Projet Informatique : Jeu de Coinche

Benjamin DONNOT

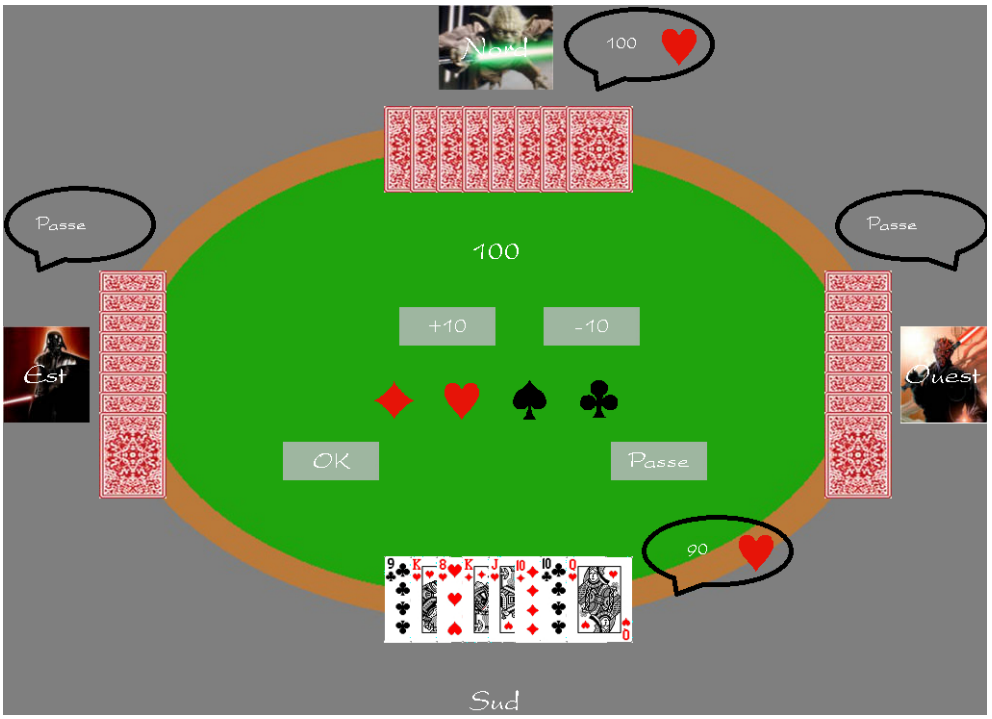

Table des matières

Introduction

Depuis le cours de python de première année, j'ai découvert un champ assez vaste informatiquement : celui de l'*intelligence artificielle*. Ce terme assez générique a plusieurs sens. Ce que j'entends par "intelligence artificielle" dans tout ce rapport, et plus généralement dans le projet tout entier, c'est la capacité pour un ordinateur à prendre des décisions "cohérentes" et ainsi à jouer "convenablement" à la coinche.

La "coinche", comme de nombreux jeu de cartes ne se joue pas en information parfaite : les cartes des adversaires sont cachées ! Résoudre [1] un jeu de coinche pourrait donc permettre de faire des avancées dans certains types de problèmes rencontrés dans la vie de tous les jours qui ne se limitent pas simplement aux jeux de cartes. Chaque jour, nous sommes amenés à prendre des décisions dans un environnement incertains [2].
Ainsi, il n'est pas rare que la recherche en intelligence artificielle se concentre sur des jeux. C'était notamment le cas au début des années 90 avec IBM et "Deep Blue", le premier ordinateur à avoir battu un grand maître aux échecs [3]. Encore aujourd'hui, de nombreuses compétitions de jeux se voient affronter les meilleurs programmes informatiques du monde, notamment au Go [4], ou encore le Bridge [5].
D'un point de vue fondamental, se concentrer sur des jeux présente plusieurs intérêts, notamment le fait que les données soient faciles à acquérir (il suffit de faire jouer un ordinateur contre lui-même un grand nombre de fois pour avoir des parties simulées), mais également parce que ces jeux sont souvent des versions (très) simplifiées de problèmes réels.
Ce type de jeu de cartes est également en relation avec la filière statistique et apprentissage, dans laquelle j'effectue ma 3e année à l'ENSAE. L'"Intelligence Artificielle" est le "Machine Learning" sont des champs très proches, notamment lorsque l'environnement n'est pas entièrement connu par les acteurs. Dans ce cas, il faut souvent faire des statistiques afin de prendre la meilleure décision possible. Il n'est donc pas surprenant que les meilleurs algorithmes de Bridge actuels reposent sur des techniques "Monte Carlo" : on va simuler un grand nombre de jeux possibles, tenter de résoudre chaque jeu en information complète, puis agréger tous ces résultats pour en tirer la meilleure action à faire dans l'environnement incertains. On pourra se reporter à la partie **??** page 8 ci après pour plus d'informations.

