Projet HearthBot

Équipe **Ping** : B. Choquet, C. Dallard,
M. Mohamed, V. Pollet, J. Thiébaut

Conducteur : R. Lehany
Encadrant : E. Bourreau

Université de Montpellier
Master mention Informatique

25 mai 2015

Résumé

Le projet *HearthBot* consistait en l'élaboration d'un automate informatique capable de prendre et appliquer des décisions dans un jeu fini à information incomplète : *Hearthstone*[1].

Pour y parvenir, il aura été nécessaire de franchir trois obstacles :

— Comment calculer un meilleur coup ?
— Comment récupérer les situations de jeu en temps réel ?
— Comment un programme peut-il, en pratique, jouer ?

Ce mémoire regroupe les solutions que notre équipe - PING - a su apporter aux problèmes sus-posés, une présentation des différents logiciels déployés comme solution, et quelques unes des nombreuses perspectives offertes par les outils développés.

Mots-clés : *Hearthstone*, IA, Exploration, *Backtracking*, Bot.

1. jeu de cartes stratégique développé par Blizzard

Table des matières

Tout un chacun est libre de s'inspirer de notre démarche. Les idées ne devraient pas être esclaves de concepts tels que la paternité.

Nous adressons nos remerciements à notre encadrant, *E. Bourreau*, dont la bienveillance et le support ont rendu possible la réalisation de ce projet.

Nous vous souhaitons une agréable lecture de ce mémoire.

– *L'équipe* PING

Chapitre 1

Préambule

1.1 Introduction à *Hearthstone*

1.1.1 Présentation générale de la licence

Hearthstone est un jeu de cartes stratégique au tour par tour développé par *Blizzard*. Sorti initialement sur PC et OS X en mars 2014, il a depuis été porté sur tablettes puis sur smartphones. Ce jeu reprend de nombreux éléments, personnages, mécaniques d'une autre de leur licence, *Warcraft* [1].

Le principal but du jeu est d'affronter d'autres joueurs du *Ladder*, c'est-à-dire un classement composé de 26 rangs distincts, le rang 25 étant le moins bon, et le rang Légende (correspondant au rang 0), celui regroupant l'élite du jeu. Les joueurs rencontrent donc d'autres joueurs de leurs niveaux, et montent ou descendent de rang suivant leurs résultats.

En un peu plus d'un an d'existence, *Blizzard* a réuni 30 millions de joueurs dans le monde [2] autour de *Hearthstone*, et en a fait un jeu compétitif, avec des tournois internationaux se déroulant régulièrement où s'affrontent les meilleurs joueurs de la planète.

1.1.2 Le déroulement d'une partie

Une partie de *Hearthstone* est un duel de cartes au tour par tour entre deux joueurs, chacun incarnant un Héros dont la quantité de ressources, les cristaux de mana, est limitée [3]. Chaque héros représente une classe [4], et chaque classe dispose d'un pouvoir et de cartes spécifiques ; la majorité des cartes du jeu reste cependant utilisable par toutes les classes.

Au début de la partie, chaque joueur pioche une main de départ. Pour compenser ce désavantage, le joueur qui joue en second bénéficie alors d'une carte en plus ainsi que de la Pièce [5]. À partir de la main qu'ils viennent de piocher, chaque joueur peut décider de défausser certaines de ses cartes pour en piocher le même nombre ; la partie commence alors.

1. Univers Médiéval-Fantastique.
2. Chiffres officiels de *Blizzard* datant du 6 mai 2015.
3. Cette ressource est nécessaire à la mise en jeu des cartes
4. Il en existe neuf différentes
5. Carte ne coûtant aucun cristal de mana, et permettant de gagner un cristal de mana le tour où elle est jouée.

Au début du tour d'un joueur, il pioche une carte de son *deck*[6]. Si celui-ci est vide, son *Héros* subit des dégâts[7]. La pioche en début de tour constitue pour le joueur le seul moyen de pioche *passif*. Pendant le tour, le joueur peut décider de faire attaquer ses serviteurs s'il en possède sur son *board*[8], de jouer une carte de sa main, ou de faire attaquer son *Héros* si celui-ci est équipé d'une arme. Chaque carte possède un coût en cristal de mana, ce coût étant proportionnel à la puissance de la carte. Les cristaux de mana sont la ressource de base d'un tour. Chaque fois qu'un joueur joue une carte, cette réserve de cristaux diminue d'une valeur équivalent au coût de la carte ; une carte ne pouvant pas être jouée si la réserve de cristaux d'un joueur n'est pas suffisante. La réserve de cristaux de mana est initialisée à un au début de la partie, réinitialisée et augmentée d'une unité au début de chaque tour, jusqu'à un maximum de dix cristaux disponibles à chaque tour.

Un joueur remporte la partie quand il réduit à zéro les points de vie de son adversaire, initialisés à trente au début du match.

1.1.3 Les cartes

Il existe trois types de cartes : les sorts, les armes et les serviteurs. Chaque carte a également une rareté[9], généralement indicatrice de la puissance de celle-ci. Une carte peut se trouver en au plus en deux exemplaires dans un *deck*, sauf pour les cartes légendaires, limitées à un exemplaire, pour un total de 30 cartes.

Les sorts

Les sorts peuvent être de plusieurs natures ; un sort peut soigner un serviteur, un héros, il peut infliger des dégâts, faire piocher, *etc...* Un sort est aussi défini par sa portée, c'est-à-dire l'ensemble de cible auquel il fera un effet. Cet ensemble peut être une cible unique (*"Rend 6 points de vie à un serviteur."*), un côté du *board*[10] (*"Inflige 4 points de dégâts aux serviteurs adverses."*), global (*"Inflige 3 points de dégâts à tous les personnages*[11]"), aléatoire (*"Tire 3 projectiles sur des adversaires aléatoires, infligeant chacun 1 point de dégâts."*), ou sans cible(*"Piochez 2 cartes"*).

Les armes

Une arme est un moyen pour un joueur de faire attaquer son *Héros* directement. Une arme est définie par son attaque – c'est-à-dire combien de dégâts va infliger le *Héros* en attaquant–, sa durabilité – combien de fois le *Héros* pourra utiliser l'arme– , et une mécanique[12]. Il est important de noter que toutes les armes sont des cartes spécifiques à certaines classes ; seules quatre classes[13] disposent de telles cartes.

6. paquet de trente cartes
7. Ces dégâts commencent à un, puis croissent de un à chaque fois que le joueur pioche.
8. Zone du jeu réservée à la pose des serviteurs. Chaque joueur peut, au plus, mettre sept créatures de son côté du plateau
9. Dans l'ordre croissant de rareté : basique, commune, rare, épique ou légendaire.
10. Côté allié ou ennemi.
11. Le terme personnage signifie *Héros* et serviteur
12. *Cri de Guerre* ou *Râle d'Agonie*
13. Paladin, Guerrier, Chasseur et Voleur.

Les serviteurs

Un serviteur, ou créature, est une carte qui se pose sur son *board*, et qui perdure tant qu'elle n'a pas été tuée. Il dispose de points d'attaque et de points de vie. Dans la suite de ce rapport, lorsqu'on parlera d'un serviteur X/Y, cela signifiera qu'il s'agit d'une créature avec X points d'attaque, et Y points de vie [14]. Pour tuer une créature, il faut faire tomber ses points de vie à zéro. Pour cela, on peut soit lui infliger des dégâts à l'aide de sorts, soit lui infliger des dégâts avec un personnage. Lorsqu'un personnage attaque un serviteur, on parle alors de *trade*, chacun perd un montant de point de vie équivalent au point d'attaque de l'autre. Une fois qu'un personnage a attaqué, il ne peut plus attaquer pendant ce tour [15].

Un serviteur doit être invoqué (on dit aussi *posé* sur le *board*) pour être joué. Un serviteur invoqué pendant un tour allié ne pourra attaquer qu'à partir qu'au prochain tour du joueur, permettant ainsi à l'adversaire de le tuer ou non pendant son tour. Un serviteur peut posséder un *Cri de Guerre* (respectivement un *Râle d'Agonie*), c'est-à-dire une mécanique (pioche, dégâts, soins...) qui va se déclencher quand le serviteur est posé sur le *board* (respectivement tué).

Les serviteurs étant les éléments centraux de *Hearthstone*, il existe de nombreuses mécaniques de jeu qui s'appliquent aux serviteurs rendant le jeu plus intéressant. En voici une liste :

Accès de rage Quand le serviteur subit des dégâts, il dispose d'un nouveau pouvoir.

Bouclier divin Qui absorbe les dégâts de la première attaque subie.

Camouflage Le serviteur ne peut ni être attaqué, ni être visé tant qu'il n'a pas infligé de dégâts.

Charge Le serviteur peut attaquer dès le tour où il est joué.

Furie des vents Le serviteur peut attaquer deux fois par tour.

Provocation Les personnages doivent attaquer les serviteurs avec *Provocation* en premier, n'affecte pas les sorts.

Remarque 1 *Il existe certaines mécaniques qui sont propres aux classes, et qui ne seront pas présentées ici.*

Certaines créatures appartiennent aussi à des familles [16]. Bien que cette caractéristique soit moins importante que celles présentées jusqu'ici, il est possible de jouer sur cette spécificité (*"Donne +2 attaque à tous les murlocs"*).

14. De la même façon, la notation $+X/+Y$ signifiera qu'on augmente de X l'attaque d'un serviteur, et de Y sa vie.

15. On distingue "attaquer" et "se faire attaquer" ; un serviteur qui s'est fait attaqué, lui, peut se faire attaquer plusieurs fois (mais par des personnages différents).

16. Voir énumération des familles en Annexe A

FIGURE 1.1 – Un exemple de carte correspondant à un serviteur

Tout comme les sorts, les mécaniques d'un serviteurs peuvent avoir une portée. Dans l'exemple de carte présenté, on voit que la portée du *Cri de Guerre* correspond aux serviteurs adjacents ; la mécanique n'ayant bien sûr aucun effet s'il n'y a pas de serviteurs à côté. Un tel type de carte nous montre qu'il faudra, en plus de tout le reste, réfléchir aux différentes positions possibles lorsqu'on joue une telle carte. Ceci nous laisse présager d'une certaine complexité en terme de réflexion.

1.1.4 Plateau de jeu

Le jeu se présente sous la forme d'un plateau. La figure ci-jointe résume les différentes informations de la partie en cours : points de vie, serviteurs, main, *etc...* Le plateau se partage en deux parties : la partie supérieure correspond à au *board* ennemi, avec ses quatre serviteurs et la partie inférieure au *board* allié. On notera qu'un *board* contient, au plus, sept serviteurs.

FIGURE 1.2 – Plateau de jeu, avec ces différentes informations

1.2 Introduction à *HearthBot*

Le projet *Hearthbot* s'est déroulé dans le cadre du Master *mention Informatique* de l'Université de Montpellier. D'une durée fixée à cinq mois, *Hearthbot* consistait en la conception d'un automate informatique capable de jouer au jeu *Hearthstone*. L'utilisation d'automates étant prohibée par les conditions d'utilisation du jeu, les solutions d'automatisation proposées n'ont pas été déployée *en pratique*. Nous avons alors préféré orienter le projet vers un *companion* apportant des conseils au joueur en temps réel. Ce *companion* est structurellement très proche d'un automate tel qu'il l'a été initialement demandé, puisqu'il suffirait d'y ajouter un programme contrôleur de souris. Dans les faits, un tel programme a bel et bien été développé, mais n'a jamais été *connecté* au reste du logiciel.

1.2.1 Un mot sur l'organisation

Bien que l'essentiel de la mise en place de l'organisation du travail soit décrit dans le rapport de conduite de projet, nous souhaitions tout de même en dire quelques mots ici.

Étant donnée l'ampleur de la tâche à accomplir pour mener à bien ce TER, nous avons décidé dès les premières réunions, et grâce notamment aux conseils avisés de notre encadrant de projet E. Bourreau, d'évoluer de manière **agile** [17]. Un des membres du groupe ayant de plus déjà travaillé en entreprise de la sorte, il nous a été possible de mettre en place efficacement cette méthode d'organisation.

Partant de ce principe , nous nous sommes efforcés le plus possible de proposer rapidement des solutions aux problèmes rencontrés afin de mettre en place des noyaux fonctionnels, auxquels nous avons apporté des améliorations successives.

Les divers outils utilisés au cours de l'ensemble du projet sont [18] :

— **GitLab**, en tant que gestionnaire de version utilisé pour le dépôt centralisé de code.

— **Google Drive**, pour le partage de documents en relation directe avec le projet, tels que des diagrammes, carnets de sprints [19], etc...

— **Hangouts**, en outil de communication directe entre membres du projet.

Dans une moindre mesure, la **Javadoc** fournie en tant qu'outil *Java* nous a permis de rendre accessibles les différentes parties de code aux autres membres du groupe, rendant le travail bien plus modulaire.

Après ce semestre de développement acharné, nous avons été ravis de voir que cette organisation de travail a grandement contribué à la réussite globale du projet.

17. Plus exactement, nous avons utilisé le *framework* d'organisation agile **Scrum**.

18. Les solutions logicielles proposées ici ne concernent que la gestion de projet, les outils utilisés indépendamment n'en font donc pas partie.

19. En agile, un sprint est une courte période employée à la réalisation d'un ensemble objectifs

1.2.2 Motivations

Comme la plupart des licences développées par *Blizzard*, *Hearthstone* est un jeu dont la prise en main est rapide, mais dont la maîtrise est difficile car elle nécessite une connaissance parfaite du jeu. Ce qui différencie en effet un joueur lambda d'un bon joueur est sa capacité d'analyser les actions de l'adversaire pour comprendre sa stratégie, afin de pouvoir la contrer efficacement.

Étant pratiquement tous avant même le début du projet des joueurs de *Hearthstone*, l'idée de développer une IA capable de comprendre le jeu et pouvoir remplacer l'humain nous a tout de suite plu.

Ajoutez à cela toutes les questions que l'on se posait sur sa faisabilité, le projet est alors fortement attractif

1.2.3 Obstacles majeurs, *a priori*

Lors du premier *brainstorming*, nous avons pu identifier les obstacles de programmation auxquels nous allions être confronté dans la phase de développement de ce projet. Le *client* souhaitait disposer d'un automate informatique jouant à *Hearthstone*, et, après un état de l'art des automates existants, nous avons pu établir une stratégie de développement pour répondre aux problématiques du projet. Notre stratégie s'est articulée autour de trois problématiques correspondant à trois obstacles de programmation majeurs :

1. Comment peut-on calculer informatiquement un meilleur coup lors d'un tour allié ?
2. Comment peut-on récupérer automatiquement les situations de jeu ?
3. Comment un programme peut-il interagir avec un jeu vidéo pour, en pratique, jouer ?

