Rapport du projet IA

Anatole Beuzon Julen Dixneuf Corentin Dupret

Adnane Hamid Victor Paltz Sami Tabet

30 mars 2020

Table des matières

1 Introduction

Le but de ce projet est de développer une intelligence artificielle capable de jouer au jeu « Vampires contre Loups-Garous ». À la fin de l'électif, une dizaine de groupes d'IA vont s'affronter dans un tournoi afin de déterminer qui a programmé la meilleure intelligence artificielle.

2 Approche alpha-bêta : Langorou

Dans cette section nous nous intéressons à la première approche que nous avons exploré pour le jeu de vampires contre loups-garous : un min-max en version alpha-bêta.

Des instructions précises détaillant comment construire et lancer le projet sont disponibles ici.

En résumé il faut :

— Cloner le projet en faisant : `git clone git@github.com:langorou/langorou.git`

— Construire le binaire avec `make` à la racine du projet (cela nécessite d'avoir une toolchain go installée, version >= 1.11)

— Un binaire est ensuite produit au path : `build/langorou` et est utilisable de la manière suivant : `./build/langorou <host> <port>`

2.1 Choix de l'approche

Afin de réaliser notre IA, nous avons privilégié les performances et l'exploration du graphe des coups possibles. C'est pourquoi nous avons choisi de partir sur un algorithme min-max en langage Go et de l'optimiser de façon itérative.

Le code permettant de simuler les différentes étapes du jeu est contenu dans ce fichier.

2.2 L'heuristique

Nous avons fait le choix d'adopter une heuristique complexe et précise afin de pouvoir prendre en compte le futur que ne peut pas voir un algorithme seulement basé sur de l'exploration combinatoire à profondeur limitée.

Notre heuristique est également responsable de la génération de coups notamment pour autoriser ou interdire les scissions de groupes d'unités.

L'heuristique résultante contient de nombreux paramètres mais tient compte de nombreuses situations :

— **Counts** : coefficient pondérant la valorisation de notre population.

— **Battles** : coefficient pondérant la valorisation des batailles contre nos ennemis.

— **NeutralBattles** : coefficient pondérant la valorisation des batailles contre les villageois.

— **CumScore** : coefficient pour donner du poids aux branches qui atteignent un meilleur score plus rapidement en utilisant un score cumulé. Ce score cumulé est augmenté de $(1 - \alpha \times \text{depth}) \times (N_{\text{allies}} - N_{\text{ennemis}})$ à chaque profondeur, α permettant de donner plus d'importance au futur proche qu'au futur lointain.
— **WinScore** : valeur d'une victoire
— **LoseOverWinRatio** : contrôle la *risk aversion* de l'IA
— **WinThreshold** : seuil de probabilité à partir duquel on considère que la victoire est certaine (évite l'explosion combinatoire pour des apports mineurs de précision).
— **MaxGroups** : nombre maximum de groupes d'unités autorisé.
— **Groups** : coefficient pondérant la valorisation / pénalité des séparations en plusieurs unités.

Le code permettant de calculer l'heuristique est disponible ici.

2.3 Optimisations

De très nombreuses optimisations ont été réalisées pour atteindre de grandes profondeurs dans l'arbre :

2.3.1 Testé et approuvé

— **Version alpha-bêta de l'algorithme** : permet un élagage des branches de l'arbre dans l'exploration, et réduit la complexité temporelle à la racine carrée de la complexité maximale sans cette optimisation. Code disponible ici.
— **Manipulation des structures de données *in place*** : on évite au maximum les copies afin de gagner du temps. On utilise aussi des fonctionnalités avancées de Go (les `sync.Pool`) pour gagner 60% sur le temps d'exécution. Celles-ci permettent notamment de réutiliser des *slice* (tableau en Golang) allouées pour diminuer au maximum le nombre d'allocations mémoire. Le code correspondant est situé ici.
— **Utilisation du langage Go** : permet de faire tourner le programme sur tous les coeurs de la machine avec une performance proche de celle du C, notamment très utile pour paralléliser les tournois que nous avons réalisé pour optimiser les paramètres de l'heuristique.
— **Score des heuristiques cumulé** : si on est dans un cas de victoire dans l'heuristique, on veut donner la priorité aux victoires qui arrivent le plus tôt, de même pour l'action de manger des villageois. On a donc un score cumulé maintenu à chaque fois qu'un état est dupliqué comme réalisé ici.
— **Iterative Deepening et Killer Moves** : augmentation progressive de la profondeur de recherche, ce qui permet de toujours avoir une réponse dans le temps imparti. De plus, grâce aux tables de transposition, nous pouvons explorer les meilleurs coups trouvés aux profondeurs précédentes afin de favoriser des coupures alpha-bêta. Le code est disponible ici.

