Projet Informatique : Jeu de Coinche

Benjamin DONNOT

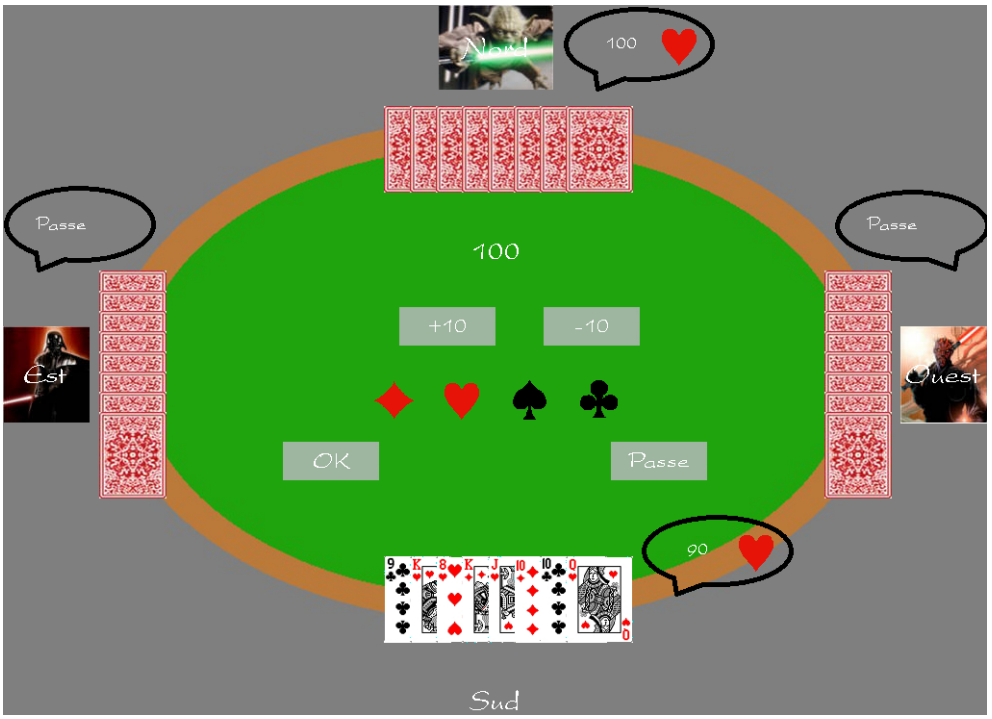

Table des matières

Introduction

Depuis le cours de Python de première année, j'ai découvert un champ assez vaste de l'informatique : celui de l'*intelligence artificielle* (IA). 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 jeux 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 où nous sommes amenés à prendre des décisions dans un environnement incertain [2]. Ainsi, il n'est pas rare que la recherche en IA se concentre sur des jeux. C'était notamment le cas au début des années 90 avec IBM et "Deep Blue", le premier ordinateur a avoir battu un grand maître aux échecs [3]. Actuellement, de nombreuses compétitions de jeux 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 : les données sont faciles à acquérir (il suffit de faire jouer un ordinateur contre lui-même un grand nombre de fois pour avoir des parties simulées), et ils représentent souvent des versions (très) simplifiées de problèmes réels.

Ce type de jeu de cartes est également en relation avec le module *statistique et apprentissage*, dans lequel j'effectue ma 3e année à l'ENSAE. L'"Intelligence Artificielle" et 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 incertain de départ. On pourra se reporter à la partie III page 10 ci après pour plus d'informations.
J'ai également tenu à coder un jeu de Coinche car c'est un jeu auquel j'aime jouer pendant mes loisirs en tant qu'amateur.

