on peut se demander si notre agent doit être efficace face à un seul ou plusieurs joueurs, dans le cas d'un jeu à plus de deux joueurs (ce qui n'est pas notre cas à nous).

La flexibilité de travail

La flexibilité d'un agent désigne sa capacité à s'adapter à n'importe quel type d'environnement, donc dans notre cas à n'importe quel type de jeu. On peut choisir de créer un agent étant un expert dans sa tâche, mais qui ne soit pas capable de réaliser une autre tâche, même très similaire ; ou bien nous pouvons créer un agent un peu moins performant, mais capable de réaliser une multitude de tâches différentes.

Dans notre cas, nos agent est plutôt flexible, pouvant jouer à n'importe lequel des jeux à condition qu'il respecte certaines règles de construction, similaire à celle du morpion.

Les différents agents modélisés

Randy

Un joueur qui se base sur une stratégie d'aléatoire : il renvoie de manière spontanée une réponse, sans comparer les bénéfices et les risques. Il n'a aucune mémoire et ne fait preuve d'aucune stratégie que l'on puisse qualifier d'intelligente.

NegAlphaBeta

Un joueur qui synthétise MinMax, NegaMax et AlphaBeta : les résultats obtenus sont les mêmes que ceux des algorithmes cités, à la différence que le calcul est nettement plus rapide.

MinMax
Un joueur qui se base sur une stratégie de Minimisation des gains adverses, et Maximisation de ses propres gains, en parcourant tous les possibilités jusqu'à un tour de la partie donné.

NegaMax
Un joueur s'inspirant du joueur MinMax, mais qui fonde son fonctionnement sur la règle suivante : min(a,b) = -max(-a,-b) .

AlphaBeta
Un joueur s'inspirant du joueur MinMax, mais qui élague les possibilités inutiles, en se basant sur un intervalle de type [alpha;beta], où alpha correspond à la maximisation du profit, et beta correspond à la minimisation du gain adverse, jusqu'à ce que alpha et beta se confondent ou se croisent, ou qu'il n'y est plus de possibilités restantes.

NegAlphaBeta_MC

Ce joueur reprend la logique de l'algorithme original NegAlphaBeta, mais rajoute une dimension de simulation. Ainsi, les bornes calculées dans l'intervalle [alpha;beta] sont modélisées en fonction du résulat des simulations.

UCB

Le joueur UCB, ou bandit manchot, fonde sa stratégie sur la simulation. Il se base sur les résultats de ses simulations (gain, perte, égalité), et affine son choix au fur et à mesure des simulations, en donnant du poids aux actions qui favorisent la victoire.

Comment utiliser notre agent

- de vos choix, et en quoi ces choix sont justifiés pour la résolution du problème
  Pour utiliser nos agents, on juge pertinent de devoir préciser à cet agent comment il s'appelle, ainsi que le jeu auquel il doit jouer. D'autre part, chaque agent aura une stratégie unique, et aura alors besoin (ou pas) de certaines informations prérequises.
- de l'utilisation de votre code : ce que l'on doit faire pour voir votre réalisation fonctionner

Agent	Paramètres Obligatoires	Paramètre Personnalisé
AlphaBeta	Nom du joueur, Nom du Jeu	Profondeur
IterativeDeepening	Nom du joueur, Nom du Jeu	Profondeur, Nombre de secondes de travail accordées
MinMax	Nom du joueur, Nom du Jeu	Profondeur
Negamax	Nom du joueur, Nom du Jeu	Profondeur
NegAlphaBeta	Nom du joueur, Nom du Jeu	Profondeur
NegAlphaBeta_MC	Nom du joueur, Nom du Jeu	Profondeur, Nombre de simulations à réaliser
NegAlphaBeta_Memory	Nom du joueur, Nom du Jeu	Profondeur
NegAlphaBeta_Memory_MC	Nom du joueur, Nom du Jeu	Profondeur, Nombre de simulations à réaliser
Randy	Nom du joueur, Nom du Jeu	Aucun
Randy_MC	Nom du joueur, Nom du Jeu	Nombre de simulations à réaliser
UCB	Nom du joueur, Nom du Jeu	Nombre de simulations à réaliser

Conditions d'utilisation

Notre agent ne traite que des jeux hérité de la classe Game, c'est à dire des jeux à exactement deux joueurs, dont on peut connaître toutes les informations sur son état courant : à qui est le tour de jouer, quelle est la configuration du jeu (son "state").

Les tests

> On doit se donner une définition de l'intelligence, pour ensuite dire comment la tester et la mesurer.

