École Nationale Supérieure des Sciences Appliquées et de Technologie
Logiciel et Systèmes Informatiques — Deuxième année

Intelligence Artificielle

Bataille fantastique : création de l'IA

Antoine Nourry

Javid Mougamadou

Ulrich Audouard-Sadou

17 mai 2015

Chargé de cours :

Gwénolé Lecorvé

Table des matières

1 Introduction

Ce projet *Phantasticas Bellum* reprend le code de Thomas Gasquez et Pierre-Quentin Warlot, développé dans le cadre du projet Bataille Fantastique, et consiste à mettre en oeuvre un joueur artificiel (doté d'une intelligence) qui puissent affronter un autre joueur humain ou artificiel et éventuellement remporter la partie. L'objectif principal de ce projet est de créer une IA capable de battre l'*IA Aléatoire* (fourni par notre encadrant) et également de battre les IAs des autres groupes par le biais d'un tournoi.

Nous allons voir dans un premier temps l'algorithme utilisé (ici Alpha-Beta) pour l'élaboration de notre intelligence artificielle. Puis dans un deuxième temps, nous allons voir les différents paramètres et procédures utilisés dans le but d'améliorer la profondeur de recherche et l'élagage des branches. Enfin, nous verrons les heuristiques utilisées permettant d'évaluer ou de mesurer les différents coups possibles lors d'une partie.

2 L'IA : CollectifIADynamos

2.1 L'algorithme

2.1.1 Alpha-Beta

Pour developper notre IA, nous avons decidé d'évoluer sur une stratégie basée sur l'algorithme Alpha-bêta. Cet algorithme aide à la prise de décision, choisir le coup le plus intérressant dans le but de gagner la partie. Pour adapter notre problème à l'algorithme, nous avons eu besoin d'une fonction heuristique pour decider quel coup est meilleur qu'un autre. Notre Alpha-bêta a été developpé en deux phases.

- Algo Alpha-bêta premier : pour choisir le meilleur premier coup.
- Algo Alpha-bêta : pour le déroulement du veritable algrithme Alpha-bêta.

La procedure Alpha-Beta premier (dont nous parlerons plus bas) renvoi le meilleur coup à jouer par notre IA dans la partie. Elle va parcourir l'arbre à une profondeur maximale donnée. Pour cela, elle va faire appel à la procédure Alpha-Beta classique qui va calculer la valeur des utilités au niveau des feuilles. Voici le pseudo-code de notre algorithme Alpha-Beta :

procédure *AlphaBeta*(*a* :entier,*b* :entier,*partie* :Partie,*profondeur* :entier,*joueurIA* :booléen) :

var entier $i \leftarrow 0$;
var Coup[] $ListdesCoups$;
Debut
Si $profondeur \leq 0$ **Alors**
$util \leftarrow GetUtilite(partie)$;
Retourner util ;
Sinon
$ListdesCoups \leftarrow GetTousCoups()$;
$ListdesCoups \leftarrow Filtrage(ListdesCoups)$;
$ListdesCoups \leftarrow MeilleurCoups(ListdesCoups)$;
Si $joueurIA = true$ **Alors**
Tant que $a < b$ **et** $i < ListdesCoups.size()$ **Faire**
$partie_{clone} \leftarrow Clone(partie)$;
$partie_{clone}.appliquerCoup(ListdesCoups[i])$;
$TourSuivant(partie_{clone})$;
Si $!EstTerminee(partie_{clone})$ **Alors**
$a \leftarrow max(a, AlphaBeta(a, b, partie_{clone}, profondeur - 1, false))$;
Sinon
$util \leftarrow GetUtilite(partie)$;
Retourner util ;
fsi

$i++$;
Fait
Retourner a ;
Sinon
Tant que $a < b$ **et** $i < ListdesCoups.size()$ **Faire**
$partie_{clone} \leftarrow Clone(partie)$;
$partie_{clone}.appliquerCoup(ListdesCoups[i])$;
$TourSuivant(partie_{clone})$;
Si $!EstTerminee(partie_{clone})$ **Alors**
$b \leftarrow min(b, AlphaBeta(a, b, partie_{clone}, profondeur - 1, true))$;
Sinon
$util \leftarrow GetUtilite(partie)$;
Retourner util ;
fsi
$i++$;
Fait
Retourner b ;
fsi
fsi
Fin

2.1.2 Adaptation

Nous avons néanmoins du adapter ce code. En effet, l'IA doit pouvoir répondre à la méthode *getCoup()* qui renvoie un coup. Nous devons donc sélectionner le meilleur coup (celui avec la plus forte utilité). C'est pour cela que nous avons ajouté une méthode de "sur-couche" qui copie l'alpha-beta : *alphabetaPremier()*. C'est la première méthode appelée. Elle copie l'alpha-beta mais renvoie le meilleur coup. Elle correspond à la racine de l'arbre et elle appelle les méthodes alpha-beta classique.

