Reinforcement Learning Library based on Tangled Program Graphs: Development of New Learning Environments and Library Features

Internship Report, IETR-Vaader 2020

Pierre-Yves Raumer[1], Nicolas Sourbier[1], Mickaël Dardaillon[1], Maxime Pelcat[1], and Karol Desnos[1]

[1]Univ Rennes, INSA Rennes, CNRS, IETR - UMR6164

August, 2020
Version v1.0

Abstract

Tangled Program Graph (TPG) is a reinforcement learning technique built on genetic programming principles. GEGALATI is a c++ library for training and inferring TPGs, developed at the IETR. In this document, we report the latest developments of the library, resulting from the work achieved during a 3-month master internship. First, a set of new learning environment were developed to assess the capabilities of the TPG model. The newly developed environment include: a state-of-the-art retro-gaming emulator, a multi-player tic-tac-toe game, and a cyber-physical system consisting of a robotic arm equipped with 6 servo-motors. Second, a set of practical features was developed to ease the usage of the library, notably by parsing meta-parameters used for training from a JSON file, and by providing a customizable infrastructure for logging the progress of TPGs training. Third, a dedicated training algorithm was prototyped for adversarial multi-agent learning environments. Promising result were achieved in several domains, notably on the use of TPGs for controlling cyber-physical systems, and on the use of TPGs in adversarial contexts.

1 Introduction

1.1 L'IETR, un Institut de Recherche

L'IETR (Institut d'Electronique et des Technologies du numéRique) est un Institut de recherche public regroupant près de 400 enseignants chercheurs implanté à plusieurs endroits, dont notament Rennes et Nantes. Ses domaines de recherche sont très variés et gravitent autour de l'électronique, des télécommunications et de l'informatique, principalement de bas niveau. Sur Rennes, l'IETR est installé, entre autres, parmi des locaux de l'INSA. C'est dans ces locaux, au sein de l'équipe VAADER (Video Analysis and Architecture Design for Embedded Resources), que j'ai réalisé mon stage.

Les domaines d'étude de l'équipe VAADER couvrent de nombreux thèmes allant de la conception des algorithmes de traitement d'images et de vidéo jusqu'aux implantations sur des systèmes numériques embarqués.

1.2 Le projet Gegelati

Le TPG (Tangled Program Graphs) est une notion imaginée par S. Kelly au travers de sa thèse publiée en Juin 2018 [1]. Cette technique d'apprentissage par renforcement permet le développement progressif d'un graphe servant à prendre des décisions discrètes en fonction de l'état d'un environnement.

Particulièrement rapide à l'exécution, léger et efficace en ce qui concerne sa vitesse d'apprentissage, le TPG semble prometteur. Ces caractéristiques rendent son utilisation des plus pertinentes dans le cadre de systèmes embarqués, ceux-ci étant très souvent contraints en terme de ressources. C'est pourquoi Karol Desnos, enseignant-chercheur à l'INSA, a souhaité développer une bibliothèque, "Gegelati", afin d'utiliser et d'expérimenter les TPG en C++.

Outre K.Desonos, Nicolas Sourbier, doctorant, travaille sur le thème des TPG en lien avec la cybersécurité et contribue au développement du projet. Afin d'avancer davantage, le choix de proposer un stage sur ce thème a été effectué. C'est ainsi qu'a commencé le stage dont ce document constitue le rapport, il aura duré du 1er juin au 28 août, avec un arrêt de 2 semaines en août pour cause de fermeture de l'INSA.

1.3 Un environnement de travail inconnu

Travailler au sein de la recherche était quelque chose de nouveau. Si un stage réalisé au sein de la plateforme de recherche Nano-Rennes durant la 2-STPI m'a permis d'aborder une première fois ce milieu, ma mission consistait cependant en un travail technique simple, à savoir la réalisation d'un site internet.