2.3.2 Testé et désactivé

— **Tables de transposition : caching** : afin de ne pas recalculer tous les résultats des évaluations des branches de l'arbre, nous gardons en mémoire ces derniers. Les performances sont grandement boostées grâce à cela. En effet, l'étude des marches aléatoires en 2D nous montre que revenir sur des cases proches du point de départ est très fréquent, le cache est donc très efficace. Cependant puisque nous utilisons une heuristique favorisant les chemins les plus lucratifs, deux états similaires peuvent avoir un score cumulé différent et nous ne pouvons donc plus utiliser des résultats précédents, c'est un problème connu sous le nom de *graph history interaction*. Nous avons donc décidé de désactiver cette partie là pour éviter d'avoir des incohérences dans le parcours des coups possibles. Le calcul du *hash* d'un état a été optimisé en réutilisant le même *buffer* de bytes tout au long du programme (pas d'allocation mémoires) et en utilisant la fonction de hash `murmur3`, le code correspondant est situé ici.

— **Structures de données pour l'état du jeu** : afin d'augmenter les performances de notre IA nous avions essayé d'utiliser différentes structures de données pour l'état du jeu :

  — Une liste de `(coordonnées, race, nombre)`.

  — Une matrice creuse de taille `width * height` dont les éléments étaient de la forme `(race, nombre)`

  — Un dictionnaire `coordonnées` → `(race, nombre)` statique partagé par tous les noeuds au début d'une nouvelle recherche alpha-bêta accompagné d'un dictionnaire de mises à jour `coordonnées` → `(race, nombre)` dynamique qui est recréé à chaque nouveau noeud. Ainsi lorsque l'on veut accéder à une cellule du jeu, on commence par regarder le dictionnaire de mise à jour avant de regarder le dictionnaire statique.

  Cependant toutes ces approches étaient plus lentes qu'un simple dictionnaire `coordonnées` → `(race, nombre)` cloné à chaque nouveau noeud.

2.4 Sélection des meilleures heuristiques

Afin de choisir les meilleurs paramètres de l'heuristique, nous avons réalisé des tournois entre différentes heuristiques ayant des paramètres différents.

Les paramètres retenus sont les suivants :

— **Counts** : 1

— **Battles** : 0.02

— **NeutralBattles** : 0.03 (un peu plus élevé que le coefficient valorisant les batailles contre des ennemis)

— **CumScore** : 0.0001, faible valeur afin de ne pas fausser l'évaluation, mais apporte une différence de valeur pour prioriser les situations réalisant les actions utiles le plus tôt.

— **WinScore** : 1e10 (une valeur plus élevée que celles que l'on peut obtenir avec les autres composants de l'heuristique)

— **LoseOverWinRatio** : 0.8 (afin d'inciter l'IA à jouer de manière agressive en donnant plus d'importance aux victoires qu'aux défaites)

— **WinThreshold** : 0.8 (compromis entre simplification de l'arbre de jeu et exactitude des simulations)

— **MaxGroups** : 2, utile pour réduire l'explosion combinatoire

— **Groups** : 0, fonctionnalité jugée non pertinente

2.5 Tests

Afin de s'assurer du bon fonctionnement de notre IA, en plus des tests manuels réalisés en lançant des parties contre une IA aléatoire (ou contre une IA similaire à celle testée), nous avons mis en place une batterie de tests disponibles ici.

3 Approche Deep Reinforcement Learning : Vamperouge

Dans cette section, nous nous intéressons à une autre approche pour réaliser une IA pour le jeu de vampires contre loups-garous : le *Deep Reinforcement Learning*. Plus précisément, nous implémentons un algorithme proche de Alpha Zero.