Nous étions conscients que solutionner ces problèmes représentait une tâche conséquente, et un quatrième obstacle a pu être ajouté :

4. Comment solutionner les trois premiers problèmes en seulement cinq mois de travail à temps partiel ?

Chapitre 2

Développement

« Tous les jeux, y compris ceux qui paraissent les plus simples, recèlent d'antiques sagesses. »

– B. Werber

2.1 Stratégie adoptée

Le tableau suivant récapitule les problématiques de développement précédemment évoquées, les solutions envisagées, et les solutions préférées.

Problématique	Solutions envisagées	Solution préférée
Calcul de meilleur coup	Exploration & évaluation exhaustive des coups possibles	**Exploration & évaluation**
Récupération des informations de jeu	Écoute des boîtes réseaux Reconnaissance visuelle Lecture des journaux de débogage	**Lecture des journaux de débogage**
Automatisation	Injection de paquets Contrôle de pointeur	**Contrôle de pointeur**

FIGURE 2.1 – Problèmes - Solutions

Ces axes pouvant être développés indépendamment nous nous sommes efforcés de suivre un patron de conception MVC [1] pour séparer efficacement les différentes composantes du programme. Ainsi pouvions nous développer parallèlement les solutions choisies aux trois problématiques et une fois certains paliers atteints, faire communiquer les composantes..

1. Modèle-Vue-Contrôleur

2.2 Simulation de *Hearthstone*

Pour dépasser le premier obstacle, à savoir concevoir une façon d'évaluer la pertinence d'un coup, nous avons opté pour une exploration et évaluation de l'ensemble des coups possibles étant donnée une situation de jeu. En effet, les automates existants embarquent des intelligences artificielles se basant sur le même principe de d'exploration exhaustive, et nous n'avons pas pu apporter de solution plus élégante au problème. Dans l'optique d'une exploration et évaluation des coups possibles, il est intuitif de développer une simulation du jeu qui permettrait d'évaluer la pertinence d'un coup en comparant les différents paramètres de la simulation avant et après qu'il ait été simulé.

La simulation devait être la plus fidèle possible, nous voulions disposer d'une réplique du moteur du jeu ; la précision de l'évaluation des coups dépendant de la précision de notre simulateur, la pertinence des meilleurs coups trouvés serait directement impactée par des erreurs de simulation.

Les intelligences artificielles existantes, du moins celles pour lesquelles nous avions accès au code source, utilisent des simulateurs grossiers et les automates embarquant ces *IA* dépassent rarement le rang 10. Nous devions donc nous atteler à l'élaboration d'une simulation précise et robuste.

Nous voulions également arriver rapidement à un produit semi-fini prêt à passer les phases de tests. Nous ne pouvions donc pas envisager d'implémenter les éléments de contexte[2] d'une part puis toutes les cartes *en dur*[3] d'autre part. Notre encadrant nous a donc encouragés à étudier la question de la factorisation des interactions, entre les cartes et les éléments de jeu, en une grammaire abstraite pour parvenir à une solution plus efficace.

Cette factorisation était en effet cruciale puisque la principale difficulté dans la modélisation de *Hearthstone* est d'interpréter le texte des cartes ; comme cela a été mentionné dans l'introduction, les textes correspondent à de nombreuses mécaniques différentes et la factorisation permettait de réduire de manière conséquente le temps de débogage suivant l'implémentation d'un ensemble de cartes. Pour y parvenir, nous avons mis au point un standard de désambiguïsation des cartes, et étendu ce principe à un langage de notre cru permettant la description complète d'une partie.

2.2.1 Modèle et langage descriptif

Modèle incrémental

Notre première tentative fût de parcourir toutes les cartes en étendant le modèle à chaque fois qu'un texte ou une carte ne pouvait être décrit. Vite dépassés par la quantité de travail que cela représentait, cette approche a très vite été abandonnée au profit d'une conception incrémentale. L'idée était de décrire, implémenter, tester et déboguer les cartes par niveaux.

Les niveaux proposés correspondent à des paliers de complexité dans le sens des mécaniques[4] : pour toute carte il existera un niveau minimal en dessous duquel le modèle ne sera plus assez puissant pour la modéliser entièrement.

2. Le *canevas* informatique nécessaire à la simulation d'une partie avec des cartes
3. Une classe *C#* pour chaque carte comme dans le projet *HearthSim* par exemple
4. Mécanismes induits par les textes de cartes.

Niveau 0 - Pour initialiser la construction incrémentale du modèle et du simulateur nous avions besoin de décrire un minimum de cartes pour pouvoir simuler une partie. Nous avons donc décidé qu'à ce niveau, seules les cartes *serviteur* seraient modélisées en faisant abstraction de leurs textes ; un *serviteur* est donc représenté par son nom, son coût, son attaque et sa vie.

Exemple 1 *Description de carte niveau 0*

```
Yéti Noirot
{
        Coût : 4
        Attaque : 4
        Vie : 5
}
```

Niveau 1 - Nous avons ensuite introduit les textes les plus simples. Ces textes correspondent à des mécanismes génériques puisque la simple mention du nom de la mécanique dans le texte suffit à l'appliquer : aucune information supplémentaire ne doit être extraite du texte. Ces mécanismes sont appelés *mécaniques passives génériques* et ne concernent que les cartes *serviteur* . La description est donc étendue d'un champ, celui des mécaniques.

Exemple 2 *Description de carte niveau 1*

```
Marche-Soleil
{
        Coût : 6
        Attaque : 4
        Vie : 5
        Mécaniques : Provocation, Bouclier Divin
}
```

Niveau 2 - Certaines mécaniques passives ne sont pas génériques. Pour extraire une règle d'application du texte « Bonus aux dégâts des sorts[5] : 4 », deux informations doivent être extraites ; la partie *abstraite* de la mécanique, **Bonus aux dégâts des sorts**, et la quantité de ce bonus à appliquer, **4**.

L'objectif au niveau 2 était d'aboutir à une description de ces mécaniques. La description suit le même principe qu'au niveau précédent, à l'exception des mécaniques non génériques qui doivent être représentées par un doublet.

Exemple 3 *Description de carte niveau 2*

```
Mage de Dalaran
{
        Coût : 3
        Attaque : 1
        Vie : 4
        Mécaniques : (Bonus aux dégâts des sorts, 1)
}
```

5. Un bonus aux dégâts des sorts de X augmente de X les dégâts de base des sorts.

Niveau 3 - À ce stade de la modélisation les mécaniques de serviteurs passives sans activation étaient toutes prises en compte. Restaient les mécaniques actives, qui concernent des textes dont les applications sont très variées. Nous nous sommes intéressés - à ce niveau 3 - uniquement à la description des effets non-aléatoires que ces textes induisent. Il n'existe donc pas de carte de niveau 3 à proprement parler.

Définition 1 *Un* effet *est la donnée d'au plus trois paramètres :*

1. *Le* type*, paramètre obligatoire. Le type d'un effet correspond à la partie générique de son texte, c'est à dire la partie du texte qui peut être commune à tout un ensemble d'autres cartes*[6]*.*
2. *Une* portée*, paramètre optionnel. La portée correspond à la manière d'appliquer l'effet. L'application peut être globale*[7] *ou ciblée. C'est la présence des mots-clés "à tous les.." ou "à un" dans le texte correspondant à l'effet qui déterminera la valeur de ce paramètre. Pour certains effets comme la pioche ou l'invocation de serviteurs la portée est la même pour toutes les cartes et il n'est donc pas nécessaire de renseigner ce champs.*
3. *Une* précision*, paramètre optionnel. Certains types d'effets sont abstraits dans le sens où ils nécessitent une information supplémentaire dans le texte pour être appliqués. Cette précision peut être une quantité ou le nom d'une carte.*

Exemple 4 *Les textes* Infliger 3 dégâts à tous les... *et* Infliger 6 dégâts à un.. *correspondent à des effets qui partagent le même type* ***Dégâts****. En revanche leur portée diffère puisque le premier est* ***global*** *alors que le second est* ***ciblé****. La précision est également différente,* ***3*** *pour le premier et* ***6*** *pour le dernier.*

Rend 3 points de vie à un.. → *(Soin,Ciblé,3)*
Invoque un Diablotin.. → *(Invocation,Diablotin)*
sont d'autres exemples de textes et comment ils sont désambiguïsés.

Niveau 4 - Le niveau précédent posait les bases de la modélisation des mécaniques actives. C'est à ce niveau qu'ont été introduits tous les types de cartes. Les textes des sorts correspondent à des mécaniques actives, et les mécaniques de serviteurs manquantes telles que *Râle d'agonie* ou *Cri de guerre* allaient pouvoir être elles aussi décrites. C'était également l'occasion de faire évoluer la simulation de manière à ce qu'elle émule l'utilisation des armes.

Définition 2 *Une* mécanique active *est la donnée de deux paramètres :*

1. *Un* effet*, comme décrit plus haut*
2. *Un* ensemble de cibles*. Cet ensemble correspond aux cartes en jeu qui peuvent être concernées par l'application du texte.*

Exemple 5 *En reprenant et complétant les textes de l'exemple précédent*

Infliger 3 dégâts à tous les personnages	*[(Dégâts,Global,3),* ***Tous****]*
Infliger 6 dégâts à un personnage	*[(Dégâts,Ciblé,6),* ***Tous****]*

6. L'énumération est fournie en Annexe A
7. Aucune cible ne doit être désignée par le joueur

Dans le premier cas Tous *les serviteurs et héros seront touchés lorsque ce texte sera appliqué. Dans le second le joueur doit choisir une cible parmi* Tous.

Description de carte niveau 4

```
Cercle de soins
{
        Coût : 0
        Mécaniques : Sort[(Soin, Global, 4), Serviteurs]
}
```

Le niveau 4 est le plus haut ayant été implémenté, et le tableau récapitule les différents niveaux imaginés (y compris ceux manquants), par ordre croissant de *complexité* au sens de la modélisation et les mécaniques ajoutées par rapport au niveau précédent.

Niveau	Mécaniques	Exemples de texte (".") ou carte
0	-	-
1	Passives simples	"Charge", "Provocation", "Camouflage"
2	Passives quantifiées	"Surcharge(X)", "Dégâts des sorts(X)"
3	-	
4	Actives	"Cri de guerre..", Sort
5	Aléatoires	"..X-Y dégâts", "Pioche..","Défausse.."
6	Événementielles	"Lorsque..", "A chaque fois que.."
7	Conditionnelles	"Si..", "Tant que.."
8	Exceptionnelles	"Copier une carte de la main adverse"

FIGURE 2.2 – Modélisation incrémentale

Un projet libre[8] fournissait l'ensemble des cartes au format *JSON*. Ce *JSON* contenait toutes les informations explicites de la carte (coût, attaque, vie, mécaniques passives simples) et son texte (sous forme de chaîne de caractères). Ainsi disposions-nous de toutes les informations nécessaires, ne restait qu'à désambiguïser ces chaînes de caractères. Le standard que nous avons imaginé pour expliciter les textes est le suivant :

CARD_ID{

$$\text{Mécanique}_1[\text{Effet}_1^{(1)} : \text{Cibles}_1^{(1)}, \dots, \text{Effet}_p^{(1)} : \text{Cibles}_p^{(1)}]\,;$$
$$\vdots$$
$$\text{Mécanique}_n[\text{Effet}_1^{(n)} : \text{Cibles}_1^{(n)}, \dots, \text{Effet}_q^{(n)} : \text{Cibles}_q^{(n)}]\,;$$

}

où un effet est décrit par un triplet et les ensembles cibles prennent des valeurs énumérées en Annexe, comme défini précédemment.

Remarque 2 *À l'heure à laquelle ce mémoire est écrit, nous utilisons deux fichiers qui permettent de décrire un peu plus de deux cents textes allant jusqu'au*

8. pour plus d'informations, `http://hearthstonejson.com/`

niveau 4 de modèle. Ces fichiers ont été produits à l'aide de mini-*programmes facilitant la description des cartes par un humain, les résultats étant compilés en un fichier texte exploitable par notre suite logicielle.*

Langage descriptif

En avançant sur la modélisation des cartes, nous nous sommes demandés s'il était possible d'étendre le principe aux parties jouées. Était-il possible de modéliser globalement les phases de jeu, et si oui, comment ? Était-il possible de coder les informations représentant le déroulement d'une partie du début à la fin ? Cette modélisation, si elle était possible, nous aurait alors permis de poser des bases saines pour l'implémentation du moteur de jeu.

Hearthstone étant par essence un jeu au tour par tour, nous avons naturellement essayé de représenter chaque tour par un ensemble de données le décrivant presque totalement [9]. Nous nous sommes rapidement rendu compte qu'un tour était une succession d'états de jeu et de séquence d'actions, aussi vastes que ces définitions puissent être.

En effet, au début du tour d'un joueur, l'ensemble des créatures sur sa partie du plateau et des cartes de sa main est dans une configuration donnée. Après quoi le joueur peut poser des créatures, jouer des sorts, attaquer les créatures de l'adversaire, etc... A la fin de sa série d'actions, le joueur passe son tour et, par symétrie, l'adversaire débute son tour, réalise la même séquence de jeu, et ainsi de suite jusqu'à la victoire d'un des deux protagonistes.

Pour pouvoir suivre le fil de la partie, il est donc important d'avoir chaque tour à la fois la configuration de l'ensemble du plateau/mains et les actions effectuées qui vont faire évoluer ces configurations.

Toute cette réflexion nous a poussé à mettre au point un format de description d'états de partie directement inspiré des premiers schémas utilisés en exemple au début du projet, comme montré ci-dessous.