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]. C'est aussi parce que cette interface était convenable que j'ai porté mon dévolu sur ce jeu de Coinche. De cette façon j'ai pu me concentrer sur ce qui, pour moi, est l'essentiel : l'***IA***.
Au cours de ma deuxième année, je m'étais concentré sur deux aspects principaux : l'interface graphique, ainsi que la prise. La partie "jeu" en elle-même n'avait été que peu étudiée, et faisait l'objet

1. Entendre ici : performer aussi bien que les humains en général.
2. Qui est 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 pendant ma césure, n'est pas exempte de défauts, mais ne fait pas l'objet de ce rapport, ni même de ce projet.

d'une implémentation rudimentaire. Ce projet aura donc pour but de réaliser une intelligence artificielle capable de jouer correctement sur toute une partie de Coinche, en utilisant des techniques Monte Carlo décrites précédemment.

Ce projet a été principalement codé grâce à codeblocks dans un environnement Linux (Ubuntu 14.04 LTS), le compilateur principal est gcc 4.8.2.
Ce projet dépend également de boost (pour la partie génération de nombres aléatoires : cette dépendance sera sûrement enlevée dans un futur proche) et utilise le standard c++11.
Parce que je voulais que mon jeu soit également disponible à des personnes utilisant windows, je l'ai également compilé avec codeblocks sous windows (avec le compilateur MingW). Comme ce compilateur ne me satisfaisait plus, j'ai essayé de faire en sorte que ce projet soit compilable avec visual studio (Visual C++). L'étape de compilation fonctionne, mais il reste quelques bugs dans l'application compilée. Je n'ai pas toujours eu le temps de m'y attardé, d'autant que ces bugs n'apparaissaient pas avec GCC...

Le fonctionnement est donc garanti avec :

— gcc 4.8.2
— codeblocks 13.12
— boost 1.55
— SDL 1.2 (version de développement)

La compilation avec Visual Studio reste "expérimentale".

I Préambule

I.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) pour 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, et présenterait peu d'intérêt.
En revanche, chaque "header" est construit de la même façon. Tout d'abord, une courte description des classes qu'il contient est présentée, ensuite d'autres "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. Ces "TO DO" sont présents parce que beaucoup d'aspects restent à améliorer, et il est donc important pour moi de garder quelques parts les améliorations auxquelles je pense.
Cette partie sera uniquement dédiée à l'architecture du code en général. On pourra se reporter au header définissant une classe si l'utilité ou le fonctionnement 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 se déroulent sans accros les unes après les autres. Un point est a soulevé ici. Pour ce projet, je n'ai pas fait de tests unitaires et les erreurs ne sont pas gérées. Une première piste d'amélioration est donc clairement la gestion des exceptions, et la création de tests unitaires, notamment parce que le projet grossit.

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 parler.

Pour l'instant, la classe "*Player*" se divise en deux : "*Player_AI*" et "*Player_Human*" qui gère respectivement les joueurs "artificiels" (IA) et les joueurs humains. Dans l'architecture actuelle, la classe joueur va faire descendre les informations nécessaires à la prise de décision à des "wrappers" spécialisés ("*AIPlayMonteCarlo*", "*AIPlayScores*" ou "*Player_Bid_Graphic*" par exemple). Dans la version précédente (projet Belote de 2A), les différentes façon de jouer étaient gérées via un polymorphisme dynamique (fonction "play" virtuelle) : ceci nécessitait de créer une classe par type de joueur IA et une pour les joueurs humains. Aujourd'hui, l'utilisation de "templates" et de *wrappers* 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".

I.2 Limitations principales

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

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

— Les variantes de jeu 'tout atout' et 'sans atout' n'ont pas été prises en compte. Ces variantes modifient profondément les règles du jeu [9], 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 cela 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 signifier 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 raisons, la prise a été mise de côté. Les joueurs IA prennent selon des critères très rudimentaires.
— Il n'y a pas de prise en charge d'un "protocole réseau". On ne peut jouer que seul contre l'ordinateur. Deux joueurs humains ne peuvent pas jouer une même partie en même temps.