Approche pour tester l'intelligence

On peut se donner comme premiers indicateurs d'une certaine forme d'intelligence la capacité à résoudre des problèmes, rapidement, efficacement, avec un minimum d'effort (*i.e peu de calculs)*, et pour des problèmes dont la forme varie.
Appliqué au cadre de nos programmes, on parlera des caractéristiques de rapidité, efficacité, et flexibilité.
On s'attend à ce qu'un agent supposé "intelligent" soit globalement plus performant sur ces trois caractéristiques qu'un agent supposé non intelligent.

Cependant ce critère ne nous satisfait pas totalement, étant donné qu'on peut concevoir des agents performants dans ces trois critères, mais qui soient totalement déterministes et donc difficilement qualifiables d'intelligents (c'est un point de vue). Par exemple, l'algorithme NegAlphaBeta est totalement déterministe mais fournit de très bonnes performances.

On ajoute une condition à la définition d'agent intelligent : utilisation d'une stratégie basée sur la mobilisation d'une mémoire ou d'une simulation.

On s'attend à ce qu'un agent utilisant une mémoire, ou des simulations, ait des meilleures performances qu'un algorithme déterministe.

Application

Une fois posés ces critères d'évaluation, on propose la stratégie de test suivante. On va se faire affronter nos agents "intelligents" contre des agents dit non intelligents, sur le jeu du Morpion, et mesurant à chaque fois les statistiques des parties, afin de comparer les niveaux de rapidité, d'efficacité et de flexibilité.

Pour la rapidité on regarde tout simplement le temps nécessaire en moyenne à un agent pour prendre une décision. On se donne pour valeur témoin minimale la rapidité de *Randy*, puisque c'est un algorithme qui renvoie une réponse aléatoire spontanée. en valeur maximale, on se réfère à *MinMax* qui est un algorithme très gourmand en calculs puisqu'il étudie toutes les possibilités d'action, et est donc le plus lent de nos algorithmes.

Pour l'efficacité on regarde en moyenne le taux de réussite. On se donne pour valeur témoin minimale d'efficacité celle de *Randy* : c'est en effet un programme qui est très peu efficace par construction (choisit ses coups au hasard). En valeur maximale, on se base sur MinMax, qui nous donne à chaque fois une valeur exacte, du fait qu'il compare toutes les possibilités.

Pour la flexibilité, il se trouve que tous nos agents ont été conçus pour pouvoir jouer à tous les jeux hérités de *abstract_game*. Ils auront donc tous la même flexibilité sur tous ces jeux, on pourra dire que cette caractéristique est constante pour tous nos agents. On ne va donc pas la tester ici.

Agents “intelligents” qu’on evalue :

- Negalphabeta_Memory => intelligence basée sur la mémoire
- UCB => intelligence basée sur la simulation et les statistiques (*loi forte des grands nombre*)
- Negalphabeta_Memory_MC => intelligence basée sur la simulation et la mémoire

Pour tester nos agents, on utilise entre autre le fichier *main_parties.py* afin de simuler des parties.

Résultats des test

Test de rapidité

Randy VS NegAlphaBetaMC

On modifie légèrement les fonctions *manche* et *partie* de *main_parties* pour obtenir le temps moyen de prise de décision de nos agents.
Après avoir réalisé des tests de rapidité entre *Randy* et *NegAlphaBetaMc*, nous avons obtenu le résultat suivant : Randy effectue ses calculs de manière quasi-instantanée (10^{-6} seconde au compteur par approximation informatique), tandis que *NegAlphaBeta_MC* effectue ses calculs en 0,00019 seconde environ et en moyenne.
Nous pouvons donc en conclure que *Randy* est bien l’algorithme le plus rapide des deux. Nous supposons qu’il est également le plus rapide de tous nos algorithmes.

Tableau récapitulant les temps de calculs (en seconde) :

Adversaire / Joueur	**NegAlphaBeta_memory**	**NegAlphaBeta_memory_MC**	**UCB**
Randy	?	0.00019	0.033
MinMax	?	?	0.006 (pourMinMax) VS 0.031 (pour UCB)

Les temps récoltés ci-dessus sont issus de simulations de parties.
A chaque fois, nous prenions en référence une simulation avec 2 parties (2 fois 2 manches), avec une profondeur de 2, et un nombre de simulations de 500 pour les versions Monte-Carlo.
Cependant, nous avons dû faire face à de nombreux échecs dans nos tentatives. En effet, il nous arrivait souvent de ne pas pouvoir avoir de résultats, à cause d'une erreur récurrente (issue du Exception de la méthode "manche", située dans le fichier main_parties).
Une donnée n'a donc pas pu être mesurée, mais on estime que MinMax s'exécute à l'ordre du centième de seconde, tandis que *NegAlphaBeta_Memory_MC* s'exécute à l'ordre du millième de seconde.