Vamperouge est une IA proche de Alpha Zero entraînée sur une version simplifiée du jeu de vampires contre loups-garous. Elle a été entraînée à l'aide d'un GPU NVIDIA Tesla P100 pendant environ 15h : 15 itérations de 100 parties ont été effectuées, avec 25 simulations de Monte-Carlo par tour de jeu (à titre de comparaison, AlphaGo Zero a été entraîné pendant 200 itérations de 25000 parties avec 1600 simulations par tour de jeu). Voir la section *Fonctionnement de l'algorithme* pour plus de détails concernant ces paramètres.

Le code source de Vamperouge est disponible ici.

3.1 Utilisation de l'IA

L'IA se lance de la manière suivante : `python3 vamperouge.py {adresse IP} {port}`. La version de Python prise en charge est la version `3.7.4`. Les *packages* `pytorch` et `numpy` sont nécessaires.

La configuration utilisée est celle de `main.py`, commune avec celle de l'entraînement. L'entraînement peut se lancer avec `python3 main.py` mais nécessite beaucoup de RAM et, idéalement, un GPU. Il n'est donc pas recommandé de la lancer sur sa propre machine.

3.2 Fonctionnement de l'algorithme

3.2.1 Monte-Carlo Tree Search

L'algorithme que nous implémentons essaie de se rapprocher de Alpha Zero. Il se base sur la recherche *Monte-Carlo Tree Search* (MCTS). Dans une position donnée, un certain nombre de simulations sont effectuées. Pour chaque simulation, l'arbre est parcouru jusqu'à arriver à un noeud non exploré ou un noeud terminal signifiant la fin de la partie (tous les monstres d'une équipe ont disparu ou le nombre maximal de coups a été atteint). À chaque noeud rencontré :

— **soit le noeud n'a pas encore été rencontré :** dans ce cas, le réseau de neurones est utilisé pour obtenir une *policy* (vecteur de probabilités des actions possibles) et une *value* (valeur représentant le joueur qui a l'avantage) initiales pour cette position

— **soit le noeud a été rencontré :** dans ce cas, on continue la simulation en choisissant le noeud correspondant à l'action appropriée, qui est celle qui maximise le critère UCB : $Q(s,a)+c_{\text{PUCT}}\times P(s,a)\times\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$, avec (pour une position s et une action a) $Q(s,a)$ la moyenne des scores des parties commençant par ce coup, $P(s,a)$ la probabilité antérieure telle que donnée par le réseau de neurones, $N(s,b)$ le nombre de parties commençant par l'action b (c_{PUCT} est un hyper-paramètre). Les noeuds de l'arbre contiennent et mettent à jour en permanence les informations permettant de calculer cette quantité.

Dans tous les cas, la *value* est propagée en arrière par le chemin parcouru.

Après avoir effectué ces simulations, l'IA choisit une action au hasard en se basant sur le nombre de fois que les différents noeuds correspondant aux différentes actions ont été explorés (en partant du principe que plus un noeud est exploré, plus l'action qui mène à ce noeud est bonne). Elle se base aussi sur un paramètre appelé température qui permet d'équilibrer l'exploration et l'exploitation. (Lorsque l'IA joue de manière compétitive, elle choisit la meilleure action 100% du temps : il n'y a pas d'exploration.)

Les simulations de la MCTS permettent d'obtenir une meilleure *policy* qu'une utilisation directe du réseau de neurones, dont nous détaillons maintenant la structure.

3.2.2 Réseau de neurones

Nous rappelons que le réseau de neurones permet, à partir d'une position donnée, d'obtenir la *policy* et la *value* correspondantes (il apprend les deux en même temps). Nous détaillons ici son architecture.

D'abord, la position du jeu est encodée sous forme d'un tenseur de dimensions $3 \times 16 \times 16$ (nous partons du principe que la taille maximum d'une carte est de 16 cases par 16 cases). Chaque plan de taille 16×16 contient des entiers correspondant au nombre de membres d'une espèce qui se trouvent dans une case (il y a 3 plans pour les 3 espèces).

La *value* est un entier entre -1 et 1. La *policy* est un vecteur de probabilités de taille $8 \times 16 \times 16$ (8 directions possibles pour chaque case du terrain). Les *splits* et les tours avec plusieurs coups ne sont pas pris en considération.