I.3 Interface graphique

Une des raisons qui m'ont poussées à choisir ce sujet était le fait que j'avais déjà codé une interface graphique pendant que j'étais en 2A.
La librairie utilisée est *SDL 1.2*, une librairie codée en C. Durant ce projet, j'ai été plusieurs fois confronté à des problèmes d'utilisation, parce que les fonctions que j'avais codées n'étaient pas simple à utiliser et assez instable. Pour l'interface graphique, j'avais eu recourt à une architecture particulière, basée sur l'héritage multiple, qui rend les classes que j'ai créées assez "obscures", même pour moi qui les ais codées.
Ce pourrait donc être une bonne idée de recoder cette partie. Mais, comme une nouvelle version de cette librairie est maintenant disponible (la version 2.0), je pourrai également en profiter pour mettre à jour mon code. Je pense à également à re-coder l'interface pour utiliser *SFML* une autre librairie graphique qui a le mérite d'être codée en C++.

Dans cette sous-partie, je voulais également mentionner que, par rapport à l'interface originale, j'ai ajouté une partie "Multi-Threading" [10]. Lorsqu'un joueur IA prend une décision pour jouer une carte, l'interface graphique ne se fige pas, contrairement à ce qui se serait passer si l'application n'utilisait qu'un seul "thread".

I.4 Débugage

Une telle application a connu pas mal de bugs. Aujourd'hui, je pense en avoir éliminé une grande partie. Ceci n'a pas été facile, d'autant que les débugueurs ralentissent grandement la vitesse d'exécution des programmes, et que certains bugs apparaissaient au bout de plusieurs minutes d'exécution "standard" [11]. Pour arriver au même endroit avec un débugueur, il aurait sans doute fallu attendre plusieurs dizaines de minutes, ce qui est très frustrant.

9. Avec la disparition de l'atout...
10. En utilisant les header de la librairie standard "thread" et "future". Ceci implique entre autre que le projet ne puisse plus être compilé avec Codeblocks sous windows (qui utilise MingW), parce que le compilateur en question ne prend pas en compte ces librairies.
11. Comprendre ici : sans que le debugueur ne soit activé.

J'ai donc beaucoup pratiqué le débugueur via l'affichage à l'écran. Pour que le code que j'ai créé soit relativement générique, j'ai donc codé deux wrappers, codés dans le fichier "*DebugWithPrint*". Il s'agit de la classe "*WrapperPrint*". Cette classe est templaté par un entier. En général elle ne fait rien, mais une spécialisation de cette classe a lieu lorsque cet entier vaut 1 et dans ce cas elle va afficher (via "vfprintf") du texte dans la console.
Cette petite astuce m'a permis de localiser des bugs de façon assez rapide. Elle permet également de ne pas ralentir l'exécution du programme (cela prend du temps d'afficher beaucoup de texte) une fois que le bug a été trouvé, tout en ne déclarant que quelques variables.

II *Intelligence Artificielle* : premier pas

Les deux prochaines parties vont rentrer plus dans les détails de l'intelligence artificielle lors de la phase de jeu. Il serait sans doute plus facile de coder une intelligence artificielle qui est en information parfaite. Mais ceci ferait perdre beaucoup d'intérêt à ce projet. Donc, les 3 joueurs gérés par l'ordinateur ne connaissent pas le jeu des autres, et n'ont donc accès qu'au leur.

Cette partie va se concentrer sur l'architecture du code concernant l'IA, ainsi que sur les IA "basiques" qui sont aujourd'hui implémentées.

II.1 Architecture

Les décisions que l'ordinateur doit prendre concernant le jeu sont prises via la méthode "what_card_do_i_play". Dans cette méthode, il s'agit de choisir une parmi une liste de "cartes jouables" [12]. Aujourd'hui, pour le joueur IA, cette méthode se contente (dans le cas où l'application n'est pas multi-threadée) de rediriger la sortie du wrapper "PlayAI" (template).

À ce jour, ce wrapper peut être de 3 natures différentes :
— "*AIPlayRandom*" : va permettre au joueur de jouer une carte aléatoire. On parlera dans ce cas d'**IA aléatoire**.
— "*AIPlayScores*" : va permettre au joueur de jouer une carte selon des scores prédéfinis. On parlera plus dans ce cas d'**IA scores**.
— "*AIPlayMonteCarlo*" : va jouer avec la méthode "Monte Carlo". On pourra se reporter à la partie III pour plus d'informations. On appellera ce type de jeu "**IA Monte Carlo**".