Ce stage, cependant, comportait une part importante de recherche. En effet, les applications et améliorations qui ont été effectuées dans le cadre de celui-ci avaient en général pour but de comparer, mesurer ou expérimenter quelque chose. Par ailleurs, les discussions techniques récurrentes ou encore les réunions sur l'avancement futur du travail sur Gegelati m'ont permis de m'intégrer à la mission de recherche se trouvant derrière Gegelati et d'entrevoir ce milieu de l'intérieur.

J'ai également pu découvrir l'organisation et le mode de travail de l'équipe. Cependant, la situation sanitaire (Covid-19) a rendu mes observations relativement lacunaires puisque, beaucoup de personnes étant absentes, je n'ai pas eu l'occasion de comprendre le rôle exact

de chacune. Cependant, l'opportunité que j'ai eu de travailler en présentiel tous les jours de mon stage m'a permis de profiter au mieux de mon expérience. Par ailleurs, un aspect positif de la situation a été de me permettre d'observer la façon dont l'Institut et l'INSA se sont adaptés au problème.

Sans rentrer dans tous les détails, l'une des choses m'ayant le plus marqué est l'aspect peu hiérarchique de la recherche. En effet, en comparaison avec mes stages en entreprise où il était aisé de réaliser un organigramme vertical représentant la hiérarchie, l'organisation de la recherche semble davantage horizontale. Malgré la présence de responsables les chercheurs sont relativement autonomes dans leurs travaux. Partageant leurs travaux par l'intermédiaire de séminaires et de réunions, et travaillant en équipes, la collaboration est pour autant une part importante de leur travail. A titre personnel, je réalisais des réunions chaque semaine avec mes encadrants concernant l'avancement du stage et plus généralement le travail autour de Gegelati.

2 Contexte technique du stage

2.1 TPG

Les TPG (Tangled Program Graphs) sont des graphes permettant de prendre des décisions en fonction de l'état d'un environnement, apprenant par renforcement. Inventés par S. Kelly dans le cadre de sa thèse [1], ils ont, comme évoqué, plusieurs avantages et se sont avérés relativement performants. S. Kelly a utilisé des jeux Atari 2600 via la bibliothèque ALE (Arcade Learning Environment) [2] pour évaluer les TPG et les comparer à d'autres techniques telles que DQN (Deep Q Learning). Il s'avère que les TPG semblent comparables voire parfois meilleurs que ceux-ci en terme d'efficacité.

Le TPG peut être considéré comme un graphe orienté proche d'une arborescence, mais avec plusieurs noeuds racines. Ses sommets/feuilles peuvent être de trois types : noeuds (dont certains sont racines), programmes et actions atomiques (ces dernières étant les feuilles). Les noeuds organisent les parcours du graphe, les programmes permettent de choisir le chemin suivi et les actions atomiques de mettre fin au parcours en ayant pris une décision. Les rôles de ces éléments seront davantage explicités dans les paragraphes suivants.

Un visuel de TPG simplifié est proposé dans la figure 1. En réalité, les TPG sont en général beaucoup plus grands (pouvant parfois posséder des milliers de noeuds) et comme évoqué ils possèdent plusieurs noeuds racines, là où la figure n'en présente qu'un.