Voici le pseudo-code de notre algorithme Alpha-Beta premier :

procédure *AlphaBetaPremier*(*partie* :Partie,*profondeur* :entier) :
var entier $a \leftarrow -\infty$, $b \leftarrow +\infty$;
var entier $i \leftarrow 0, a_{new} \leftarrow 0$;
var Coup[] *ListdesCoups*;
var Coup $CoupSelectionne \leftarrow null$;
Debut
$ListdesCoups \leftarrow GetTousCoups()$;
$ListdesCoups \leftarrow Filtrage(ListdesCoups)$;

$ListdesCoups \leftarrow MeilleurCoups(ListdesCoups)$;
Tant que $a < b$ **et** $i < ListdesCoups.size()$ **Faire**
$partie_{clone} \leftarrow Clone(partie)$;
$partie_{clone}.appliquerCoup(ListdesCoups[i])$;
$TourSuivant(partie_{clone})$;
$a_{new} \leftarrow max(a, AlphaBeta(a, b, partie_{clone}, profondeur - 1, false))$;
Si $a_{new} > a$ **Alors**
Si $a_{new} > CoupSelectionne.getUtilite()$ **Alors**
$CoupSelectionne \leftarrow ListdesCoups[i]$;
fsi
$a \leftarrow a_{new}$;
fsi
$i++$;
Fait
Retourner CoupSelectionne ;
Fin

2.2 Particularités mise en oeuvre

2.2.1 Filtrage des coups

Une fonction prédéfinie *getTousCoup()* nous renvoyait la liste de tous les coups possibles. Cependant, cette liste comportait également des coups visant les membres de notre propre équipe. Pour remedier à ce problème, nous nous sommes inspirés très fortement de l'*IA Aléatoire Agressive*. Ainsi, la liste obtenue comportait uniquement des coups visant l'adversaire. Ce filtrage est efficace à la fois pour éviter d'avoir des dégats inutiles et à la fois pour pouvoir mieux élaguer les branches dans notre algorithme Alpha-Beta.

2.2.2 Beam Search

Après filtrage des coups, nous sélectionnons les 10 meilleurs coups jugés les plus utiles grâce aux heuristiques définies plus bas. Pour cela, nous ajoutons d'abord tous les coups possibles obtenus par la liste filtrée dans une *TreeMap*. Ensuite, nous trions par heuristique la plus élevée de manière à obtenir les meilleurs coups en tête. Enfin, nous sélectionnons seulement les 10 premiers meilleurs coups de notre *TreeMap*.

Cela permet d'élaguer plus rapidement et également d'obtenir des profondeurs d'exploration de notre arbre élévée (aux alentours de 10). Ainsi nous avons pu mettre en place une profondeur dynamique (itératif) au lieu d'une profondeur de recherche fixée à 2.

2.3 L'utilitée

Nous avons défini la fonction d'utilité tel qu'elle renvoie un nombre de type double représentant notre plus ou moins grande chance de gagner. Si nous sommes à une feuille, l'utilité sera très grande en positif ou en négatif (car nous serons dans un cas de défaite ou de victoire).

Cette fonction est appelée en feuille, ou en cas d'arrêt de l'alpha-beta pour cause de profondeur trop importante.

Notre fonction d'heuristique a été développée en 2 phases. Tout d'abord, pour des raisons de développement de l'alpha-beta, elle renvoyait un double aléatoire. Ensuite, nous avons implémenté 2 cas : soit nous examinons une feuille, soit nous examinons un noeud lambda. Au final, nous avons constaté que les 2 pouvaient être résolu avec une fonction de points de vie. L'utilisation d'un logarithme permet de rapidement déterminer les valeurs menant à la victoire et la valeur menant à une défaite (nous faisons ln(Vie de notre équipe / Vie équipe adverse)

2.4 L'heuristique

Pour permettre d'élaguer plus rapidement les branches et de déterminer les meilleurs coups à jouer, nous avons déterminé plusieurs heuristiques qui vont permettre de mesurer un coup par le biais de la fonction *calculHeuristique()*. Elle est utilisée avant le *Beam Search*, ie avant la sélection des meilleurs coups et est nécessaire pour le tri et la comparaison des coups. Cette fonction prend en compte la partie à jouer et le coup qui doit être évalué ou mesuré.