Le réseau de neurones est composé de :

- 1 couche convolutionnelle
- 4 couches résiduelles, comprenant chacune 2 couches convolutionnelles (AlphaGo Zero comprend 19 couches résiduelles mais nous en utilisons moins pour diminuer le temps d'entraînement et d'inférence, en partant aussi du principe que nous n'obtiendrons pas suffisamment de données pour que le réseau puisse apprendre autant de paramètres)
- à partir de là, deux réseaux différents pour la *policy* et la *value* :
  - pour la *policy*, 2 filtres de convolution puis une couche complètement connectée
  - pour la *value*, 1 filtre de convolution puis deux couches totalement connectées

Les données utilisées pour entraîner ce réseau de neurones sont générées au cours d'itérations pendant lesquelles l'IA joue contre elle-même.

3.2.3 Itérations d'entraînement

Pour l'apprentissage, l'IA joue contre elle-même pour tenter de s'améliorer à chaque itération.

Pour chaque itération, un certain nombre de parties sont jouées. Ceci permet d'obtenir un ensemble de positions associées à des *policies* améliorées (par simulations de MCTS) et des *values* déterminées par l'issue des parties. Cet ensemble de positions permet d'entraîner le réseau de neurones. Le nouveau et l'ancien réseau peuvent alors jouer des parties entre eux et, si le nouveau réseau gagne avec une certaine marge, l'IA est mise à jour pour utiliser ce nouveau réseau de neurones lors de ses parties.

Comme le jeu est symétrique par rotation et par réflexion, une même position correspond en fait à 8 positions différentes. Nous utilisons cette propriété pour augmenter la quantité de données pour l'entraînement du réseau de neurones. Nous transformons également, lors de la MCTS, les positions de manière aléatoire avant de les évaluer à l'aide du réseau de neurones, pour lisser l'évaluation de Monte-Carlo sur différents biais.

3.3 Limites

Nous avons déjà évoqué ci-dessus certaines limites de cette approche :

- Nous avons limité la taille du terrain à 16 par 16, bien que nous aurions pu tester des méthodes pour éviter une telle contrainte. Si Vamperouge reçoit une carte plus grande, il y aura des erreurs d'indice (nous pourrions les éviter, mais Vamperouge ne serait pas capable de "voir" la carte entière, ce qui a peu d'intérêt, notamment si ses coups possibles sont en dehors de son "champ de vision").

— Nous ne prenons pas en compte les *splits* et les tours avec plusieurs coups ne sont pas possibles, bien que cela aurait pu se concevoir avec plus de temps pour les implémenter dans la *policy* (cependant, il aurait également fallu prévoir un temps d'entraînement plus long).

— Cet algorithme demande une puissance de calcul considérable : nous avons donc dû ajuster les paramètres du papier de Deep Mind pour les adapter à nos ressources.

— Nous introduisons un biais lors de la génération aléatoire des terrains sur lesquels l'IA doit s'entraîner, qui ne correspondent pas nécessairement aux terrains rencontrés lors du tournoi.

3.4 Performances

Vamperouge est capable de battre l'aléatoire et de gagner quelques parties contre Langorou. Cependant, il reste très faible par rapport à ce dernier. En effet, Vamperouge a tendance à éviter de chercher le combat avec les humains (probablement car il a été entraîné sur des cartes où un nombre important de cases avec des humains conduit à la défaite en cas de combat). Il ne se suicide donc pas très souvent, mais il a beaucoup de mal à trouver le chemin permettant de combattre les humains qu'il est capable de convertir (c'est-à-dire les cases où les humains sont moins nombreux). Langorou, au contraire, est capable de trouver ces chemins.

L'IA obtenue par entraînement selon l'algorithme Alpha Zero est donc décevante, mais pourrait probablement être meilleure avec plus de ressources allouées à l'entraînement.

4 Conclusion

Concevoir une IA performante pour ce jeu n'est pas une chose aisée.

D'une part, concernant l'approche alpha-bêta, la combinatoire immense du jeu ne permet pas d'explorer l'arbre des possibilités avec une grande profondeur. Il est donc nécessaire de trouver une bonne heuristique, ce qui est une tâche difficile.

D'autre part, concernant l'approche Deep Reinforcement Learning, la complexité et les composantes aléatoires du jeu posent des défis concernant l'adaptation de l'algorithme d'apprentissage. De plus, les ressources nécessaires pour entraîner une IA performante sur un jeu aussi complexe sont gigantesques.

Cependant, Langorou maîtrise les principes de base du jeu, et obtiendra sans doute une place décente à l'issue du tournoi !