Cette architecture permet rapidement de créer de nouveaux types d'IA (avant, il aurait fallu recréer une classe héritant de "*Player_AI*"), mais permet également de faire interagir des types d'IA entre eux. Par exemple, il y a deux sous-façons de jouer pour une "IA Mont Carlo" : la première est de simuler des parties où les joueurs jouent aléatoirement (**IA Monte Carlo aléatoire**), la seconde d'en simuler lorsque les joueurs jouent avec des scores (**IA Monte Carlo scores**). Ce schéma aurait été compliqué à réaliser sans recours aux templates : il aurait fallu utiliser l'héritage multiple (ou pire, le copier coller).

II.2 Aléatoirement

La classe qui s'occupe de ce type de jeu est la classe "*AIPlayRandom*". L'ordinateur va choisir aléatoirement une carte parmi celle qu'il peut jouer.

Cette "intelligence" artificielle est la première que j'aie implémentée. C'est la plus simple et va pouvoir servir de référence pour comparer l'efficacité des autres.

II.3 Avec des scores

Ce type de jeu est géré par la classe "*AIPlayRandom*". L'ordinateur va calculer le score de chacune des cartes qu'il peut jouer, et jouer celle avec le plus gros score.

12. Ces cartes sont déjà connues à ce stade.

Le score de chaque carte est influencé par différentes "conditions". Ces conditions sont fixes (et ne peuvent pas être modifiées). En revanche les scores sont lus depuis le disque dur grâce à la classe "*Datas*". Cette classe pourrait également réécrire de nouveaux codes qui seraient calculés. On pourrait donc établir précisément une méthode de calibration. Ceci n'a toujours pas été fait, mais est envisagé dans un futur proche.
Dans le même ordre d'idée, il pourrait également être envisagé d'imiter le comportement du joueur humain grâce à une technique d'apprentissage supervisé. Ceci pourrait constituer une première façon de calibrer les scores, peut-être moins gourmande en ressources que l'apprentissage par renforcement. La performance de cette classe est directement influencée par l'humain ayant codé les conditions et les scores associés.

Cette classe présente également des avantages. En effet, elle interagit de façon claire avec la mémoire du joueur, et permet donc une communication rudimentaire entre les joueurs, avec la gestion des "appels" [13].

13. Lors d'une partie de Belote (ou de Coinche) il est possible de faire passer des informations à son partenaire en jouant certains types de cartes dans certaines conditions. C'est ce que j'appelle "appel".

III *Intelligence Artificielle* par évaluation Monte Carlo

La deuxième grande méthode dont nous allons parler utilise des techniques qu'on pourrait qualifier de "Monte Carlo". Cette méthode est implémentée via la classe "*AIPlayMonteCarlo*" et repose sur la simulation de nombreux "mini-jeux", qui sont ensuite joués, leurs résultats agrégés et la meilleure action est faite en fonction de l'agrégation précédente. Cette méthode repose donc sur trois principes, chacun fera l'objet d'une sous-partie dédiée :

1. Donner les cartes encore en jeu, en accord avec les actions qui ont été faites précédemment dans les tours précédents par tous les joueurs.
2. Jouer des "mini-jeux" en respectant les règles bien entendu.
3. Agréger les résultats puis sélectionner la meilleure carte parmi les cartes jouables.

Ces mini-jeux sont joués par des "*PlayerMiniMonteCarlo*", et la méthode qui permet de jouer ces jeux fait en fait appel à la classe "*MCPlayGames*", qui est templaté (entre autre) par la façon dont les mini-joueurs vont jouer.

III.1 La mémoire

La mémoire des actions faites par les joueurs est ici fondamentale. Plus elle sera élaborée, moins des jeux "impossibles" seront distribués, jeux qui viendront biaiser les résultats.