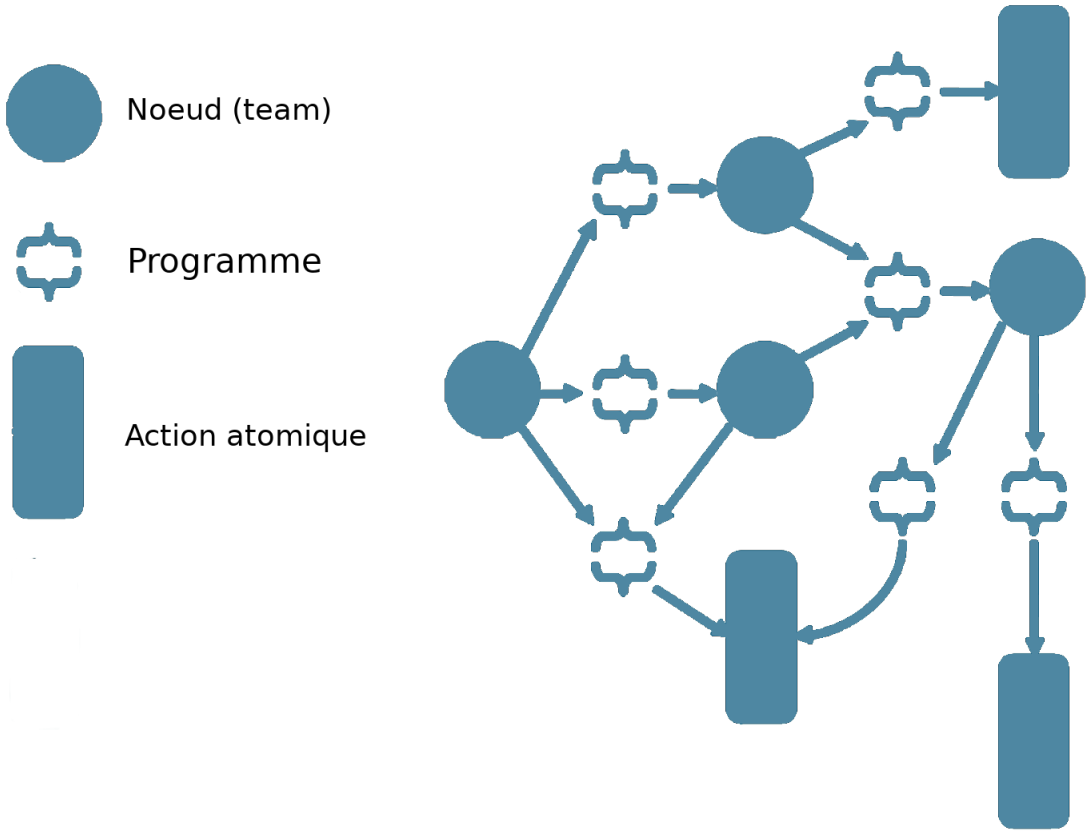

Figure 1: Représentation d'un TPG simple

D'un point de vue global et modulaire, le principe de l'utilisation des TPG se décompose en trois temps : il faut tout d'abord sélectionner les variables que l'on souhaite transmettre au TPG (dit l'"état"), puis parcourir le graphe d'un noeud dit "racine" à une sortie (une action atomique), et enfin appliquer une décision représentée par cette sortie. Ces trois étapes sont répétées autant de fois qu'il y a de décisions à prendre. Par exemple, dans le cas des jeux Atari 2600, il est possible de prendre une décision à chaque "frame", celles-ci apparaissant à une fréquence de 60 Hz.

Pour commencer, les variables d'entrée constituent un élément central dans le bon fonctionnement de l'agent. En effet, il est intuitif que si les variables d'entrée ne représentent pas du tout l'environnement dans lequel l'agent doit agir, il ne pourra prendre la bonne décision. Dans l'exemple de l'utilisation des TPG avec des jeux Atari 2600, puisque à chaque instant donné (une "frame") le TPG devra prendre la décision d'appuyer sur une touche de la manette ou non, les variables auxquelles il a accès doivent représenter l'état actuel du jeu. Il peut s'agir d'un simple tableau à deux dimensions contenant l'ensemble des pixels de l'écran, ceux-ci pouvant être représentés dans un seul octet en mode SECAM. En l'occurrence, S. Kelly a utilisé une forme de convolution permettant de représenter en

un seul octet un bloc de 5x5 pixels.

Une fois ces variables obtenues, il est nécessaire de parcourir le graphe pour prendre une décision. Il faut alors déterminer un noeud racine dans le graphe. Celui-ci en a plusieurs, mais un parcours nécessite d'en choisir un. Ce sera la position de départ lors de ce parcours.