En 2e année, j'ai pu suivre le cours de C++, et j'ai réalisé un projet de Belote [6]. Pour la réalisation de ce projet, je disposais donc d'une interface graphique correcte [7]. Cette interface est primordiale dans des jeux de cartes et est très longues à coder. Ne trouvant que peu d'intérêt dans la création d'une seconde interface graphique au cours de ma scolarité, j'ai donc décidé de faire ce projet sur un jeu de Coinche, jeu très proche de la Belote, au moins dans les règles.
Au cours de ma deuxième année, je m'étais concentré sur deux aspects principaux : l'interface gra-

1. Entendre ici : performer aussi bien que les humains en général.
2. Qui est plus nettement plus compliqué que l'environnement simplifié du jeu de Coinche.
3. Cette victoire a suscité de nombreuses polémiques qui ne seront pas abordées ici, car ce n'est pas le sujet. Toujours est-il qu'aujourd'hui, il est communément admis que les ordinateurs jouent mieux que les humains aux échecs.
4. On pourra se reporter au "Computer Go" : `http://en.wikipedia.org/wiki/Computer_Go`.
5. Jeu relativement proche de la Coinche, mais beaucoup plus populaire outre Manche et outre Atlantique. On pourra se reporter au "Computer Bridge" : `http://en.wikipedia.org/wiki/Computer_bridge`.
6. Les sources de l'époque sont disponibles dans le répertoire SVN suivant : `https://subversion.assembla.com/svn/belote_cpp/`.
7. Celle-ci, même si je l'ai recodée en grande partie n'est pas exempte de défauts, mais ne fait pas l'objet de ce rapport, ni même de ce projet.

phique, ainsi que la prise. La partie jeu à proprement parlé n'avait été que peu étudiée, et faisait l'objet d'une implémentation rudimentaire. Ce projet aura donc pour but de réaliser une intelligence artificielle capable de jouer de façon "correcte" sur toute une partie de Coinche, en utilisant des techniques Monte Carlo décrites précédemment.

I Préambule

1 Architecture

Ce code est relativement volumineux, il contient plus de 7 000 lignes de codes [8]. Il compte 73 fichier de déclaration (.h) et 45 fichier source (.cpp) et environ 90 définitions de classes. Le détail de chaque fichier, ou même de chaque classe est donc beaucoup trop long pour être présenté en exhaustivité dans ce rapport.
En revanche, chaque "header" est construit de la même façon. Tout d'abord, un courte description des classes qu'il contient est présentée, ensuite d'autre "headers" sont inclus, puis enfin des "TO DO" sont présents. Encore une fois, comme ce projet s'inscrit pour moi dans une vision à plus long terme que la validation de ce cours, ceux-ci sont présents parce que beaucoup d'aspects restent à améliorer, et il est donc important de garder quelques parts les améliorations auxquelles je pense pour améliorer ce code.
Cette partie sera uniquement dédiée à l'architecture du code en général. On pourra se reporter au header définissant une classe si l'utilité de celle-ci n'est pas claire.

On rentre dans le programme via le fichier "main.cpp". Celui-ci va déclarer une instance de la classe "Game_Coinche". Cette classe a pour but d'orchester le jeu. Elle ne va rien calculer, mais va faire en sorte que les différentes phases du jeu se déroulent sans accros, c'est à dire que toutes les phases du jeu se déroulent sans accro les une après les autres.

Ce sont d'autres classes qui sont chargées de gérer quel joueur fait quelle action à quel moment, il s'agit des classes "*Cards_Deck*" (qui va donner les cartes), "*Bidding*" (qui va organiser la phase des enchères) et "*Trick*" qui assure le bon déroulement des 8 plis. Globalement, ces classes vont donner des instructions à la classes "*Player*" qui est chargée de faire les actions à proprement parlé.

Pour l'instant, la classe "*Player*" se divise en deux : "*Player_AI*" et "*Player_Human*" qui gère respectivement les joueurs "Artificiel" et les joueurs humains. Dans l'architecture actuelle, la classe joueur va faire descendre les informations requises à la prise de décision à des "wrappers" spécialisés ("*AIPlayMonteCarlo*", "*AIPlayScores*", ou "*Player_Bid_Graphic*" par exemple). Dans la version précédente (2A), les différentes façon de jouer étaient gérées via un polymorphisme dynamique (fonction virtuelle) : ceci nécessitait de créer une classe par type de joueur IA et une pour les joueurs humains. Aujourd'hui, l'utilisation de template et de *wrapper* permettrait de s'en passer et de n'avoir qu'une seule classe "*Player*" : ce serait au moment de la compilation (et non de l'exécution) que le choix du *wrapper* serait décidé. Il n'y aurait plus d'héritages qui concernerait les joueurs.