Ainsi, un travail assez important a été fait pour être sûr que le joueur IA retienne ce qui s'est passé et en déduise des choses pertinentes.

Afin de prendre les bonnes décisions, les joueurs doivent retenir ce qui s'est passé, ceci est géré par les classes dénommées "*Memory*". Celles-ci sont principalement gérées par trois classes "en cascade" (chacune hérite de la précédente). Une fois encore, il serait probablement plus efficace d'utiliser des templates plutôt que d'avoir recours à un polymorphisme dynamique.
La première, "*AIGameMemory*", est une mémoire "basique". Elle va retenir les cartes tombées, si des joueurs ont encore des cartes dans telle ou telle couleur ou ce genre d'informations primordiales.
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 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 cœur 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 concernée 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 10 pour plus de détails).

Pour l'heure, j'envisage deux pistes d'amélioration principales pour la mémoire :
— Comme je l'ai mentionné précédemment, celle-ci pourrait également prendre en compte les conséquences directes de ce qu'elle vient d'observer. Pour l'instant cette tâche est confiée à la méthode "*computeConsequences*" de la classe "*AIPlayMonteCarlo*". Il serait sans doute bien de déplacer cette fonction dans la mémoire directement.

— Une des améliorations majeures pourrait être la prise en compte d'information "probables", et non uniquement certaines. Ainsi, la mémoire pourrait déduire des choses comme : "un joueur a joué cette carte, il doit avoir cette autre carte, car il ne serait pas logique qu'il en soit autrement". On pourrait ainsi associer des degrés de certitude. Ceci pourrait sans doute permettre de simuler des jeux plus proches de la réalité lors de la phase "donner les cartes" cf. sous partie III.2 ci-après, en utilisant des techniques "d'importance sampling" ou de MCMC par exemple.

III.2 Donner les cartes

La première étape de ce type de jeux est donc de générer un "grand" nombre de jeux différents, ce qui présente deux principales difficultés :

— il faut garantir que les cartes données à un joueur aient bien le droit de l'être, en accord avec les informations stockées dans la mémoire du joueur qui joue.

— il faut respecter un certain aléa. Cette méthode, comme toutes les méthodes Monte Carlo, repose sur le fait que l'"aléatoire" généré est "fidèle" à la réalité.

Pour surmonter ces difficultés, j'ai mis en œuvre l'algorithme suivant :

Étape 1 : Donner les cartes contraintes [14]

cas a : si le jeu est soluble [15] on continue à l'"**Étape 2**".

cas b : on se souvient que le dernier joueur à qui on a donné une carte ne pouvait en fait pas la recevoir, on défait ce qui a été fait [16], et on recommence à l'"**Étape 1**".

Étape 2 : Choisir une carte aléatoirement (actuellement uniformément) parmi celle qui reste, donne la à un joueur aléatoirement (toujours uniformément) parmi les joueurs qui peuvent la recevoir et on recommence à l'"**Étape 1**".

Informatiquement, ceci a été fait itérativement grâce à une "stack" [17](pile de priorité) qui retient toutes les actions faites à chaque étape, et qu'on va "dépiler" (pop) lorsque le jeu donné s'avère en fait impossible. Les actions entreprises pourront dont être annulées.

Cet algorithme va donc garantir que les informations contenues dans la mémoire seront respectées, mais également que les jeux simulés seront au moins en partie "aléatoire".

III.3 Jouer les jeux

Une fois que les jeux ont été donnés, on peut alors faire jouer ces "mini joueurs" en supposant qu'ils sont en information parfaite, c'est à dire que chacun connaît le jeu de tout le monde. C'est ce qui est fait grâce à la classe "*MCPlayGames*".

14. Il y a deux types de cartes contraintes : les cartes qui ne peuvent être données qu'à un seul joueur ou les joueurs qui peuvent recevoir exactement autant de carte qu'on peut leur donner.