En somme, on peut déjà dire que *Randy* est bien l'agent le plus rapide.
MinMax, quant à lui, n'est finalement pas le plus lent, puisque UCB montre un temps de traitement nettement supérieur (c'est d aux simulations).

Théoriquement, on devrait observer une différence notable sur la rapidité des agents utilisant une mémoire (*NegAlphaBeta_memory*) et ceux basés sur de la simulation. En effet par défaut *NegAlphaBeta* a une mémoire vide au départ, ce qui fait qu'il exécute systèmatique des calculs, mais au bout d'un certain temps il n'en fait presque plus car ne fait qu'utiliser les résultats qu'il a enregistrés dans sa mémoire, et est dont beaucoup plus rapide à prendre des décisions, tandis que les agents basés sur la simulation (*UCB*) font systématique beaucoup de simulations, ce qui implique beaucoup de calculs.

Test d'efficacité

Nous n'avons malheureusement pas réussi à exécuter la totalité de nos tests (pour cause d'une erreur récurente lors de l'exécution de la fonction *partie*). Cela veut donc dire que nos agents ne sont pas totalement robustes, bien qu'ils passent tous les tests d'évaluation officiels.

Cependant, nous savons que les classes étudiées (*UCB, NegAlphaBeta_memory et NegAlphaBeta_Memory_MC*) renvoient toutes la même décision, seuls les temps de traitement et de réponse diffèrent (comme nous pouvons le remarquer dans la section précédente).

Conclusion

Critique de notre travail

- Nous n'avons pas réussi a concevoir des tests suffisament robuste pour nous permettre de founir une évaluation correcte de nos agents (*confère section Test plus haut*). La raison vient du fait que, lors de la conception de nos agents, nous les avons considéré comme terminés à partir du moment où ils validaient tous les tests officiels de *main_test*. Mais on s'est rendu

compte après coup que cela ne garantissait pas pour autant qu'ils soient totalement robuste.

- Les codes de nos agents peuvent sûrement être beaucoup plus optimisés. La raison de ce manque d'optimisation réside entre autre dans notre connaissance peut-être trop succintes des classes mères *abstract_player* et *abstract_game*.
- Malgré cet échec, nous sommes contents d'être parvenus à concevoir des agents avec différentes approches de programmation d'intelligences avancées, et surtout d'avoir saisi comment ces formes d'intelligence fonctionnent.

> Perspective future et ouverture

Le point incontournable est le problème issu de nos tests : si l'on veut comparer nos algorithmes, nous avons besoin de pouvoir les faire s'affronter et les évaluer. On va donc commencer par comprendre et régler ce problème.

D'autre part, nous n'avons pas exploré la dimension "flexibilité" de notre algorithme : pouvoir jouer à des jeux est intéressant, mais qu'en est-il de résoudre des équations, ou encore de jouer à des jeux basés sur la sémantique de mots ou leur orthographe ?

Enfin, nous avions pensé à réaliser une taxonomie de nos différents types d'agents, en fonction des différentes caractéristiques développés plus haut. On l'aurait fait dans un repère en 3 dimensions : l'une pour la rapidité, l'autre pour son efficacité. On aurait pu l'étendre à la troisème dimension si étudier la flexibilité était pertinente. Il aurait suffit, pour réaliser cette représentation, de récupérer les différentes valeurs successives des paramètres dans des listes et tout simplement d'en faire un *plot*.

Il aurait aussi été possible de représenter l'évolution des différentes caractéristiques en fonction des paramètres des agents (profondeur, nombre de simulation). En particulier on se demande s'il existe des valeurs de profondeur et de nombres de simulations pour lesquels les agents NegAlphaBeta_memory et UCB sont équivalents en terme de temps et ou d'efficacité.

Une telle représentation et analyse taxonomique des différentes méthodes aurait permis de mieux appréhender les atouts de chacunes d'entre elles. Malheureusement cet objectif n'a pas pu être réalisé, ce qui est dommage car il semblait idéale pour appréhender la comparaison entre les algorithmes.

Annexe

Pour évaluer la rapidité des algorithmes, nous avons placé des marqueurs temporel dans la fonction *manche* de *main_parties*, à chaque fois que l'un des deux joueurs doit prendre une décision. Ceci permet d'évaluer en fin de partie combien de temps chaque joueur met à prendre

une décision (le temps total étant pondéré par le nombre de tours du joueur afin d'obtenir un temps moyen).

D'autre part, nous avions modélisé une méthode *rapidité* et une méthode *fight*, permettant respectivement de calculer les statistiques de temps des joueurs dans une partie (en complément de celle fournie dans *main_parties*), et de générer X nombre de parties entre deux joueurs. La dernière méthide était forte utile pour les tests.