La figure 1 met en évidence qu'après chaque noeud, on rencontre des programmes. Tous les programmes fils du noeud racine doivent être évalués. Pour cela, ils effectuent des opérations diverses sur les données d'entrée sans toutefois les modifier et expriment chacun une unique valeur réelle.

Le principe est d'utiliser un banc de registres pour permettre une communication entre instructions. Le programme n'est qu'une liste d'instructions et celles-ci sont exécutées dans l'ordre. Chaque instruction réalise alors une opération unitaire, telle une multiplication ou un cosinus, en prenant comme opérandes des éléments des variables d'entrée, ou l'un des registres du banc. La sortie de l'opération est placée dans un registre. Après avoir exécuté les instructions du programme, on a donc des variables d'entrée qui n'ont pas changé et un banc de registre rempli avec des valeurs diverses. L'un de ces registres, désigné dès le départ sous le nom R[0], est lu et sa valeur est utilisée comme sortie du programme. Cette valeur est appelée "mise" (ou "bid" dans la langue originale des TPG). On a donc, pour chaque programme fils du noeud racine, une mise. Le principe est alors simplement de choisir le chemin dans lequel le programme a la mise la plus élevée. Le parcours nous amène alors, après le passage du programme, à un autre noeud ou à une sortie. S'il s'agit d'un autre noeud, il faut évaluer ses programmes fils pour poursuivre et finir par arriver à une sortie.

On constate que le graphe dispose de plusieurs sorties. Chacune contient une valeur, un entier naturel, comprise entre 0 inclus et le nombre d'actions possibles exclu. Par exemple, les jeux Atari 2600 utilisent un total de 18 actions possibles, il s'agit de l'appui de boutons, de combinaisons de boutons et de "l'action vide" ne faisant rien. Une fois l'action effectuée, une frame passe, l'état de l'environnement évolue, et le processus de décision est réitéré.

2.2 Programmation génétique

De la structure des TPG présentée dans la section 2.1, on peut déduire qu'il y a besoin de déterminer un certain nombre d'éléments pour qu'il puisse répondre au problème posé. Ne partant de rien, il est tout d'abord nécessaire de concevoir sa structure. Par ailleurs, au sein de cette structure, on trouve des programmes qui effectuent comme évoqué des opérations diverses sur les données. Il est donc également nécessaire de déterminer quelles opérations utiliser, dans quel ordre et avec quels opérandes et sorties.

Tout ceci se détermine à l'aide de programmation génétique. Le concept de programmation génétique provient de la théorie de l'évolution. Sans parler de TPG, le concept générique est de suivre une telle procédure :

1. Générer des individus aléatoires, constituant une première génération
2. Évaluer tous les individus de la génération
3. Supprimer les plus mauvais des individus (typiquement 90%)
4. Faire reproduire les individus conservés en leur appliquant des mutations, obtenant une nouvelle génération

5. Retourner à l'étape 2 et perpétuer ce cycle autant de fois que nécessaire, cherchant souvent à converger vers un comportement optimal (au moins localement)

Nous allons expliciter l'application d'un tel concept aux TPG.

Au départ, un graphe aléatoire est constitué. Il s'articule selon des noeuds et des programmes générés aléatoirement, en respectant des contraintes définies. Par exemple, il est possible de limiter le nombre de sorties qu'aura un noeud, ou le nombre de noeuds racines. Ensuite, nous allons considérer chaque noeud racine comme un "individu". En effet, parcourir le graphe à partir de différents noeuds racines peut conduire à différents résultats. Une génération est alors un unique TPG.

Dans notre situation, considérant un environnement tel que les jeux ATARI 2600, le score n'est souvent exploitable qu'à la toute fin de la simulation (à la fin du jeu). Une génération est donc entraînée sur une ou plusieurs parties entières. On procède ainsi : pour chaque noeud racine, une simulation est effectuée, ou plusieurs si les résultats d'une seule partie ne semblent pas significatifs de la qualité de l'individu. À chaque frame de la simulation, c'est en parcourant le graphe à partir de ce noeud racine que l'on prend la décision. À la fin de la simulation, ou des simulations, un score est extrait et il est utilisé pour classer les noeuds racines.