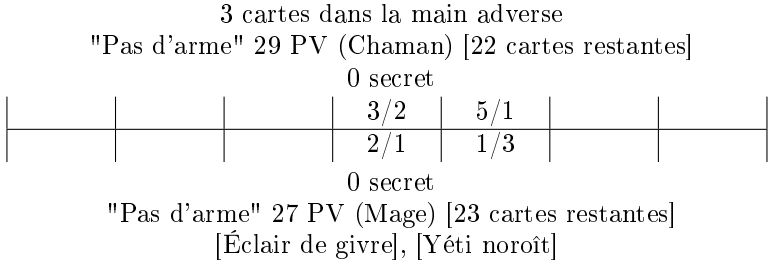

FIGURE 2.3 – *Schéma d'une situation simple entre un Chaman et un Mage*

C'est ce schéma qui nous a amené à penser au format **HSC** (**H**earth**s**tone **C**onfiguration). Il s'agit d'un langage déclaratif simple permettant de représenter un état de partie à un moment donné grâce à quelques lignes. De nouveau à titre indicatif, voici quelques exemples de lignes tirées d'un fichier *.hsc* :

9. Les cartes et le *deck* de l'adversaire étant inconnus, nous chercherons à décrire tout ce qui peut l'être.

```
ALLY_HERO  CARDID=HERO_08  MAX_HEALTH=30  CURRENT_HEALTH=28
           ATK=0  ARMOR=0  MANA=4  MANA_USED=0  EXHAUSTED=0
```

Cette ligne permet de décrire toutes les informations concernant un héros (ici le héros allié). Toutes les classes/cartes sont codées par un identifiant unique utilisé dans la base de données de cartes de *Hearthstone*, cela pour éviter les confusions entre des cartes homonymiques. Par exemple ici *HERO_08* correspond au héros *Mage* ; le booléen *EXHAUSTED* permet de savoir si le héros (ou une créature) à déjà attaqué à cet état de jeu.

Une autre ligne d'un fichier *.hsc* possible concernant une créature adverse :

```
ENEMY_BOARD  CARDID=NEW1_010  ENTITY_ID=62  ATK=3  MAX_HEALTH=5
             CURRENT_HEALTH=3  ZONE_POSITION=1  EXHAUSTED=1
             SHIELDED=1  TAUNT=1  WINDFURY=1
```

Dans cet exemple on voit que l'adversaire a posé sur son côté du plateau **Al'Akir** (*NEW1_010*), et que celui-ci possède encore *Provocation* et *Furie des vents* mais qu'il ne peut plus attaquer.

Remarque 3 *Le nombre de cartes restantes dans le deck d'un joueur n'étant pas une donnée essentielle pour décrire un état de partie, nous l'avons mis de côté.*

Dans un même temps, il nous a fallu trouver l'ensemble des différentes actions possibles pour un joueur au sein d'une phase de jeu. Nous les avons identifiées et factorisées au nombre de 5, pour les intégrer dans un fichier de séquences d'actions au format **HSS** (**H**earth**s**tone **S**cript) :

Mot-clé HSS	Action
draw <*carte*> id	Le joueur pioche <*carte*> avec un identifiant *id* pour le reste de la partie.
playFromHand id pos (cible)	Le joueur joue la carte *id* de sa main à la position *pos* et si elle a un effet ciblé, prend pour cible *cible*.
newTurn	Permet simplement d'indiquer le changement de tour.
heroPower (cible)	Le joueur utilise son pouvoir héroïque, avec une cible s'il peut en prendre une.
trade *att cible*	La créature *att* attaque et prend pour cible *cible*.

En ajoutant au début du fichier *.hss* une primitive **load** indiquant un fichier *.hsc* à prendre en entrée, il nous était alors possible de prendre une situation initiale et de lui appliquer une suite d'action aboutissant à un *.hsc* final.

Forts de cette modélisation du déroulement du jeu, nous avons voulu pousser un peu plus loin le concept.

Si une partie se décrit comme étant une succession d'états avec des actions entre elles, alors en partant d'un état initial C_1 auquel on applique une séquence d'actions S_1, on obtient un état suivant C_2. On peut alors lui appliquer une nouvelle séquence d'actions S_2, etc ...

Cela à eu pour conséquence immédiate la fusion des deux fichiers - HSC et HSS - en un fichier **HSG** (**H**earth**s**tone **G**ame) représentant l'ensemble du déroulement d'une partie.

2.2.2 *HearthSimulation*

HearthSimulation est le nom du paquet fournissant les classes nécessaires à la simulation d'un coup ou d'une partie. Outre la myriade de classes décrivant les différents éléments de contexte (plateau, *deck*, main, mécaniques, ou encore les types énumérés), quelques unes méritent une attention particulière :

Simulation, contient 2 *board*, 2 mains, 2 *Héros* et 1 "routeur"

Card, une carte, peut être un serviteur, un *Héros*, un sort, une arme

Enchant, enchantement pouvant se lier à une carte

Routeur, permet de tenir un manifeste des cartes mises en jeu lors d'une partie et de localiser rapidement la zone de jeu ou sa position dans la zone dans laquelle se trouve la carte.

Remarque 4 *Lorsqu'une carte est instanciée dans la simulation elle récupère auprès d'une base de données, chargée localement, la description standard de son texte puis elle est ajoutée aux tables du routeur. À chaque carte peut être attaché un enchantement comme il est décrit dans notre modèle. Ces classes constituent le coeur de notre simulateur.*

Simulation met à disposition du programmeur un ensemble de méthodes permettant soit d'appliquer directement une mécanique décrite selon notre modèle soit de procéder à une action de jeu [10] et à l'application de ses conséquences.

Une classe "boîte-à-outils" a également été développée pour manier des fichiers ou des chaînes de caractère en langage *HSS, HSC* et *HSG*. Elle permet par exemple d'instancier une simulation dans une situation décrite par une *HSC*, d'exécuter une série d'actions tirées d'un *HSS* ou encore de simuler la totalité d'une partie sauvegardée dans une *HSG*. Les opérations inverses (sauvegarde d'une situation ou d'une partie) étant également implémentées, cette boîte-à-outils est la pierre angulaire du calcul de meilleur coup puisqu'elle permet de revenir sur nos pas lors d'une exploration de l'espace de recherche.

À ce jour, notre simulation de *Hearthstone* est stable, dans la mesure où plusieurs semaines ont passé depuis le dernier bogue repéré, et prend en compte les textes jusqu'au niveau 4 de modèle.

10. Jouer une carte, utiliser le pouvoir héroïque, faire attaquer ou finir son tour

2.3 Calcul de meilleur coup

Un des obstacles majeur à l'aboutissement d'une intelligence artificielle telle que nous la souhaitions, outre le fait de pouvoir simuler *Hearthtstone* correctement, était de pouvoir juger la qualité d'un coup par rapport à un autre.

En effet, la question se pose : qu'est-ce qui différencie un coup d'un autre ? Qu'est-ce qui fait qu'un coup est *meilleur*, ou *pire*, qu'un autre ?

Comme dans beaucoup de jeux, cette notion de *bon coup* peut être facile à concevoir pour n'importe quel joueur, mais en réalité elle est beaucoup plus fine et difficile à saisir dès lors que l'on cherche à l'étudier et la théoriser.

On aurait par exemple tendance, comme beaucoup de joueurs, à vouloir utiliser notre vie restante et celle de l'adversaire comme métrique pour juger si on est en train de gagner, ou de perdre. Bien quelle soit utile pour cela, elle ne suffit pas car un joueur à 10 points de vie face à un adversaire qui en a 30 a beaucoup plus de chance de gagner si cet adversaire n'a plus aucune carte. Dans le même ordre d'idées, si un joueur à un plateau rempli de créatures et plus aucune carte dans la main, et que son adversaire n'a aucune carte en jeu et quelques cartes en main, si ce dernier utilise un sort pour tuer toutes les créatures en jeu par exemple, il obtient un net avantage, son adversaire ayant trop misé sur la présence sur le plateau.

Afin de s'approcher de ces notions clés sur le jeu, nous nous sommes penchés sur les indices et les réflexions sur lesquelles se basent les très bons joueurs de *Hearthstone* et des jeux de cartes similaires (les jeux de duels basés sur des cartes à collectionner, les TCG/CCG [11], ...).

Nous avons alors remarqué que certain points se retrouvaient souvent dans les explications données par ces joueurs à propos de leur style de jeu, de *decks* [12], etc... Mais concernant plus précisément *Hearthstone*, il y a une notion essentielle qui permet de s'assurer une bonne manière de jouer et donc d'augmenter son ratio de victoire : l'*avantage des cartes* [13].

Définition 3 *Un joueur possède l'avantage des cartes sur son adversaire si la somme de ses cartes en main et de ses cartes en jeu est supérieure à celle de son adversaire.*

En partant de ce principe on peut affirmer qu'un joueur est en train de gagner, dans la majorité des scénarios, si il possède à la fois l'*avantage des cartes* et le *contrôle du plateau*.

Ce qui amène évidemment à la question suivante : Comment avoir l'avantage des cartes et le contrôle du plateau ?

Les deux facteurs amenant à ces avantages sont les *mécaniques de pioche* et les *trades efficients* [14].

Les mécaniques de pioche sont un concept simple à comprendre. Si un joueur joue de sa main une carte le faisant piocher une ou plusieurs cartes, il a alors

11. TCG : Trading Card Game, CGC : Collectible Card Game

12. La façon dont un *deck* est conçu influe également beaucoup sur les chances de victoire mais n'est pas abordée ici.

13. *Card Advantage* dans sa version originale.

14. Le mot français étant *échange de coups*, nous préférerons par la suite le terme anglais par facilité d'écriture.

au moins autant de cartes dans sa main. Cela lui permet notamment de faire tourner son jeu et par conséquent de disposer de plus de solutions. C'est ce pourquoi les cartes à mécaniques de pioches sont si utiles dans un jeu où on ne pioche en principe qu'une carte par tour (au début d'un tour).

Les *trades* efficients sont en revanche beaucoup plus fins à comprendre et à mettre en pratique. C'est ce point qui forme la barrière pour entrer dans le milieu compétitif de *Hearthstone*. Cette notion s'explique en deux parties : le *Trade Up* et le *Trade* $1-N$, N étant un entier.

Le *Trade Up* est le fait d'utiliser une carte faible ou à faible coût pour tuer une carte plus forte. Cela permet de garder ses créatures fortes alors que leur nombre diminue pour l'adversaire. Par exemple, si l'adversaire pose un *Ragnaros*[15], et que l'on utilise *Assassiner*[16] alors on a effectué un *Trade Up*. Idem si on tue tout simplement une créature 5/1 (pour 3 mana) avec une 1/1 (pour 1 mana). Ce procédé est en fait un *Trade 1-1*, dans le sens où on utilise une carte pour en gérer une autre, mais de plus faible valeur, générant un léger avantage de cartes.

Le *Trade* $1-N$ est la généralisation de cette notion. Le principe est de maximiser le nombre de cartes jouées par l'adversaire pour gérer chaque carte alliée. Plus N est grand, plus l'adversaire a joué de cartes pour la contrer, et plus on a gagné en avantage de cartes.

En plus de cette notion d'*avantage de cartes* très importante, il est nécessaire de rappeler qu'elle n'est pas la seule à rentrer en compte. En effet la vie des héros, le nombre de cartes en jeu, le potentiel d'attaque des créatures, la puissance des sorts, l'aléatoire, etc... sont également à prendre en compte. C'est pourquoi après cette étude plus précise d'un *coup* et sa qualité, nous avons tenté de modéliser les divers coefficients pouvant être utiles au calcul d'un meilleur coup.

2.3.1 Fonction de *scoring*

La fonction d'évaluation (ou fonction de *scoring*) utilisée pour mettre une valeur sur un coup est une somme de coefficients reflétant au mieux les notions abordées plus haut. Cette fonction a été sommairement imaginée au début du projet, avant d'être refondue pour la rendre homogène. Le fait qu'elle soit au cœur du projet a été la principale raison pour laquelle elle a été ajustée à maintes reprises, pour la rendre au fur et à mesure de plus en plus précise et cohérente.

Avant de poser le résultat final qu'est la fonction de scoring, nous allons d'abord présenter les divers coefficients permettant son calcul. Les valeurs ayant un indice **AC** sont des valeurs **A**près **C**oup, c'est à dire une fois que le coup a été simulé, les autres étant les valeurs à l'état initial, *i.e* avant ce coup.

Soit **CC** le **C**oefficient de **C**ontrôle représentant l'écart de menace (en terme de potentiel de dégâts infligeables) après un coup, défini tel que :

$$CC = \frac{\sum Attaque_{allieeAC} \times \sum Attaque_{ennemie}}{max(1, \sum Attaque_{alliee} \times Attaque_{ennemieAC})}$$

15. Serviteur 8/8 (coût 8) : Ne peut pas attaquer. À la fin de votre tour, inflige 8 points de dégâts à un adversaire aléatoire.
16. Sort (coût 5) : Détruit un serviteur adverse

Soit **CP** le **C**oefficient de **P**erte, représentant la notion d'avantage de présence sur le plateau après un coup, défini tel que :

$$CP = \frac{|Creatures_{allieesAC}| \times |Creature_{ennemies}|}{max(1, |Creatures_{alliee}| \times |Creatures_{ennemiesAC}|)}$$

On note alors **X** le Coefficient de Gestion de Menace, permettant d'allier les deux coefficients précédents, défini tel que :

$$X = CC + CP$$

Soit **Y** le Coefficient de Létalité, représentant l'écart entre les points de vie des joueurs après un coup, défini tel que :

$$Y = \frac{VieHeros_{allie} \times max(1, Attaque_{allieeAC})}{VieHeros_{ennemi} \times max(1, Attaque_{ennemieAC})}$$

Soit **Z** le Coefficient de Cartes, cherchant à maximiser le nombre de cartes en jeu alliées et minimiser le nombre de cartes en jeu ennemies, et défini tel que :

$$Z = \frac{|Cartes_{allieAC}| \times |Cartes_{ennemi}|}{max(1, |Cartes_{allie}| \times |Cartes_{ennemiAC}|)}$$

Si les coefficients **X** et **Y** sont de même valeur, alors le coefficient **Z** permettra de départager les deux coups en privilégiant celui utilisant le moins de cartes.

Grâce à ces coefficients, nous pouvons alors définir la *fonction de scoring* **S** telle que :

$$S = aX + bY + cZ \qquad (a, b, c) \in \mathbb{R}^3$$

La fonction de *scoring* admet des variables d'ajustement a, b, c qui permettent de corriger empiriquement la valeur du score. Il est en effet admis qu'en fonction du type de *deck* joué ou de certains événements arrivant au cours de la partie, ces variables s'ajustent pour favoriser un coefficient plutôt qu'un autre.

À l'heure actuelle, cette fonction d'évaluation S a fait ses preuves, et bien qu'elle ne soit pas encore totalement aboutie, elle a permis de mettre en exergue une intelligence certaine dans les coups proposés, ce qui est plutôt satisfaisant pour un jeu aussi complexe que *Hearthstone*.

2.3.2 *HearthAI*

La simulation du jeu et la fonction d'évaluation précédemment présentées sont tout ce dont a besoin une intelligence artificielle pour calculer une meilleure suite de coups étant donnée une situation de *Hearthstone*.