Nous distinguons 2 cas : si le coup est une attaque ou un déplacement. Dans le cas où le coup est une attaque, l'heuristique principale est :

$$h := log(\frac{mesPV}{PVennemie - degats})$$

A cela s'ajoute plusieurs autres cas :

- Mini Bonus de $25pt * nbPersoCible$: Si l'attaque a pour effet de ralentir un personnage
- Mini Bonus de 20 pt : si l'attaque vise une zone
- Bonus de 200 pt : Si le personnage adverse meurt et qu'il n'a pas encore joué. Si celui ci meurt, nous gagnons donc son tour.
- Bonus de 100 pt : Si le personnage adverse meurt et qu'il a déjâ joué. Dans les deux cas, nous privilégions la mort d'un personnage adverse plûtot que d'infliger un maximum de dégats.
- Bonus de 50 pt : Si l'attaque est effective (ex : une attaque aérienne sur un personnage aérien comme le Cavalier, etc...).
- Malus de 100 pt : Si l'attaque n'est pas effective. (ex : Tremblement de Terre sur un Cavalier).
- Bonus de 200 pt : Si le personnage adverse est un Magicien. Nous accordons une grande importance au magicien de l'équipe adverse car celui ci peut attaquer tous les personnages à distance.
- Bonus de 100 pt : Si le personnage adverse est un Guerrier. Nous accordons une importance au Guerrier (moindre par rapport au Magicien) de l'équipe adverse car celui ci peut faire beaucoup de dégats avec son attaque Baliste de Feu.

Ainsi, les personnages de notre IA ont pour objectif d'attaquer les Magiciens et les Guerriers de l'équipe adverse en priorité et également de priviliéger les attaques qui permettent de mettre un personnage adverse KO qui n'ont pas encore joué.

Dans le cas où le coup est un déplacement, l'heuristique principale est :

$$h := 10 * distanceParcourue$$

A cela s'ajoute plusieurs autres cas :
- Malus de 10 pt : Si le personnage se déplace à proximité d'un personnage allié. Ce malus permet d'éviter que les personnages soit à proximité de son équipe dans le but de minimiser les dégats causés par une attaque de zone adverse.
- Bonus variable jusqu'à 100 pt max : Si le personnage se rapproche d'un personnage ennemi qui n'est pas un Guerrier. Cela permet de se déplacer vers l'ennemi afin de l'attaquer à courte distance.
- Malus variable jusqu'à 150 pt max : Si le personnage se rapproche d'un personnage ennemi qui est un Guerrier. Ce malus évite les déplacements des personnages près d'un Guerrier adverse car celui-ci , à faible portée, peut infliger des très gros dégats grâce à Baliste de Feu. Le Guerrier est donc un personnage à éviter à tout prix.
- Mini Bonus de $10pt * distanceParcourue$: Si le personnage qui se déplace est un Guerrier. Celui peut infliger plein de dégats à courte distance, donc il peut se rapprocher des personnages de l'adversaire.

Ainsi, les personnages de notre IA ont pour objectif de se déplacer vers le personnage ennemi sauf si celui-ci est un Guerrier. De plus, si notre personnage qui se déplace est un Guerrier alors il peut se rapprocher plus près de l'ennemi.

3 Déroulement

3.1 Taches

— Programmer et adapter l'alpha beta aux modifications du code du jeu pendant le projet.
— Conception de l'heuristique/utilitée.
— Codage de l'heuristique/utilitée.

3.2 Répartition

Au début du projet, nous étions tous les 3 sur l'analyse/conception. Puis nous nous sommes scindé en deux groupes : Ulrich et Javid étaient sur le développement de l'utilité et de l'heuristique. Antoine était sur les mises à jour/débugage de l'alpha-beta. Puis une fois le débugage terminé, nous étions tous les trois sur l'heuristique. La répartition n'a pas pus être plus poussée car le développement était très séquentiel.

3.3 Phase de test

Pour tester notre IA, nous avons lancé plusieurs parties en même temps sur des machines de l'ENSSAT. Nous avons commencé par faire des matchs contre l'IA *AleatoireAgressive* puis contre différentes versions actuelles de notre propre IA *CollectifIADynamos* (avec ou sans heuristiques) dans le but de visualiser et de valider les dernières modifications apportées à notre IA.

3.4 Difficultées

— Nous avons rencontré quelques problèmes lors du débugage de l'alpha-béta avec notamment des *null pointeur* intempestif et peu explicites.
— Beaucoup de temps perdu car nous n'avons pas détecté la dis-fonction de certaines fonctions (tel que *estTerminée()* ou *tourSuivant()*).

3.5 Ameliorations possibles

— Nous aurions pu faire des matchs contre les IA des autres groupes afin d'affiner/modifier nos pondérations d'heuristique, mais nous n'avons malheureusement pas eu le temps.
— Nous avions prévu aussi de caractériser les heuristiques suivantes le personnage joué (avec une optimisation certaine du voleur et du cavalier).

4 Conclusion

Ce projet nous a permis de mettre en pratique l'algorithme Alpha-Beta vu en cours et en TP, ainsi de bien maitriser les differents concepts de l'Intelligence Artificielle. Au cours de ce projet, nous avons été conscients de l'importance de la phase d'analyse et de conception qui nous a permis d'implementer plus rapidement notre IA et de pouvoir coder des heuristiques efficaces.