Une fois ces noeuds racines classés, les plus mauvais sont élagués. Les programmes ou autres éléments se trouvant isolés sont également élagués. Il arrive qu'un noeud, autrefois descendant du noeud racine, se retrouve lui-même racine puisque sans prédécesseur. Suite à cela, de nouveaux noeuds racines sont générés en copiant les éléments de ceux qui ont été conservés, appliquant au passage des mutations. Les mutations incluent des ajouts, suppressions ou modifications de programmes, des absorptions d'autres noeuds par un nouveau noeud racine (ces nouveaux noeuds perdant alors leur qualité de noeuds racines), des ajouts de nouvelles sorties du noeud...

On recommence ainsi jusqu'à avoir réalisé suffisamment de générations. Ceci permet le développement d'une modularité croissante, chaque noeud dans le graphe correspondant en général au traitement d'un cas particulier de l'environnement. Le graphe change et grandit au fil du temps, jusqu'à adopter un comportement optimal pour peu que l'apprentissage réussisse.

2.3 Gegelati

Les TPG étant théorisés dans la thèse de S. Kelly et rendus partiellement accessibles par ce dernier, la création d'une bibliothèque rendant leur utilisation facile et efficace semblait pertinente. C'est l'ambition de Gegelati, implémentant le concept en C++ dans le but de pouvoir expérimenter autour de ce concept émergent. Créée dès la fin d'année 2019, elle est devenue réellement utilisable en 2020 en insistant sur deux points : la parallélisabilité de l'exécution de la bibliothèque et son déterminisme. Le premier point permet à plusieurs individus durant une génération d'être évalués simultanément à l'aide des différents coeurs du CPU et le second permet la reproductibilité des résultats. L'usage des mêmes paramètres avec un environnement lui aussi déterministe doit conduire précisément au même TPG, aux mêmes comportements et par conséquent aux mêmes scores. Finalement, après avoir implémenté le concept original des TPG en s'assurant de ces deux caractéristiques, Gegelati a pu être améliorée, optimisée et servir de base d'expérimentation. À titre d'exemple, des modules pratiques tels des loggeurs ou un parser ont été implémentés, des optimisations de temps de calcul ont été trouvées et des types d'apprentissages tels la classification, l'adversarial ont été permis.

Gegelati ayant pour ambition d'être une bibliothèque fiable, des tests unitaires ont été créés et l'utilisation de Travis a été mis en place dans une démarche d'intégration continue. Présente sur Github, la bibliothèque se voit compilée, testée et analysée à chaque modification (push et pull request). L'analyse inclut la vérification de la couverture des lignes ajoutées/modifiées par les tests unitaires, l'estimation de la qualité du code, la vérification de la documentation complète du code et la vérification du formatage correct de celui-ci. Ainsi, les exigences de l'intégration continue rendent impossible la modification des sources en ligne si l'un de ces éléments ne donne pas de conclusion positive. Par ailleurs, de bonnes pratiques de code sont recommandées et des éléments, comme un Changelog ou un Readme, sont régulièrement mis à jour.

3 Missions réalisées

3.1 Applications

Une part importante de mon stage a été de créer des applications afin d'utiliser Gegelati pour apprendre à un TPG à y être performant. Parmi elles, il y a un jeu "TicTacToe", des jeux de la console Atari 2600 et un bras robotique dont, bien que l'équipe en ait physiquement un, une bibliothèque permet la simulation. Il est à noter que pour évaluer toutes ces applications, il était nécessaire de pratiquer un certain nombre d'entraînements. Cela nécessitant des ressources, j'avais à ma disposition un ordinateur de bureau de 6 coeurs et un cluster de 24 coeurs, que j'utilisais en complément de mon ordinateur personnel.