L'exploration et évaluation a l'avantage d'être un algorithme de recherche exhaustive relativement simple, facile à concevoir et à implémenter. L'Algorithme 1 est le premier algorithme de backtrack récursif que nous avons implémenté.

Algorithme 1 : ExplorerEvaluer

```
/* L'algorithme utilise une fonction d'évaluation, qui sera
   notée f : {Coups possibles} → ℝ                          */
```

Entrées : Une simulation du jeu au tour allié, notée S
Une pile de coups courante, notée $C_{courante}$
Une pile de coups optimale, notée C_{opti}
Résultat : C_{opti} contient la suite de coups dont le score est le plus élevé
début
  Noter P l'ensemble des actions alliées réalisables dans S ;
  si P *est vide* **alors**
    si $f(C_{opti}) < f(C_{courante})$ **alors**
      $C_{opti} \leftarrow C_{courante}$;
  sinon
    Noter $s_{initiale}$ la situation de jeu courante ;
    pour chaque $A \in P$ **faire**
      pour chaque *cible potentielle* x **faire**
        Simuler A sur x dans S ;
        Ajouter (A, x) à $C_{courante}$;
        $Explorer\&Evaluer(S, C_{courante}, C_{opti})$;
        Retirer (A, x) de $C_{courante}$;
        Revenir à $s_{initiale}$ dans S ;

Remarque 5 *Le coup retourné est celui dont le score était le plus élevé lors du parcours exhaustif des possibles. La fonction de scoring permet d'évaluer une partie des possibilités futures puisque certaines quantités utilisées lors du calcul correspondent à une potentielle menace au tour suivant.*

Hearthstone étant un jeu à information incomplète, des algorithmes comme celui du *min-max*, qui améliorent la pertinence de la fonction d'évaluation en tentant de prédire ce qu'il se passera les tours suivants (alliés et ennemis), sont difficiles à mettre en place. Les cartes que l'ennemi a en main sont inconnues, et nous ne disposions pas de moyen d'inférer la présence de telle ou telle carte dans le jeu ennemi. On sait par contre ce que contient déjà le *board* ennemi (avant le début de son tour), et également la nature de son pouvoir héroïque. On dispose donc de quelques actions ennemies possibles pour évaluer les risques auxquels le joueur allié sera exposé lorsqu'il cliquera sur le bouton *Fin de tour*.

L'algorithme procédant à cette prédiction est une extension de celui proposé plus haut. La différence se fait au niveau des appels récursifs terminaux. Au lieu de comparer les scores des suites de coups courants et optimaux, on compare ces scores après avoir simulé un meilleur coup pour l'ennemi avec les informations connues. Le changement est mineur et les résultats sont bien meilleurs. C'est cette variante qui est déployée dans la version actuelle[17].

17. Au 25 Mai 2015

2.4 Récupération des situations de jeu

La deuxième contrainte technique fût celle de la récupération des informations fournies par le jeu lors d'une partie. Ces informations sont diverses, comme les points de vie des différents héros et des serviteurs sur le plateau, mais aussi l'identité et les caractéristiques des serviteurs (sur le plateau et dans la main) et les éventuels enchantements ajoutés (ou enlevés) sur chaque serviteur. Ces informations sont la base de toute simulation, et plusieurs solutions ont été proposées, pour finalement n'en retenir qu'une.

2.4.1 Premières tentatives

Capture réseau

Une solution était d'écouter les communications entre *Hearthstone* et le serveur de jeu *Blizzard* et de capturer les paquets tout au long de la partie. En se basant sur le projet *Hearthy*[18], un décodeur réseau du protocole utilisé par *Hearthstone* et développé en python, il était relativement aisé de créer notre propre décodeur et de récupérer les paquets circulant pour mettre à jour une base de données dynamique représentant le jeu à un instant t. Malheureusement, le mois qui suivit, *Hearthstone* fût mis à jour et le protocole de communication par la même occasion. Ceci souleva des inquiétudes quand à la suite du projet, et il nous fallait trouver rapidement une solution. Le risque que le protocole soit de nouveau mis à jour avant la fin du projet nous ôtait le choix de garder la capture réseau comme solution fiable pour la récupération des situations de jeu.

Reconnaissance d'images

Forts de notre mésaventure avec la capture réseau, nous cherchions une solution stable pour l'analyse et la récupération des informations que le jeu fournit. Dans ce second temps, nous nous décidâmes à développer un analyseur d'images capable d'extraire depuis une image du jeu toutes les informations possibles. L'idée était de faire des impressions écran successives, offrant la dernière capture en date pour l'analyse de l'image. Chaque image se voyait découpée en zones, chaque zone pouvant contenir elle même plusieurs sous-zones, pour finalement analyser, comparer et extraire l'information en chaque zone.

Deux problèmes existent avec cette solution : le premier est la consommation CPU, bien plus élevée que dans le cas de la capture réseau ; le second est qu'il faut connaître toutes les images des cartes, héros et serviteurs avant de pouvoir déduire ce qu'il y a dans une zone. Un script python nous permit de récupérer l'ensemble des cartes sous forme d'images, chaque image étant nommée par l'ID dans *Hearthstone* de ladite carte. Un fichier JSON permet de faire le lien entre ID et caractéristiques de la carte.

Imaginons nous dans la phase d'analyse de l'image, plus précisément dans une zone non-divisible. Imaginons aussi que cette zone corresponde à l'emplacement d'un serviteur sur le plateau. Alors, en premier lieu, il faut comparer si l'image est différente ou non de la dernière image traitée. Si oui, alors il faut chercher à savoir ce qui a changé, si non alors on peut ignorer la nouvelle image. Pour chercher efficacement dans la base de données, nous avons implémenté

18. projet libre sur GitHub

plusieurs fonctions de recherche, se rapprochant des techniques de *branch and bound*, où nous élaguions les branches de l'arbre si les moyennes des couleurs de l'images n'étaient pas assez proches, réduisant ainsi considérablement le temps de recherche d'une image correspondante dans la base de donnée.

Finalement, une solution alliant efficacité et fiabilité fût découverte, et le projet de reconnaissance d'images fût abandonné.

2.4.2 *HearthDelogger*

Alors que cette solution fût précédemment mise de côté car n'apportant pas suffisamment de détails sur l'état du plateau, et ne semblant pouvoir fournir qu'une partie des situations de jeu, les journaux de débogage générés par *Hearthstone* sont devenus à terme l'unique source d'information. Il nous aura fallu trouver, dans le code source d'applications utilisées par certains joueurs de *Hearthstone*, le bon fichier de configuration pour activer les logs de débogage. Une fois ledit fichier correctement placé, un fichier journal contenant toute information utile au débogage pour les développeurs de *Hearthstone* est généré à chaque lancement de jeu ; il ne restait plus qu'à le comprendre.

Le débogage est technique. Bien que formaté, et relativement compréhensible à une lecture humaine, le but de ce rapport n'est pas de rentrer dans ce genre de détails. Notons tout de même qu'à chaque action, réalisée par le joueur ou automatique, les lignes de débogages correspondantes sont crées dans le journal ; à nous ensuite de les filtrer, les interpréter, les mettre en relation, et les stocker, pour obtenir une base de données complète et saine correspondant à une situation de jeu.

Basiquement, l'on gère le jeu comme deux joueurs possédant chacun un héros, une main et un certain nombre de serviteurs en jeu. Puisque le *parsage* de fichier n'est, en soit, pas un exercice coûteux si intelligemment réalisé, il fallait essayer de limiter au maximum la demande de ressources au processeur. Pour ce faire, des mécaniques d'introspection et des structures de données précisément choisies ont permis d'alléger au maximum l'impact du *thread* correspondant. Ainsi, toute requête à la base de données se fait en *O(1)*, et la redondance est évitée.

Remarque 6 *Nous nommerons* scope *l'ensemble des actions se faisant d'une même foulée. Par exemple lorsqu'un des joueurs pose une carte, un serviteur avec un* Cri de Guerre*, un scope contiendra au moins le fait que le joueur pose ladite carte, l'emplacement du nouveau serviteur sur le plateau, et les effets éventuels de son* Cri de Guerre.

À chaque lecture complète d'un scope d'actions, le *thread delogger* avertira le *thread* gérant la simulation qu'il doit mettre à jour sa situation de jeu. Quand un scope précisira que c'est à nous de jouer, et une fois la simulation à jour, le calcul du meilleur coup sera lancé sur la partie I.A. Un coup est proposé au joueur, et on attend qu'il intéragisse avec le jeu. Le prochain scope correspond à son jeu, s'il correspond à celui attendu, on compare la simulation avec celle attendue par le calcul du meilleur coup ; si la simulation devait être différente, soit parce que le joueur a joué sans prendre compte du conseil, soit parce que la simulation n'était pas correcte ou que certains effets aléatoires ne pouvaient pas être prédis, alors un nouveau calcul du meilleur coup est initié.

2.5 Automatisation du processus

2.5.1 *HearthPlayer* - Automate joueur

Motivations

Une fois qu'on est en mesure de calculer la meilleure séquence de coups à jouer à un instant de la partie, la prochaine étape est d'automatiser le processus. Faire jouer son ordinateur à sa place peut sembler être un objectif étrange pour un néophyte des jeux vidéos, mais la répétition de certaines actions ou objectifs à accomplir dans certains jeux a permis à de nombreux logiciels de voir le jour, afin d'effectuer ces tâches à la place des joueurs. Dans le cas de *Hearthstone*, le principal objectif d'un tel automate serait de jouer des parties de manière autonome afin de pouvoir gagner des pièces d'or, une monnaie d'échange permettant acquérir de nouvelles cartes ou débloquer certains contenus.

Les solutions

Il existe deux principales méthodes pour faire jouer l'ordinateur à sa place. La première, envisagée mais rapidement repoussée, fonctionne par injection de paquets dans les boîtes réseaux. Les inconvénients d'un tel procédé sont que cela peut être facilement repéré, et qu'elle n'est pas robuste à un changement de protocole de la part de *Blizzard*, comme on a pu l'expérimenter lors de la capture réseau.

La seconde méthode consiste à donner le contrôle du pointeur de souris au programme, via une librairie *Java* et donc reproduire les mouvements en jeu. Les mécaniques du jeu étant légion, il a été nécessaire de pouvoir reproduire toutes les façons d'effectuer une action (suivant la position de la carte dans la *main*, sa position sur le *board*, si le *serviteur* invoqué possède un *Cri de guerre* ciblé ou non, *etc...*). Pour des raisons de légalité qui seront évoquées plus en détail dans la quatrième partie de ce mémoire, l'automate *HearthPlayer* n'a pas été intégré dans le projet. Malgré tout, on peut s'interroger sur les difficultés que pourraient rencontrer un tel automate.

Limites et perspectives de solution

Actuellement, grâce à *HearthDelogger* on est en mesure de pouvoir traquer la situation du jeu à chaque instant. Cependant, il n'est *a priori* pas capable de repérer à quel moment l'animation d'une action prend fin. En effet, il ne faut pas confondre le moment où une action est écrite dans les journaux de débogage et le moment où elle apparaît sur son écran. Une solution "simple" serait de considérer que le temps d'une animation ne dépasse pas 2 secondes. Cependant, il se peut qu'une telle borne ne suffise pas dans certains cas ; un exemple extrême est le tour de force qui a été effectué par des joueurs consistant en une action de jeu durant plus de 45 heures.

De plus, lorsqu'on joue contre l'*IA* de *Blizzard*, toutes les actions du tour de l'*Aubergiste*[19] sont écrites dans les journaux de débogage dès le début de son tour, avant même qu'il ne commence à jouer quoi que ce soit. Dans ce cas, c'est le fait de devoir détecter le début de notre tour qui pose problème, même si

19. nom de l'IA de *Blizzard*

une reconnaissance d'image sur le bouton *Fin de Tour* est une solution que l'on peut facilement implémenter grâce au travail effectué en la matière.

Enfin, un dernier problème qu'on peut rencontrer est la potentielle existence d'un *Test de Turing* du mouvement de la souris par *Blizzard*. Les mouvements effectués par un humain du curseur de la souris ne sont pas aussi rectilignes et de vitesse aussi régulière que pour un automate "basique". On peut donc imaginer que *Blizzard* puisse détecter, seulement grâce à la souris, si c'est un humain qui l'utilise ou non. Suivant la robustesse de leur potentiel *Test de Turing*, il faudrait alors envisager de faire des courbes plus lissées à vitesse variable afin de se rapprocher autant que nécessaire du mouvement de la souris d'un être humain. N'ayant aucune information sur leurs protocoles de détection des automates, il nous est à l'heure actuelle impossible de savoir si *HearthPlayer* se ferait détecter ou non par *Blizzard*.

2.5.2 Communication entre composantes

D'un *game companion*

Une fois *HearthDelogger*, *HearthAI* et *HearthSimulation* fonctionnels nous avons pu imaginer un assistant de jeu affichant, *via* un *overlay*[20], la suite de coup calculée par l'IA à chaque nouvelle situation enregistrée par le *delogger*.

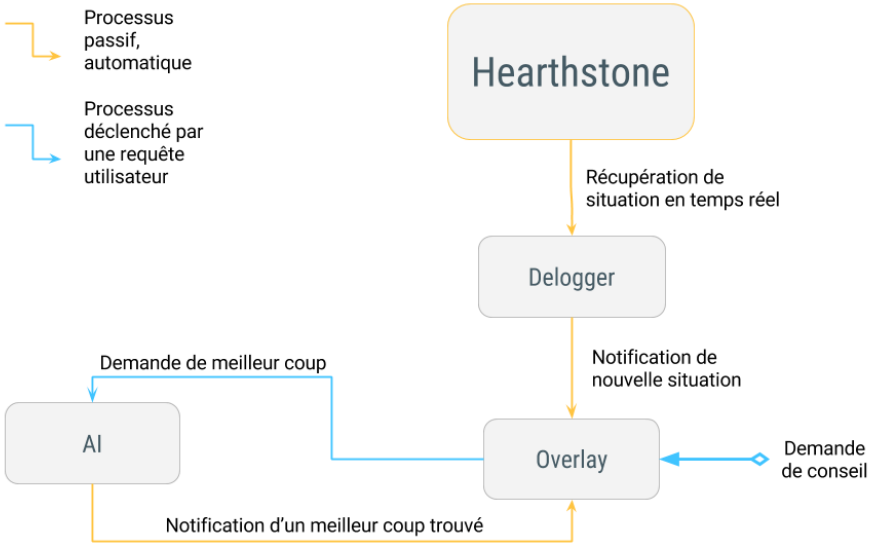

FIGURE 2.4 – Diagramme fonctionnel : game companion

Nous voyons clairement, dans ce diagramme, deux types de communication : l'une est de type *active* (il faut la déclencher), l'autre *passive* (elle advient sans demande). Une *demande de conseil* est initiée par le joueur, qui engendre la demande d'un meilleur coup à la partie AI. Le résultat retourné sera implicitement attendu par l'*Overlay*.