15. On l'entendra ici par le faite que toutes les contraintes peuvent être satisfaites sans en violer d'autres.

16. C'est à dire qu'on reprend les cartes qui avaient été données et qu'on oublie que l'avant dernier joueur à qui on avait donné une carte ne peut pas la recevoir, puisque dans ce cas le problème était déjà présent plus en amont.

17. Aussi appelée "lifo", pour "last in first out" (dernier à rentrer, premier à sortir).

Encore une fois, celle-ci peut être templatée par la méthode à utiliser pour la mémoire [18], mais aussi pour la méthode de jeu employée par les "mini-joueurs".

Le code a été fait de telle façon que toutes les méthodes de jeu qui peuvent servir à jouer "normalement" (c'est à dire sans Monte Carlo) peuvent être utilisées ici. Ceci est encore rendu possible par l'utilisation de templates.

La classe "*MCPlayGames*" permet également de jouer plusieurs fois le même jeu. Cette fonctionnalité est inutile pour des types de jeux déterministes ("*AIPlayScores*" par exemple), mais peut présenter un intérêt dans des méthodes "stochastiques" : lorsque le jeu est joué de façon purement aléatoire par exemple.

III.4 Piste d'améliorations

Cette technique pourrait être améliorée de plusieurs façons.

On pourrait regarder plus en détail ce qu'ont fait les joueurs avant nous, de façon à tirer des jeux "plus probables". Ceci permettrait de tirer parti d'informations telles que "si un joueur a joué ça, c'est qu'il doit probablement avoir ça". Ceci permettrait de tirer profit des appels passés par le partenaire du joueur qui joue.

La communication entre les deux partenaires est également totalement laissée de côté dans l'évaluation du jeu. Jouer avec cette technique va tenter d'optimiser son score en considérant qu'on ne passe aucune information à son partenaire. On pourrait imaginer mettre en place une réelle communication par les cartes (appels), au niveau de la simulation de la partie. Ainsi on optimiserait réellement le score de l'équipe à la fin de la partie, et plus uniquement le score en jouant son propre jeu. Ceci passerait donc par faire de nouveaux appels à son partenaire.

Enfin, cette technique est assez gourmande en ressources. On pourrait donc tenter d'utiliser des méthodes de "profiling" pour se rendre compte des véritables points faibles du programme et ainsi les corriger, augmentant ainsi le nombre de tirages faits.

III.5 Performance

Créer des IA est satisfaisant, mais vérifier qu'elles fonctionnent est encore mieux. Comme je n'ai pas eu assez de retours de "vraies" personnes qui ont joué à ce jeu, j'ai donc décidé de confronter les IA décrites au dessus en organisant 3 matchs différents.
À chaque fois, 3 000 parties ont été jouées. Deux joueurs coéquipiers étaient d'un certain type d'IA et affrontaient les deux autres joueurs qui étaient donc d'un type différent.
Les scores retenus ont été les points marqués dans le match, sans tenir compte du contrat. Les 3 000 parties jouées l'ont été vraiment, dans le sens ou si aucune des des équipes ne prenait, la partie ne

18. Même si en théorie chacun connaît toutes les cartes des autres, il n'est pas impossible de donner une mémoire globale au jeu. Attention, les "mini joueurs" n'ont pas de mémoire. La mémoire dont il est question est bien la mémoire du "planificateur" omniscient, omnipotent et bienveillant qu'on peut également rencontrer dans différentes théories économiques. Ce planificateur peut ou non décider de partager son savoir avec les mini-joueurs.

comptait pas.