Afin de prendre les bonnes décisions, les joueurs doivent retenir ce qui s'est passé, ceci est géré par les classes dénommées "*Memory*". Celle-ci est principalement gérée par trois classes "en cascade" : qui hérite de la précédente. Une fois encore il serait probablement une bonne idée d'utiliser des templates plutôt que d'avoir recourt à un polymorphisme dynamique.
La première, "*AIGameMemory*", est une mémoire "basique". Elle va retenir quelle carte sont tombées, si des joueurs ont encore une couleur ou ce genre d'informations primaires.
La deuxième "*AIGameMemoryImproved*" hérite de la première. Elle va faire des déductions plus fines, notamment en ce qui concerne les atouts. Il serait bien que cette classe gère aussi les conséquences

8. Sans compter les commentaires, ni les lignes avec un seul caractère, comme '{' par exemple. Plus de 10 000 sont présentes au total.

directes de ce qu'elle observe : par exemple si un joueur a encore 3 cartes, et qu'il ne peut recevoir que le roi de trèfle, l'as de coeur et le sept de pique, on pourrait facilement déduire qu'il a les trois cartes précédentes (cette partie sera faite) et ainsi que les autres joueurs ne peuvent pas les avoir (ceci n'est pas fait ici, mais dans la classe "*AIPlayMonteCarlo*", ce qui n'est pas forcément adapté : seuls les joueurs jouant avec cette technique déduiront ces conséquences).
Enfin, la dernière classe concernées, héritant de la précédente est "*AIMemPerfectInfo*". Elle est utile pour la classe "*AIPlayMonteCarlo*", lorsque les jeux ont été donnés. Le jeu sera alors en information parfaite (cf. partie III page 8 pour plus de détails).

2 Limitations

En plus des différents "TO DO" dans le code, le jeu est encore loin d'être parfait. Certaines limitations impactant directement le jeu n'ont pas encore été implémentées. On peu notamment citer :

- Les variantes de jeu 'tout atout' et 'sans atout' n'ont pas été prises en compte. Ces variantes modifient profondément les règles du jeu, et j'ai préféré me concentrer sur la façon de jouer plutôt que d'adapter le code historique (de 2A) de façon à ce qu'il puisse supporter ces variantes.
- La belote (fait d'avoir la dame *et* le roi dans la couleur d'atout) n'est également pas prise en compte. Il aurait fallu pour ça adapter l'interface graphique, je n'ai pas jugé cette tâche prioritaire.
- La coinche est également absente. Lors de la phase des enchères, on peut "coincher" pour dire à l'adversaire qu'il ne fera pas le contrat qu'il a annoncé. Ceci modifie entre autre les règles de comptage des points, et n'a pas été implémenté car j'ai laissé de côté la phase des enchères pour me concentrer sur le phase de jeu.
- Pour les mêmes raison, la prise a été mise de côté. Les joueurs IA prennent selon des critères très rudimentaires.

3 Interface graphique

multi-threading :-)

upgrade to SDL 2.0, voire SFML :-)

II *Intelligence Artificielle* "basique"

1 Aléatoirement

On joue une carte aléatoire parmi celles qu'on a le droit de jouer :-)

benchmark : référence de jeu.

2 Avec des scores

score de chaque carte jouable en fonction de conditions, on joue la carte avec le plus gros score.

prise en compte des appels, via la classe "Memory" (template)

III *Intelligence Artificielle* par évaluation Monte Carlo

1 Donner les cartes

problème : respect de l'aléa final : il faut une certaine uniformité dans quand on donne les cartes (cf. pistes d'améliorations)

autre difficulté : respecter les contraintes stockées dans la mémoire.

2 Jouer les jeux

3 Piste d'améliorations

Importance Sampling : regarder en détail les actions de chaque pour 'sampler' de façon plus convenable. Ceci permettrait de tirer parti d'informations telles que 'si un joueur a joué ça, c'est qu'il probablement avoir ça' [cas des appels]. Alors que la on ne tire parti que des infos du genre "un joueur a joué ça, il ne peut pas avoir ça".

Prime à la découverte / communication avec le partenaire : pour l'instant l'IA ce sont deux autistes qui prennent des décisions. On pourrait imaginer mettre en place une réelle communication par les cartes (appels).

optimisation / profiling : trouver un outil pour rendre l'évaluation plus rapide ce qui permettrait de faire plus d'évaluations, donc d'être plus performant :-)

4 Performance

Méthodologie : l'IA qui s'affronte elle même.
Deux équipes IA de types différents.
cas 1 : random vs score cas 2 : Monte Carlo random vs score cas 3 : Monte Carlo score vs Monte Carlo random

Conclusion