20. Fenêtre toujours au premier plan

A contrario de la communication *active*, des messages sont envoyés de manière spontanée entre les différentes composantes. Par exemple, le *Delogger* envoie à chaque fin de *scope* une notification prévenant qu'une nouvelle situation de jeu est disponible.

Vers un *bot*

L'assistant en jeu est très proche d'un automate. Il suffirait de remplacer l'entité *Overlay* par l'automate joueur *HearthPlayer*, nous obtiendrions alors l'automate complet qui était initialement demandé par le client. Pour des raisons légales qui seront évoquées plus loin, ce *bot* est resté théorique et nous proposons un diagramme fonctionnel palliant aux problèmes posés par l'automatisation du processus.

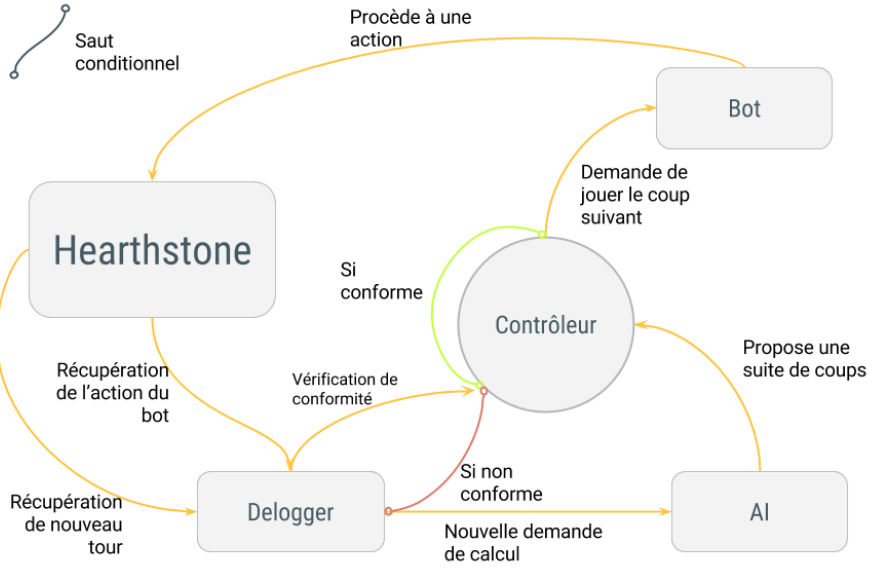

FIGURE 2.5 – Diagramme fonctionnel : bot

Pour comprendre ce diagramme, mettons nous dans le cas où nous débutons notre tour : on récupère la situation actuelle (au format *HSC* et disponible via le *Delogger*) et on lance le module AI sur cette situation. Une fois la suite de coups calculée, on demande au *Bot* d'effectuer le premier coup (c'est la flèche labellisée *Demande de jouer le coup suivant*). C'est alors le module *HearthPlayer* qui s'en charge et il faut attendre, dans les journaux de débogage, les modifications que le coup aura eu sur la situation de jeu (c'est le *Delogger* qui détectera la modification). Cette information est transmise au *Contrôleur* qui va vérifier si la situation actuelle correspond à la situation prédite par l'IA. Si c'est le cas, on demande au *Bot* d'effectuer le deuxième coup (s'il y en a un), et ce tant que la situation du jeu correspond à celle prédite et qu'il reste des coups à jouer. Dès lors que la situation diffère, le *Contrôleur* demande à l'AI de calculer une nouvelle suite de coups qui se base sur la situation actuelle (situation récupérée via le *Delogger*).

On remarque alors que quand la situation actuelle ne correspond pas à la situation attendue le calcul du meilleur coup est relancé, et que dans le cas où les situations sont similaires on évite de calculer une autre fois *pour rien* puisque nous obtiendrons le même résultat que précédemment. Ceci permet d'économiser du temps de calcul et de gagner en réactivité tout en étant le plus précis possible.

Pendant le tour adverse, le *Delogger* met à jour la situation de jeu dans sa base de données, et notifiera le *Contrôleur* quand notre tour arrivera (un autre exemple de *communication passive*).

Notons que dans le diagramme l'entité *Bot* peut correspondre au joueur humain comme à *HearthPlayer*. Le fonctionnement reste le même, c'est à dire qu'on recalcule une suite de coup dès lors que l'on a divergé de la suite proposée, et donc si le joueur humain fait fi de nos conseils, un coup proposé sera toujours le meilleur calculé pour une situation courante.

Chapitre 3

Déploiement

Nous allons à présent vous présenter les différentes versions de notre logiciel qui ont vu le jour, ainsi que les solutions utilisées pour faciliter le débogage. Le développement en méthode *agile* nous a forcé à faire des démonstrations régulières de notre produit. Les différentes composantes du projet n'ayant pas été développées à la même vitesse, nous avons dû imaginer des projets intermédiaires servant de support à la présentation de notre avancement lors des réunions. Ainsi avons-nous décidé, dans un premier temps, de concevoir un constructeur de *deck*. Puis, de faire évoluer ce logiciel en un assistant de suivi manuel de parties. Enfin, ce logiciel devait devenir obsolète à l'arrivée du premier produit fini : *HearthCompanion*.

3.1 Versions majeures

3.1.1 Assistant rudimentaire

Nous avons donc commencé par créer un modèle rudimentaire d'assistant, pouvant par la suite servir de base à des versions plus évoluées. Cet assistant ne devait alors que rappeler à l'utilisateur quelles ont été les cartes jouées, afficher celles dont la probabilité de les piocher était la plus élevée, et rappeler le *deck* dans son entièreté. Des options supplémentaires comme la création de *decks*, leur sauvegarde et chargement, l'affichage de la courbe de mana correspondante et des cartes qu'ils contiennent ont été ajoutés dans la foulée.

Pour répondre à cette problématique, il nous a fallu en premier lieu créer une base de données de toutes les cartes du jeu. La tâche fût aisée grâce au projet *Hearthstone JSON*[1] précédemment évoqué. Une simple lecture dudit fichier nous permettait donc de créer notre base de données à chaque ouverture de l'assistant. Dès lors qu'une mise à jour implémentant de nouvelles cartes serait disponible, il suffirait alors de remplacer le fichier *JSON* par un plus récent.

Dans un second temps, la partie GUI étant à ses balbutiements, il nous fallait acquérir toute les images des cartes du jeu pour plus de facilité à concevoir et analyser les *decks* créés. Sachant que des sites comme *hearthpwn.com* contiennent ces bases de données, nous avons écrit un petit programme en *Python* capable de télécharger et indexer lesdites images.

1. `http://hearthstonejson.com/`

L'assistant ainsi développé, bien que rudimentaire et sans réel intérêt de jeu, permettait tout de même de créer et sauvegarder des *decks*, ou même de les charger depuis le disque dur. Une fois un *deck* chargé, l'assistant simple - fournissant un suivi des probabilité de sorties et des cartes jouées lors d'une partie - pouvait être lancé. La figure ci-contre est une capture d'écran de cet assistant rudimentaire. On retrouve à gauche la liste des cartes encore dans le *deck* allié, à droite (en couleurs) les cartes déjà jouées par l'adversaire. Enfin, en bas se trouvent les 5 cartes ayant la plus haute probabilité de sortie à la prochaine pioche alliée. L'utilisateur peut rechercher une carte et l'ajouter aux cartes ennemies jouées. Les cartes qui ont encore une chance élevée d'être dans le *deck* ennemi sont en rouge, les autres en vert.

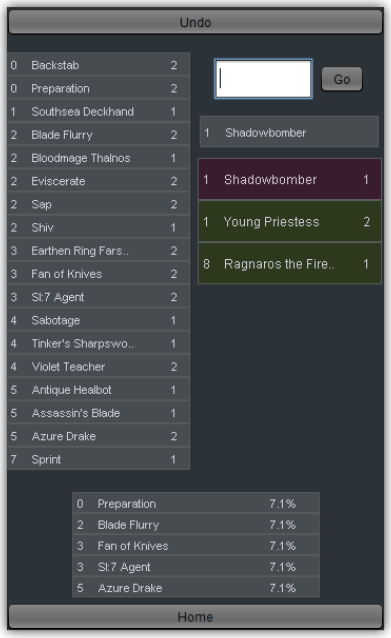

Fig. 3.1 – Assistant simple

Remarque 7 *Le format de* deck *utilisé est appelé* Cockatrice *; il est largement utilisé sur les sites proposant des* decks *préconstruits. De plus, la partie récupération des images de cartes ayant été menée à bien, les images des cartes ont pu être utilisées dans les autres interfaces graphiques, et dans le replay de parties.*

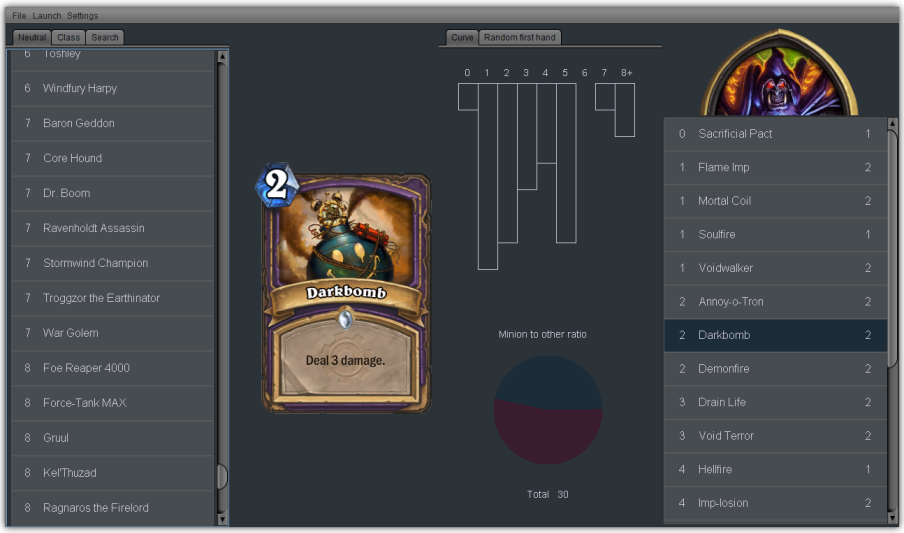

FIGURE 3.2 – Gestionnaire de *decks*

3.1.2 Assistant avancé

La livraison de l'assistant rudimentaire marquait la fin de la semaine de veille bibliographique, lors de laquelle le groupe s'attachait à dresser le bestiaire des projets libres autour de *Hearthstone*.

La stratégie de développement présentée dans le chapitre 2 pouvait commencer à être appliquée. *HearthDelogger* ayant été livré relativement tard, il nous fallait être en mesure de présenter notre avancement dans la partie simulation lors des réunions hebdomadaires. Nous avons donc décidé développer un programme de suivi manuel de parties. Deux développeurs ont pu travailler de manière synchrone à l'obtention du résultat visible en la figure suivante.

Fig. 3.3 – Assistant *avancé*

Un développeur se concentrait sur l'implémentation de la simulation, pendant que l'autre s'affairait à concevoir une *vue* simplifiée du jeu. Lorsqu'une fonctionnalité était prête à l'emploi des deux côtés, ils pouvaient alors commencer le débogage de cette fonctionnalité à l'aide de l'émulation du jeu.

La capture d'écran ci-contre correspond à l'ultime version de cet assistant manuel ; nous avions à ce stade terminé de déployer et tester les quatre premiers niveaux de modèle, et l'utilisateur pouvait simuler une partie complète en utilisant tous les types de cartes. La recherche permettait au joueur de trouver des cartes à ajouter à sa main ou au *board* ennemi. Les mécanismes du jeu comme le système de combat ont pu être restitués fidèlement dans notre simulation, et cet assistant *avancé* constituait une réelle émulation du jeu.

Remarque 8 *À la livraison d'un* delogger *fonctionnel nous avons abandonné cette approche qui était fortement* chronophage *puisqu'il fallait à la fois émuler le moteur du jeu de Hearthstone, et son interface.*

3.1.3 *HearthCompanion*

Le *HearthCompanion* est l'aboutissement de ces mois de développement. Cet assistant embarque les trois programmes que nous avons mis au point. Son diagramme de fonctionnement a été vu au chapitre précédent, et l'interface utilisateur est à l'image du diagramme : simple et accessible. Ce *companion*

n'est qu'à un pas d'être un automate. Il nous suffirait d'y intégrer *HearthPlayer* pour obtenir le tant convoité *HearthBot*.

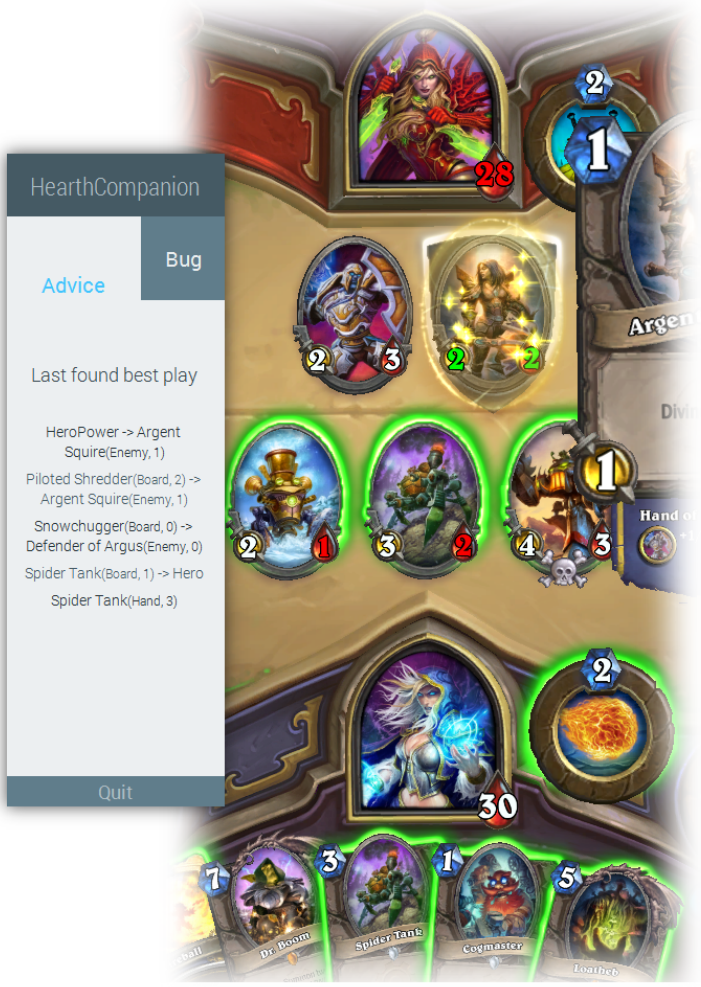

FIGURE 3.4 – **HearthCompanion** en action

Comme illustré dans la figure suivante, l'interface consiste en un *overlay* présentant deux panneaux :

— L'un, *Advice*, permet de demander et d'afficher le meilleur coup pour la situation de jeu courante

— L'autre, *Bug*, est un formulaire de déclaration de bogue.