Les match ont vu s'affronter :

Match 1 : scores (IA scores) contre purement aléatoire (IA aléatoire) (sans Monte Carlo dans aucun des cas)

Match 2 : Monte Carlo avec "mini-jeux" aléatoires (IA Monte Carlo aléatoire) contre scores (sans Monte Carlo)

Match 3 : Monte Carlo avec "mini-jeux" suivants les scores (IA Monte Carlo scores) contre Monte Carlo avec "mini-jeux" aléatoires

Dans le cas IA Monte Carlo aléatoire : 100 jeux ont été simulés, et ces jeux étaient joués 30 fois. Á chaque fois qu'un joueur devait choisir une carte, 3 000 mini-jeux différents étaient donc envisagés. Ce 3 000 n'a rien à voir avec les 3 000 parties simulées.
Dans le cas IA Monte Carlo scores : 3 000 jeux ont été simulés, chaque jeu n'étant bien entendu joué qu'une seule fois, puisque la méthode en question est déterministe.
La classe de référence pour chacun des matchs est toujours la classe citée en second dans la description précédente. Pour le Match 1, on va donc comparer le score de "IA scores" contre "IA aléatoire".

Les résultats sont présentés dans le tableau suivant :

	Victoires	Victoires (%)	Points	Points en plus(%)
Match 1	1 651	55	256 721	12
Match 2	1 811	60	281 019	37
Match 3	1 565	52	247 674	4

TABLE 1 – Résultats des 3 matchs précédemment décrits.

Comme nous pouvons le voir dans le tableau 1 ci-dessus, à chaque fois le challenger est meilleur que la référence. Ainsi, conformément à l'intuition : l'IA score sera meilleur que l'IA aléatoire (si les deux sont de même type : Monte Carlo ou "basique") et le joueur Monte Carlo aléatoire bat le joueur score.
Ce tableau illustre également qu'il y a encore beaucoup de travail de calibration des scores. En effet, l'IA score ne bat l'IA aléatoire que dans 55% des cas, ce qui est bien peu. De plus, les victoires semblent d'ailleurs généralement serrées, puisque le joueur score n'a en moyenne que 12% de points en plus que l'autre.
La meilleure amélioration apportée à l'IA a été le passage à une évaluation Monte Carlo. En effet, par rapport au meilleur joueur précédent (IA score), le Monte Carlo aléatoire marque en moyenne 37% de points en plus. Il gagne dans 60% des cas !
Enfin, le meilleur joueur créé est bien le joueur Monte Carlo scores, mais de peu, puisqu'il ne gagne en moyenne que 4% de points en plus par partie par rapport au joueur Monte Carlo aléatoire, ce qu'il fait qu'il gagne seulement 52% des parties qu'il joue [19].

Ce mémoire ne serait sans doute pas complet sans un petit mot sur le temps pris par l'IA pour jouer dans ces conditions. Dans le match le plus intensif, sur un processeur cadencé à 1.8Ghz, il a fallu environ 10h pour jouer les 3 000 parties. Globalement, l'ordinateur est donc capable de jouer

19. Pour être décisive, cette confrontation demanderait d'être évaluée sur plus de parties. Je ne suis pas certain que 3 000 parties suffisent à les discriminer.

300 parties par heure, il lui faut donc environ 12 secondes pour jouer une partie complète. Un joueur IA avec les caractéristiques précédentes va donc mettre en moyenne $0.4s$ pour jouer [20], ce qui est totalement compatible avec une utilisation "ludique" du jeu dans laquelle il serait impensable d'attendre une dizaine de secondes par coup.

20. Attention cependant ceci n'est qu'une moyenne, certains coups notamment les premiers, vont prendre un petit peu plus de temps, peut-être 1 ou 2 secondes.

Conclusion

En conclusion j'ai codé un jeu de Coinche qui manque encore de certaines fonctionnalités, mais pouvant servir de bases à des approfondissements, tant au niveau de l'intelligence artificielle qu'au niveau des graphismes.

La partie la plus importante a été le développement d'une méthode basée sur des évaluations Monte Carlo de "mini-jeux" pour jouer.
Cette méthode permet de simuler de nombreuses mains possibles, et de résoudre le jeu correspondant en information parfaite. Elle est complètement compatible avec les autres types d'IA qui ont été implémentés jusqu'à présent, notamment grâce à l'utilisation de templates.

Ce jeu peut donc être modifié relativement rapidement (en ce qui concerne l'intelligence artificielle), mais ses performances ne sont pas encore suffisantes pour battre un joueur humain un peu expérimenté.