Lorsqu'un utilisateur n'est pas satisfait d'un coup calculé par notre intelligence artificielle, il peut immédiatement soumettre un rapport de bogue qui sera envoyé en quelques secondes à notre base de données.

Le débogage de cette version fût particulièrement long puisqu'à ce stade le nombre de composantes logicielles avait largement augmenté ; le dispositif de

rapports de bogues s'est alors avéré être l'une des idées les plus puissantes que nous ayons été amené à mettre en pratique.

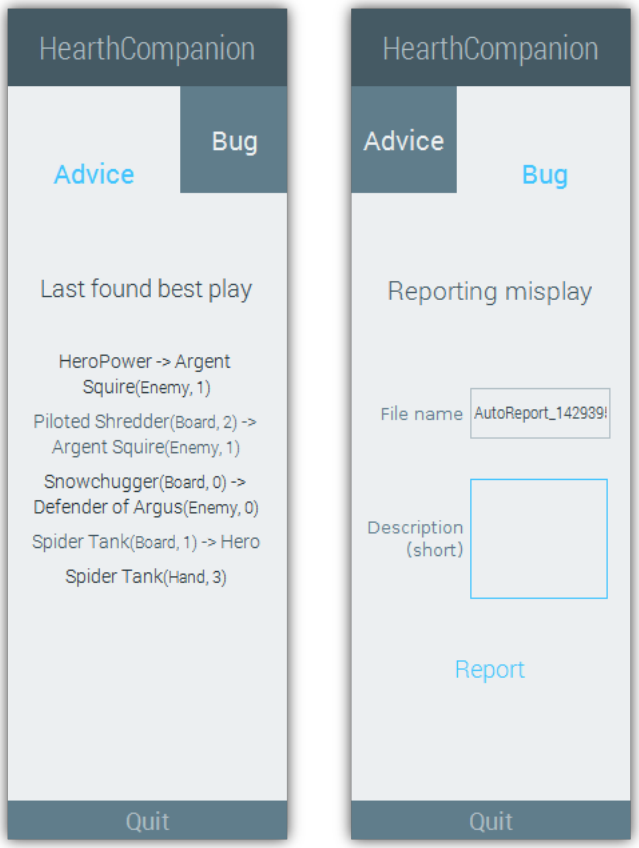

FIGURE 3.5 – Un *overlay*, deux panneaux

Cette version fût la première expérience de développement *LTS*[2] pour les membres de l'équipe. En effet, *HearthCompanion* n'est pas terminé puisque sa simulation ne prend pas en compte toutes les cartes du jeu. Si le texte d'une carte excède le niveau de modèle actuellement implémenté, alors notre intelligence artificielle ne peut jouer cette carte ; il nous fallait, d'une part, nous assurer que la présence de ces cartes lors d'une partie ne gênait pas le fonctionnement du logiciel, et d'autre part que le logiciel "survivrait" à une évolution du niveau de simulation.

Nous avons donc distribué une première version de test, qui s'est améliorée et stabilisée à mesure que des bogues étaient rapportés par les testeurs, ou que de nouvelles cartes étaient prises en charge.

Remarque 9 *Au terme de la conduite de ce projet, le* HearthCompanion *est fonctionnel et stable. Si un utilisateur suit à la lettre les recommandations du* companion*, alors il peut gagner contre le bot d'entraînement de* Blizzard *(l'*Aubergiste*) en mode expert.*

2. Long Term Support

3.2 Aides au débogage

3.2.1 Site web et base de données

Afin d'assurer la persistance des diverses données récupérées au cours de l'avancement du projet et lui donner une meilleure visibilité, nous avons décidé de mettre en place un site web [3]. Nous nous sommes de plus rapidement rendu compte qu'il serait l'outil idéal pour centraliser les informations des utilisateurs dans le cadre d'une collecte de données sur le jeu. Le site web remplit actuellement [4] plusieurs rôles :

— Présenter le projet simplement et donner des liens utiles,
— Maintenir un fil d'actualité autour du projet,
— Tenir une partie réservée aux développeurs pour leur permettre d'observer les divers bogues rapportés par les utilisateurs sur le programme (expliqué dans la partie suivante) et d'écrire des tickets sur l'avancement du projet.

L'essentiel du projet ayant été codé en *Java*, nous avons donc choisi *Java EE* pour mettre en place simplement et dans le même langage un protocole de communication client / serveur.

Plus précisément, le site a été développé en *Java EE* dans une perspective de déploiement sur serveur *Google App Engine*, et ce avec toutes les technologies qui l'entourent, le tout dans un paradigme de conception *Modèle-Vue-Contrôleur* (MVC), tout comme *Hearthbot*. À ce titre nous avons modélisé et développé les diverses parties du site web comme suit :

— **Contrôleur**. L'application serveur est codée sous forme de servlets s'exécutant dynamiquement sur le serveur web et permettant l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, appels de fonctions, etc...
— **Modèle**. Pour nous permettre d'exploiter la puissance des serveurs *App Engine* de *Google*, ces derniers proposent un service de stockage orienté **ORM** (**O**bject **R**elational **M**apping) basé sur des annotations d'objets instanciés par le serveur.
— **Vue**. La page dynamique renvoyée au client est une page classique *HTML, CSS, Javascript* améliorée par le dynamisme apporté par les technologies *JSP* et *JSTL*, permettant notamment l'utilisation des boucles, des conditions, etc... au coeur de la vue.

Il faut souligner que la vue a été implémentée grâce au *framework CSS Bootstrap* et ses bibliothèques *JQuery/Javascript* afin de respecter une certaine charte graphique adaptée à notre *design*.

La conception de ce site web a été l'occasion parfaite de se pencher sur des aspects plus *data* et *marketing* que les autres parties, avec notamment la centralisation d'informations. Elle a nécessité également de prendre du recul par

3. `http://1-dot-ping-group.appspot.com`
4. au moment de la rédaction de ce mémoire

rapport aux besoins du projet, grâce à l'écriture des avancements, ainsi qu'un effort de clarté globale, allant de l'ergonomie du site à la concision des idées exprimées.

3.2.2 Dispositif de *bug report*

Lors d'une partie, il arrive que l'IA propose un "meilleur coup" que le joueur estime mauvais ou hors de propos, et l'utilisateur devait alors annoncer à ses pairs - par mail ou de vive voix - l'existence d'un bogue et les circonstances l'ayant provoqué, ce qui rendait le processus de développement particulièrement lourd et contre-productif.

Le problème majeur pour comprendre d'où vient un bogue en sortie de chaîne est de savoir à quel moment elle a dysfonctionné, et donc chaque partie doit être au courant du bogue pour essayer de trouver l'éventuelle incohérence.

Dans notre cas, il existe plusieurs types de bogues concernant les résultats de l'IA :

— Incohérence dans la formule d'évaluation,

— Mauvaise simulation due à un soucis de modélisation,

— Mauvaise simulation due à un problème de delog de partie *Hearthstone*,

— Problème de copie de simulation lors du *backtrack*.

Chaque partie du programme ayant été développée par une personne différente, il a donc fallu mettre en place un système centralisé où chacun puisse récupérer toutes les informations ayant provoqué ce bogue et le recréer localement. Pour cela nous avons décidé de pousser plus loin le concept posé par la création du langage standard imaginé pour décrire des parties *Hearthstone* - HSS et HSC -, permettant ainsi de sauvegarder des cas limites / dysfonctionnements du logiciel et de les récupérer.

En effet, chaque état d'une partie donnée étant représenté de manière exacte par un HSC, il était alors facile de reproduire un cas de bogue une fois le HSC rendu public. Les serveurs hébergeant le site étaient alors tout désignés pour stocker une base de données de configuration de parties (HSC) et les afficher pour centraliser l'information.

Nous avons mis au point un protocole de communication client / serveur assez léger puisqu'il ne nécessite qu'un envoi de requête HTTP formaté au serveur, qui traite le bogue et l'instancie en base de données, avec un identifiant unique permettant ainsi à l'utilisateur et au développeur de parler rapidement du même bogue.

La base de données a été pensée et créée de telle sorte qu'il est possible de trier les bogues par catégorie, par date et par utilisateur.

Le tout ayant été implémenté rapidement, nous avons pu disposer d'une aide au débogage puissante et à fort potentiel communautaire.

3.2.3 *Replay* de parties

Le langage standard que nous avons imaginé pour décrire une partie de *Hearthstone* est intelligible mais reste fastidieux à lire pour un humain. Dans l'optique de rendre le processus de débogage plus simple d'une part et d'améliorer l'ergonomie de notre *Overlay* d'autre part, nous avons décidé de développer un logiciel qui serait capable de lire un HSC et d'afficher son contenu de manière schématique et visuelle pour l'utilisateur. Nous pouvons alors voir aisément l'état du jeu quand le bogue s'est produit.

On pourrait aussi lui donner un HSG en entrée, ce qui nous permettrait de revoir le déroulement de la partie entière et de disposer ainsi d'un véritable "historique animé" de celle-ci. Afin de faciliter le visionnage des parties, on pourrait alors rajouter des fonctionnalités comme, mettre en pause, revenir en arrière et avance rapide afin de rendre l'expérience utilisateur la plus agréable possible.

Chapitre 4

Limites, perspectives

Tout au long des phases de développement et de débogage, nous avons pu appréhender plus précisément les limites de notre stratégie. Certaines de ces limites sont inhérentes au projet, ce sont ses faiblesses. Certaines limites sont extérieure au projet, c'est le cas par exemple du phénomène d'explosion combinatoire, ou des limites légales.

4.1 Limites

4.1.1 Faiblesses du projet

Notre travail se basant sur un jeu dont on ne dispose pas du code source, les principales faiblesses du projet sont inhérentes à cette contrainte importante. Là où *Blizzard* met à disposition une interface de programmation dans d'autres de ces jeux, ce n'est pas le cas de *Hearthstone* et cela nous a contraint à recoder autant que possible les diverses mécaniques du jeu. Ce décalage, inévitable, entre la simulation et le jeu rend dans certaines situation le logiciel instable. Ce fait est d'autant plus exacerbé que *Hearthstone* est un jeu en pleine évolution, chacune des mises à jour importantes - tous les 3 mois environ - apportant de nouvelles cartes et nouvelles mécaniques à devoir implémenter dans notre projet. Par la taille de celui-ci, il était difficile pour nous de pouvoir garantir une implémentation du projet permettant l'ajout de nouveaux éléments dont nous ne pouvons anticiper ni leur arrivée ni leur contenu.

Une autre faiblesse importante est notre façon de récupérer les informations d'une partie. Actuellement, la lecture des journaux de débogage est la solution la plus optimale en terme de performances et de précision. Néanmoins, si les utilitaires ou automates de jeu continuent de se développer autour de cette méthode, elle est certainement vouée à disparaître dans une mise à jour ultérieure ; le travail abandonné sur la reconnaissance d'images pourrait alors servir de solide base.

Enfin, une dernière faiblesse de notre projet pourrait être sa fonction de *scoring*, et plus précisément ses variables d'ajustement (a, b, c). Ces coefficients, censés rendre la fonction plus souple, sont actuellement trouvés de manière empirique. Cependant, une telle façon de procéder pour trouver ces coefficients ne peut être satisfaisant à long terme ; une solution plus élégante sera donc évoquée dans les perspectives d'évolution du projet.

4.1.2 Explosion combinatoire vs temps de jeu borné

« On ne sait pas, peut-être que dans le pire des cas, c'est un million de nœuds. C'est rien en machine ! »

– E. Bourreau

Le problème

Un tour dans une partie en joueur contre joueur a une durée fixée à 90 secondes. Ce temps de jeu borné est le principal inconvénient auquel on est confronté lorsqu'on essaie de calculer le meilleur coup. L'algorithme essayant toutes les combinaisons de coups possibles à effectuer à chaque début de tour, il peut sembler naturel qu'à partir d'un nombre de cartes élevé (sur le *board* ou en main), il y aura irrémédiablement une explosion combinatoire dans le nombre de coups à envisager.

Il est difficile de calculer le nombre exact de coups possibles dans tous les cas, tant ceux-ci sont variés et inhérents aux diverses mécaniques du jeu. Dans le but de donner malgré tout un ordre de grandeur de l'explosion combinatoire à laquelle nous sommes confrontés, nous ne nous intéresserons qu'à l'étude de quelques situations que nous considérerons comme problématiques, puis une situation courante avec de forte chance d'être rencontrée. Ainsi, sans connaître la valeur exacte correspondant au nombre de coup dans le pire des cas, ces situations pourront nous donner une borne inférieure de celui-ci.

Modélisation

Pour la modélisation, nous utiliserons les notations suivantes :

— m le nombre de serviteurs *au total* sur le *board*,

— m_{allie} le nombre de serviteurs sur le *board* allié,

— m_{ennemi} le nombre de serviteurs sur le *board* ennemi,

— n le nombre de cartes en main,

— n_{sort} le nombre de cartes de type sort en main,

— $n_{serviteur}$ le nombre de cartes de type serviteur en main ;

Remarque 10 *On remarque que dans cette modélisation, on ne suppose pas qu'il y ait d'arme dans les cartes en main.*

Première situation Dans la première situation, nous ferons les suppositions suivantes :

— Sur le *board* ennemi, toutes les créatures ont *Provocation*. Les serviteurs alliés ne peuvent donc pas attaquer le héros adverse,

— Les créatures sur le *board* ennemi ne peuvent pas être tuées ce tour-ci, même en concentrant toute sa force de frappe sur une unique cible,

— On dispose de suffisamment de mana, et de cartes suffisamment peu coûteuses pour pouvoir jouer toute sa main ce tour-ci,

— N'importe laquelle de nos créatures mourra après avoir attaqué n'importe quelle autre créature ennemie,

— La *classe* du héros allié est *Mage*[1]

— On ne possède que des serviteurs dans la main, dont aucun ne dispose de *Charge*, mais disposant tous d'un *Cri de Guerre* utilisable sur tous les personnages

— Au début du tour, on a : $n = n_{serviteur} = 7, m_{allie} = 7, m_{ennemi} = 7$.

Bien que très improbable, il est important de noter qu'une telle situation est, en pratique, possible.

Le *Pouvoir du Héros* peut être utilisé sur tous les personnages, c'est-à-dire $m+2 = 16$ cibles en comptant les deux héros. Chaque fois qu'un de nos serviteurs doit attaquer, il y a $m_{ennemi} = 7$ possibilités. De plus, à chaque fois qu'on pose une nouvelle créature sur le *board*, il y a au plus 7 possibilités d'emplacement. À chaque fois qu'on pose un serviteur, son *Cri de Guerre* est utilisable sur tous les personnages, hormis lui, c'est-à-dire $m + 1 = 15$ cibles. Dans cette situation, il est important de remarquer qu'on n'est pas capable de jouer nos serviteurs tels quels, puisque le *board* est complet. Il faut donc toujours que le nombre de cartes posées sur le terrain soit toujours inférieur ou égal au nombre de fois qu'un de nos serviteurs a attaqué un serviteur adverse. Le nombre de permutations entre une action de *trade* et de pose de serviteur correspond donc au septième nombre de Catalan.

Le nombre de possibilités est donc au plus de : $429 \times 16 \times (7 \times 7 \times 15)^7 = 8.10^{23}$.

Deuxième situation Dans la deuxième situation, tout aussi problématique que la première, on s'intéresse plus au problème posé par les nombreuses cibles possibles d'un sort. On fera donc les suppositions suivantes :

— Sur le *board* ennemi, aucune créature n'a *Provocation*,

— Les créatures sur le *board* ennemi ne peuvent pas être tuées ce tour-ci, même en concentrant toute sa force de frappe sur une unique cible,

— On dispose de suffisamment de mana, et de cartes suffisamment peu coûteuses pour pouvoir jouer toute sa main ce tour-ci,

— Aucune de nos créatures ne peut mourir pendant ce tour,

— La *classe* du héros allié est *Mage*

— On ne possède que des sorts dans la main, tous nécessitants une cible.

— Au début du tour, on a : $n = n_{sort} = 10, m_{allie} = 7, m_{ennemi} = 7$.

Chaque serviteur possède $m_{ennemi} + 1 = 8$ cibles possibles, en comptant le Héros adverse. De la même façon, chaque sort dispose de $m + 2 = 16$ cibles possibles. On pourra considérer pour ce calcul que le *Pouvoir du Héros* se comporte comme un sort, on en dispose donc "fictivement" de 11 au total. De plus, les différentes permutations possibles correspondent à la factorielle nombre de coups effectués, à savoir 18!.

Le nombre de possibilités est donc de : $18! \times 8^7 \times 16^{11} = 2.10^{35}$.

1. Le pouvoir du Héro du Mage permet au joueur d'infliger 1 point de dégât à un personnage.

Troisième situation Cette fois-ci, analysons une situation bien plus réaliste en pratique. Dans celle-ci, nous avons :

— Sur le *board* ennemi, une créature a *Provocation*,

— Les créatures sur le *board* ennemi ne peuvent pas être tuées ce tour-ci, même en concentrant toute sa force de frappe sur une unique cible,

— On dispose de suffisamment de mana, et de cartes suffisamment peu coûteuses pour pouvoir jouer toute sa main ce tour-ci,

— Aucune de nos créatures ne peut mourir pendant ce tour,

— La *classe* du héros allié est *Mage*

— On ne possède que des sorts dans la main, tous nécessitants une cible.

— Au début du tour, on a : $n = n_{sort} = 4, m_{allie} = 4, m_{ennemi} = 4$.

Le calcul est identique que celui précédemment, ceci étant que les créatures du *board* allié sont obligées de taper la cible avec *Provocation* et qu'on ne dispose que de 10 cibles en tout pour les sorts. Ceci nous donne un nombre de coups possible de : $8! \times 10^5 = 4.10^9$.

Conséquence de ces résultats

Ces situations montrent bien qu'il peut y avoir des cas où envisager de calculer la meilleure solution est, en pratique, impossible. À titre indicatif, dans la deuxième situation, en supposant que simuler un coup correspond à une seule instruction en machine (ce qui est une minimisation importante du problème), il faudrait disposer d'un processeur cadencé à $2.10^{33} Hz$ pour être en mesure de réaliser un tel calcul dans notre temps imparti de 90 secondes, ne laissant alors même pas le temps au joueur de jouer la solution.

Critiques des situations présentées

Le premier point critiquable des deux premières situations présentées est l'aspect extrême de celles-ci. Malgré tout, il faut garder à l'esprit que ce sont des situations tout à fait réalisables *en théorie*. D'ailleurs, la troisième situation, qui elle est beaucoup plus réaliste, donne un nombre de coups possibles relativement calculable à l'échelle d'une machine.

Ensuite, il est tout de même important de noter que certaines de nos suppositions faites peuvent être considérées comme extrêmement contraignantes combinatoirement ; notamment celle interdisant à une créature de pouvoir mourir ce tour-ci. En pratique, il est très rare avec un *board* plein et des sorts en main de ne pouvoir tuer aucun des serviteurs adverses.

En diminuant alors d'au moins un le nombre de serviteur adverse, la sous-arborescence des coups possibles à partir de ces nœuds devient beaucoup plus petite, et c'est un cas qui n'a pas été envisagé lors des calculs précédents.

On peut envisager de nombreuses heuristiques pour faire chuter de manière efficace le nombre de coups possibles. Par exemple, si on restreint les sorts de dégâts aux cibles ennemis et, de la même façon, les sorts de soins aux cibles alliées, on peut diviser par deux le nombre de cibles potentielles à chaque branchement de sorts ciblés.

4.1.3 Légalité : PING vs Blizzard

La notion de légalité pour un projet comme celui-ci est difficile à aborder. Le projet comme présenté et proposé en premier lieu ne respecte pas les conditions d'utilisation imposées par *Blizzard*. Mais de part le développement en méthode agile, toutes les versions du projet peuvent se voir comme un assistant de jeu, et alors *Blizzard* est beaucoup plus laxiste. *Tout ce qui peut se faire avec un crayon et un papier est toléré* confirme un développeur de chez *Blizzard*. Dans l'absolu, un backtrack est faisable à la main, et finalement ce n'est que l'intégration de la souris qui peut réellement poser des problèmes de légalité.

Notre tentative d'obtenir une autorisation gracieuse d'expérimenter avec les automates a échoué. Le service *anti-triche* de *Blizzard* n'a pas souhaité donner suite à notre demande[2].

Dans ces conditions, on pourrait imaginer de créer un projet clone, anonyme, duquel PING serait officiellement détaché, qui intégrerait alors la partie *automatisation du calcul et du jeu des coups*.

4.2 *Hearthstone Assistance Suite*

« Une oeuvre ne saurait être terminée. Seulement abandonnée. »

– L. Da Vinci

Le projet pouvait initialement s'étendre à l'élaboration d'un algorithme de création de *decks* compétitifs. Il s'est avéré que la main d'œuvre disponible et le temps imparti étaient incompatibles avec la réalisation de cet objectif. Aussi avons-nous tout de même pu exprimer quelques considérations théoriques sur ce sujet.

Mais les perspectives ne s'arrêtent pas à cette ébauche d'algorithme. Le portage des calculs sur des serveurs dédiés, le développement d'une application *companion* pour mobile ou encore la mise au point d'un cycle d'apprentissage pour notre intelligence artificielle ne sont que des exemples de ce qu'il serait possible de réaliser avec comme base notre projet.

L'ensemble des outils et techniques que nous avons imaginés pour *Hearthstone* pourraient être compilés dans une suite logicielle : HEARTHSTONE ASSISTANCE SUITE. Nous donnons à titre anecdotique une ébauche de présentation de cette suite logicielle en page d'accueil du site web.

Enfin, l'abandon de notre œuvre au domaine publique, couplé avec une bonne communication utilisant comme support notre site web, permettrait de faire voir le jour à cette suite logicielle d'assistance pour *Hearthstone*.

4.2.1 Kernelization des *decks*

Motivations et première idée

En observant les joueurs professionnels de *Hearthstone*, nous avons constaté qu'ils étaient, à partir de seulement une ou deux cartes du jeu de l'adversaire,

2. Une copie de notre demande faite à *Blizzard* se trouve en Annexe B

capables de connaître pratiquement entièrement le jeu de celui-ci. Ceci est dû en partie au fait qu'il n'existe à chaque instant qu'un nombre limité de *decks* jouables de manière compétitive à haut niveau. Nous nous sommes alors demandés comment il pourrait être possible, à notre échelle, de modéliser cette reconnaissance de *decks*.

Une première solution, et qui a été en partie réalisée pendant le projet, est de récupérer via les principaux sites communautaires autour d'*Hearthstone* la liste des *decks* les plus joués à l'heure actuelle. Ainsi, si pour chaque classe on récupère les cents *decks* les plus compétitifs, alors à mesure que l'adversaire joue ses cartes, on peut facilement trouver quel *deck* de notre liste se rapproche le plus de celui contre qui on est entrain de jouer. Deux cas peuvent alors se passer :
Si notre adversaire ne joue pas un *deck* de notre base de données, alors puisque celle-ci est composée des jeux les plus compétitifs, il y a de grandes chances pour que ce *deck* soit moins bien construit (peu de synergie, par exemple) et on disposera malgré tout d'un pire des cas intéressant lors de l'inférence du coup adverse. À l'inverse, si jamais l'adversaire joue un des *decks* de la liste, on sera en mesure de prédire de manière plus efficace les coups possibles à chaque tour. Par exemple, bien que peu probable, si on devine que l'adversaire joue un deck sans carte de coût 5, alors il sera inutile à partir du tour 4 d'envisager qu'il puisse jouer un serviteur ou un sort de ce coût, ce qui est un gain important sur le nombre de situations à simuler.

Un moyen efficace de modéliser un *deck*

Cependant, cette solution simple est paradoxalement difficilement applicable dans les rangs élevés (10 - 25), car il s'agit souvent de *decks* faits par un joueur qui manque d'expérience et/ou de "cartes clefs". On doit donc essayer de faire une reconnaissance de *decks* plus flexible, d'où notre seconde solution : la *kernalization de decks*. Tout comme en informatique théorique, on va tenter de réduire la taille de notre problème (ici, le *deck*) en une instance plus petite (que nous appellerons le *cœur du deck*).

Il est facile de remarquer, aux rangs élevés, qu'il existe de nombreuses cartes servant simplement à finir de compléter le *deck* (typiquement les cartes qui correspondent au Niveau 0 de notre modèle incrémental) ; dès lors, il paraît naturel d'éliminer ces cartes -que l'on appellera *classiques*- du cœur du *deck*, puisque trop facilement remplaçables. Mais on peut aussi imaginer le raisonnement inverse ; il existe en effet des cartes indispensables dans certaines classes (par exemple les premières que l'on obtient au début de chaque classe) et qui, puisque jouées dans tous les *decks*, ne permettent donc pas de différencier les uns des autres.

L'idée de ce seuillage de cartes est donc *d'écarter* toutes les cartes superflues ou indispensables afin de pouvoir *extraire* que celles qui caractérisent un *deck*. On obtient alors un ensemble de cartes à forte synergie entre elles.

Exemple 6 *Le cœur d'un deck d'un chasseur aggro :*
{Loup des bois[3]*, Hyène charognarde*[4]*, Lâcher les chiens*[5]*, Jongleur de cou-*

3. Serviteur 1/1 (coût 1) (Bête) : Vos autres Bêtes ont +1 ATQ.
4. Serviteur 2/2 (coût 2) (Bête) : Chaque fois qu'une bête alliée meurt, gagne +2/+1.
5. Sort (coût 3) : Invoque un chien 1/1 (Bête) avec Charge pour chaque serviteur adverse.

teaux[6]*, Roi des bêtes*[7] *}*.

Ainsi, à partir d'un cœur, on est capable de construire de nombreux *decks* en le complétant des cartes indispensables, puis des classiques. Tous les *decks* de cet ensemble pourront ainsi raisonnablement être considérés comme *équivalents*. La *kernalization* est donc aussi un moyen efficace de stocker plusieurs *decks* en un minimum de place.

Remarque 11 *On remarquera que plus un deck est compétitif, plus la taille de son cœur augmente.*

Extraction du *cœur*

La difficulté est donc, étant donné un ensemble de *decks*, d'*extraire* le cœur de celui-ci. Une heuristique pour ce problème est de chercher les cartes appartenant à tous les *decks* de l'ensemble. Ainsi, si on suppose que l'ensemble ne contient que des *decks* équivalents, il paraît naturel de considérer que l'intersection de ces *decks* correspondra, au mieux, au cœur de l'ensemble, et au pire un sous ensemble du cœur.

En revanche, si on veut essayer d'affiner l'extraction, et particulièrement dans le cas où il peut y avoir dans l'ensemble des *decks* un peu différents, il devient rapidement plus compliqué de pouvoir calculer le cœur.

Modélisation du problème Pour modéliser le problème, on notera $C = \{c_1, c_2, \ldots, c_n\}$ un ensemble de cartes, $D = \{D_1, D_2, \ldots, D_p\}$ un ensemble de *decks*, où $\forall i = 1, \ldots, p, D_i$ est un sous ensemble de C de taille 30, et tel que $C = \bigcap_{i=1}^{i=p} D_i$. De plus, notons $E = \{(c_i, D_j) : c_i \in D_j\}$.
On considère $G = (C \cup D, E)$ le graphe biparti correspondant à l'appartenance des éléments de C dans D.

On s'intéresse donc au problème d'optimisation suivant :
Données : Un graphe biparti $G = (C \cup D, E)$ défini plus tôt, un entier l strictement positif.
Problème : Trouver $K_{i,j}$ sous graphe biparti complet tel que $1 < i \leq l$ et j maximum.

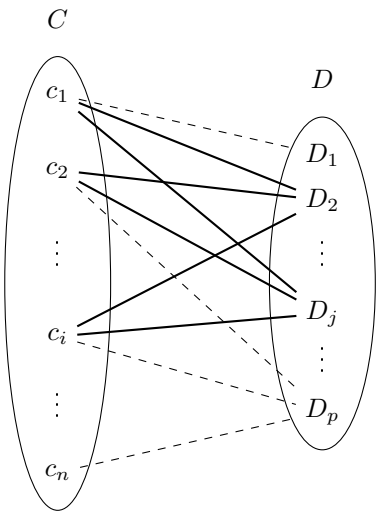

Fig. 4.1 – En gras un exemple de plus grand sous biparti complet

Chercher le *cœur* de l'ensemble de *deck* D revient alors à trouver une solution de ce problème, avec $l = 30$.

6. Serviteur 3/2 (coût 2) : Inflige 1 point de dégâts à un adversaire aléatoire après que vous avez invoqué un serviteur.
7. Serviteur 2/5 (coût 5) :Provocation. Cri de guerre : gagne +1 ATQ pour chacune de vos autres Bêtes.

Malheureusement, il n'a pas été démontré pendant le projet que ce problème est NP-Complet. Afin de donner l'intuition au lecteur que c'est bien le cas, voici un résultat montré par Garey et Johnson[8] :
Données : Un graphe biparti $G = (V, E)$, un entier positif $k \leq |V|$.
Question : Existe t-il un sous-graphe biparti équilibré de G de taille k ?
est un problème NP-Complet.

Algorithmes génétiques et *decks* innovants

Cette façon de représenter les *decks* de manière minimale offre de nombreuses autres possibilités. Une de celle-ci est de pouvoir générer de nouveaux *decks* en se basant sur le principe des algorithmes génétiques.

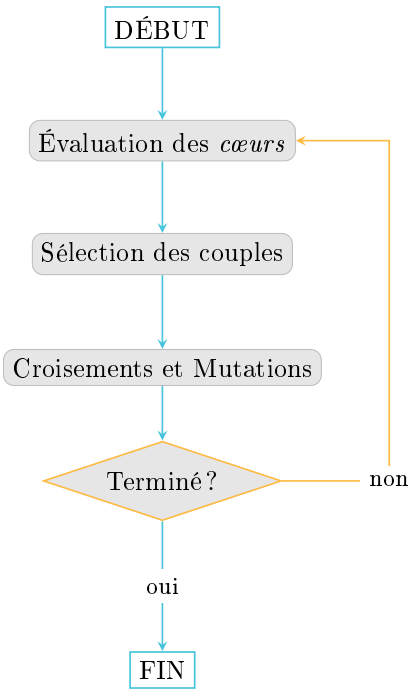

Fig. 4.2 – Création de nouveaux *cœurs* grâce aux algorithmes génétiques

On part d'une population de base de *cœurs*, récupérés par exemple de manière empirique, à laquelle on est capable de donner une valeur à chaque *cœur* à l'aide d'une fonction d'évaluation. Une fonction d'évaluation efficace pouvant être, par exemple, le taux de victoire moyen fasse aux autres *cœur* de la population. Cela nous permet alors de ne garder que les *cœurs* les plus compétitifs.

On effectue alors des croisements et des mutations sur des couples de *cœurs* choisis aléatoirement. Un croisement correspond à créer de nouveaux *cœurs*, des fils, issus des différents éléments du couple de *cœurs*. Une mutation est un phénomène arrivant aléatoirement et qui va remplacer une carte d'un *cœur* fils par une carte au hasard. On dispose alors d'une nouvelle population, à laquelle on peut répéter l'opération autant de fois qu'on souhaite.

À la fin du processus, si celui-ci dure assez longtemps, on aura réussi à créer de nouveaux *cœurs* – et donc de nouveaux *decks* – , avec de nouvelles synergies entre les cartes.

La *kernalization* de *decks* est donc un outil à la fois passionnant à étudier théoriquement, mais aussi avec de nombreuses applications directes, allant de l'inférence à la création de nouveaux *decks*, en passant par le gain en espace de stockage. Mais même au delà de l'univers de *Hearthstone*, la question de savoir extraire d'une collection d'ensemble un ensemble qui le représente le mieux semble être un sujet intéressant.

8. Nous ne sommes pas en mesure de donner la référence de ce résultat, puisqu'il semble qu'il s'agisse d'un papier non publié.

4.2.2 Passage à l'échelle et application mobile

Bien que le projet soit arrivé à une version stable et fonctionnelle, il est toujours bon de se demander si il ne fonctionnerait pas mieux par exemple dans un autre langage, s'il ne pourrait pas être plus ergonomique, moins gourmand en ressources système, etc...

L'avènement du site web et de sa base de données a ouvert tout un champ de perspectives quant au futur du projet. En effet, ce dernier a été pensé pour répondre avant tout à un besoin de programmeur, avec une panoplie d'outils consacrés au développement. Il serait intéressant d'ajouter au site une dimension communautaire, présente sur nombre de sites web autour de *Hearthstone*. Cet aspect pourrait être mis en place grâce à des modules sociaux et des fonctionnalités client / serveur rendant le tout beaucoup plus axé sur le joueur. Voici quelques idées pensées dans cette optique :

— La mise en place d'un **espace membre**, où chaque joueur pourrait se connecter et mettre en ligne ses parties jouées ou ses *decks*, par exemple. Le client *HearthCompanion* étant alors une application client communiquant au serveur l'ensemble des données de jeu sur le joueur, lui permettant alors de voir ses statistiques, ses points forts ou faibles, son ratio de victoire sur une période, etc...

— Dans le même ordre d'idées, un **espace de partage** pourrait contenir des guides, des aides pour les nouveaux joueurs, un forum, ou des actualités autour du jeu.

— Avec une base de données concernant les informations relatives à tous les joueurs utilisant le logiciel, une grosse partie ***data mining*** ferait alors surface.

Cet aspect *data* permettrait notamment des calculs statistiques utiles à la fonction d'évaluation de l'intelligence artificielle. Pour rappel, cette dernière prend en paramètre des *coefficient d'ajustements* empiriques. Cela permettrait également d'orienter la recherche lors du *backtrack*. En effet il est facile d'imaginer qu'avec une grande base de données de parties jouées, avec des annotations sur les *decks* gagnants et toutes les étapes de jeu, il est possible de trouver des liens de causalité entre les cartes jouées et la victoire. En reliant ce principe à la *kernelization de* decks vue précédemment, nous nous doterions d'un outil assez puissant pour avoir un ratio de victoire élevé. Par exemple, il serait possible de détecter que le *Mage* adverse joue un *deck méca aggro*[9], et que, dans cette configuration donnée, jouer telle carte mène la victoire dans 75% des cas.

Une des idées les plus intéressante serait de porter l'ensemble du code du simulateur et de l'intelligence artificielle sur des serveurs web dédiés au calcul. Cela permettrait dans un premiers temps d'accélérer drastiquement le temps de réponse du logiciel et d'alléger la charge CPU de l'ordinateur du joueur. Dans un second temps, cela pourrait rendre l'ensemble du programme beaucoup plus ergonomique pour une utilisation sans la partie *bot* :

9. *Aggro* est un terme désignant une façon de jouer consistant à être le plus aggressif possible

Imaginons que l'actuel *HearthCompanion* devienne une application distribuée. Le joueur n'aurait que le délogger - un processus léger invisible - a lancer au démarrage de *Hearthstone*, ne faisant que délogger les informations de jeu et les communiquer directement au serveur sans interaction directe avec le joueur. Imaginons également que la partie *vue* du logiciel soit une application mobile. Une fois le calcul du *backtrack* terminé sur le serveur, le joueur aurait alors une notification sur le meilleur coup trouvé, avec si besoin un schéma d'explication sur la séquence de coups optimale à jouer.

Cette application mobile pourrait également permettre au joueur d'avoir toutes les informations du site le concernant directement sur mobile.

4.2.3 Machine learning et *méta*

Comme cela a été évoqué dans la section 4.1.1 (Faiblesses du projet), le jeu est en constante évolution. Les versions du jeu se succèdent, faisant évoluer ce que l'on appelle la *méta*, qui désigne l'ensemble des stratégies, des cartes et des *decks* dont le fonctionnement et l'efficacité ont été éprouvés dans le *ladder* de la version courante. Le développement n'étant pas terminé nous ne pouvions pas encore expérimenter sur ce sujet ; cela ne nous a pas empêché d'imaginer des solutions à nos heures perdues, nous avions au passage l'occasion de faire des recherches sur des sujets qui étaient encore flous pour nous. Les considérations suivantes supposent que notre simulation fonctionne rigoureusement à l'identique du jeu, et que notre dispositif de sauvegarde de partie est parfait. [10]

En section 2.3 nous vous présentions la fonction d'évaluation que nous avons utilisé. Vous avez pu remarquer l'existence de coefficients arbitrairement fixés (a, b, c). Il va sans dire que ces coefficients fixés sont incompatibles avec l'évolution constante de la *méta*. Imaginons que, pour la version i du jeu, les coefficients (a_i, b_i, c_i) conduisent à une évaluation correcte des situations, rien ne nous garantit que l'évaluation restera correcte lors du passage à la version $i + 1$.

Pour palier à cela nous avons pensé à une solution basée sur le *machine learning*, et les différents outils que nous avons pu produire. Nous disposons d'une simulation du jeu, d'une intelligence artificielle capable de calculer de bons coups à jouer étant donnée une situation, et d'un programme *relogger* produisant des fichiers *HSG* retraçant des parties jouées. Cette solution, à notre grand regret, n'a pas pu être déployée, et nous ne proposons qu'un début de théorie.

Considérons une partie sauvegardée au format *HSG*. Elle contient la succession de coups joués par le joueur à chaque tour. On peut donc extraire de cette partie les situations en chaque début de tour, et comparer ce qu'aurait fait notre intelligence artificielle avec ce qu'a effectivement joué l'être humain. Supposons maintenant que l'on puisse ajuster les coefficients (a, b, c) de manière à ce que notre intelligence artificielle trouve la même suite de coup que le joueur. Dans ces conditions l'intelligence artificielle aurait appris du joueur à évaluer de manière plus pertinente une situation.

La démarche qui vient d'être expliquée peut être aisément automatisée ; il suffit pour cela de concevoir un programme qui, étant donné n parties gagnées par un excellent joueur, calcule les coefficients permettant de jouer exactement

10. Il n'y a pas de perte ou de déformation de l'information

comme l'être humain sur ces parties. On peut alors imaginer un processus d'apprentissage basé sur le *data mining*. Il nous suffirait pour cela de récupérer sur nos serveurs le fichier *.HSG* produit à la fin de chaque partie suivie par le *HearthCompanion*. Nous disposerions alors d'une vaste base de données de parties de laquelle l'intelligence artificielle pourrait tirer des leçons.

Nous n'avons pas entrepris l'implémentation de cette solution par manque de temps, d'autant que trouver un moyen efficace d'ajuster les coefficients parait être une tâche laborieuse et basée sur l'empirisme.

4.2.4 Vers l'Open Source

Plusieurs motivations nous poussent à rendre le projet libre et accessible à tous. Il nous a demandé énormément de temps et d'efforts, et abandonner, sous prétexte d'une fin d'année scolaire, un travail d'une telle ampleur serait une erreur. Bien au contraire, nous aimons considérer le projet comme ayant un futur, même si nous ne devions plus être amenés à le développer activement.

Il y a une réelle demande de ce genre de programmes au sein de la communauté des joueurs de *Hearthstone*. *Replay* des parties, probabilité de piocher une certaine carte, historique complet de la partie... Il existe aussi une demande pour des programmes, payants, qui jouent seuls et permettent d'obtenir de l'or sans avoir à faire les quêtes journalières soi-même, ou bien, avec le *deck* adapté, à grimper dans le classement en laissant tourner le programme une nuit durant.

Notamment avec le système de *bug-report*, mais aussi avec la documentation et notre investissement pour les mois et années à venir, le projet est prêt à accueillir une communauté de joueurs développeurs pour améliorer et compléter *HearthCompanion*. Notre code peut aussi servir de base à d'autres projets, que ce soit la partie concernant la simulation du jeu, la lecture des journaux de débogage ou la gestion de la souris. C'est aussi le cas des scripts de téléchargements automatiques des images des cartes et des meilleurs *decks*.

Concernant la licence sous laquelle distribuer le projet, il nous a semblé judicieux d'utiliser la licence *X11*[11], qui en plus de nous protéger d'éventuelles poursuites oblige quiconque utilise le code de mettre les noms des auteurs dans le copyright (ce que la licence BSD ne permet pas).

11. disponible en Annexe C

Annexe A

Énumération des Types

Voici l'énumération des différents types existants, rangés par catégorie.

A.1 Effets

Altérer, Contrôler, Gain d'Armure, Blesser, Désarmer, Défausser,
Piocher, Équiper, Geler, Soigner, Tuer, Placer la Carte, Révoquer,
Silence, Restaurer, Invoquer, Transformer, Fixer la vie.

A.2 Portées

Ciblé, Global,
Cible Aléatoire,
Cibles Aléatoires ;

A.3 Ensembles de Cibles

Soi-même, Adjacents,
Personnages, Personnages sauf soi,
Alliés, Alliés sauf soi, Ennemis,
Serviteurs, Serviteurs sauf soi, Serviteurs alliés, Serviteurs Alliés sauf soi, Serviteurs Ennemis,
Héros, Héros Adverse, Héros Allié
Secrets, Secrets Alliés, Secrets Adverses ;

A.4 Familles

Bête, Méca, Pirate, Murloc, Démon, Recrue d'Argent, Dragon
Un_Mana, Deux_Mana, Quatre_Mana, Légendaire, Totem ;

Annexe B

Courriel adressé à *Blizzard*

Voici une copie courriel que nous avons envoyé au service anti-triche de *Blizzard*.

> Greetings,
>
> We [six people] are currently studying for a Master's Degree in Theoretical Computer Science at the Faculty of Science (University Of Montpellier, France).
>
> A requirement this semester is the completion of a team project. The one we chose consists in elaborating an artificial intelligence able to guess the most efficient way of playing a *Hearthstone* turn using tree exploration algorithms.
>
> As the use of such an AI is prohibited by the game's Terms of Use, we were wondering if you could give us permission (or conditions) to conduct in-game trials with the software we are working on.
>
> We hope - for the sake of Science (and our diplomas) - we will come to an agreement, and we are looking forward to your answer.
>
> The PING team.

Annexe C

Licence *X11*

« Copyright © 05/25/2015, PING Team

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions :

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The Software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders X be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the Software.

Except as contained in this notice, the name of the PING Team shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from the PING Team. »