3.1.1 TicTacToe

La première mission, après la lecture de la documentation associée aux TPG, fut de créer une application simple avec laquelle utiliser Gegelati. Le choix de créer un Tic-tac-toe, autrement appelé morpion, a alors été pris. Le principe de ce jeu est simple; chaque joueur ayant un symbole (croix ou rond) il faut parvenir à en aligner 3 dans une grille de jeu en 3x3 pour gagner.

Une implémentation du jeu en C++ a été rapidement réalisée, puis la connexion avec Gegelati s'est également déroulée sans encombre. Afin d'entraîner les TPG à jouer à ce jeu, un joueur aléatoire a été créé. Le principe est que le TPG doit proposer l'une des cases de 0 à 8 lors de son tour, puis le joueur aléatoire choisit l'une des cases libres au hasard. Un score de 1 est attribué lors d'une victoire, de 0,5 lors d'un match nul et de 0 lors d'une défaite. À ce score est ajouté un malus de -1 dans le cas où, durant le jeu, le TPG aurait proposé de jouer sur une case déjà occupée; c'est alors un mouvement non légal. Lorsque cela arrive, une décision de jeu aléatoire est prise à la place de ce que proposait le TPG.

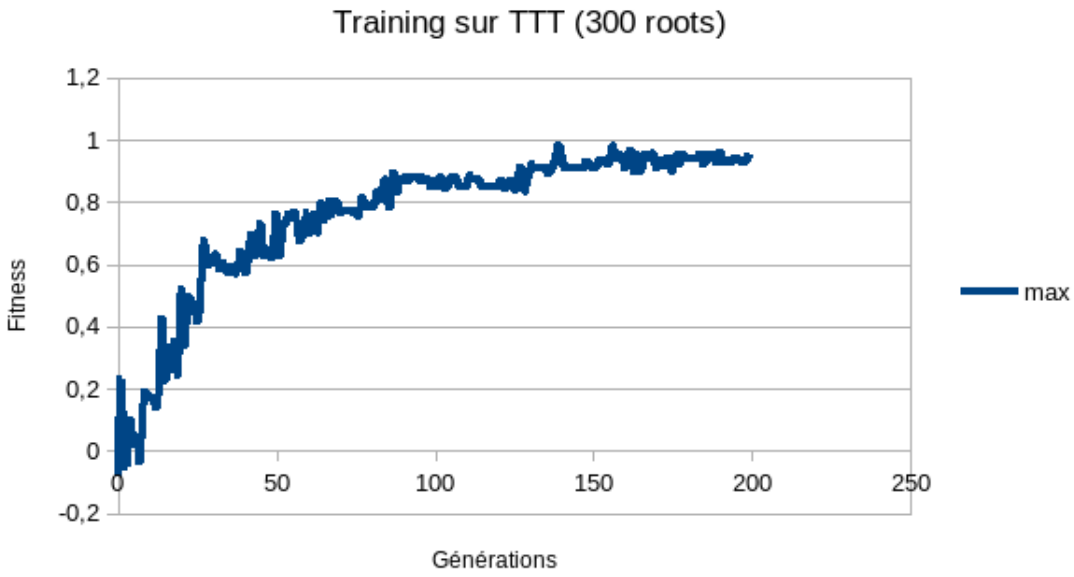

Figure 2: Evolution du score du meilleur individu dans Tic-tac-toe

La figure 2 représente l'évolution du score qu'obtient l'agent en fonction des générations. À chaque génération, on a 1000 noeuds racines, c'est à dire 1000 individus, et 95% de sélection entre chaques générations. On peut constater sur les résultats de la figure qu'au début de l'entraînement, le TPG ne parvient pas à jouer en respectant les règles et par conséquent il a tendance à perdre ou à faire des matchs nuls tout en proposant des mouvements non légaux, gardant un score négatif. Cependant, il parvient rapidement à atteindre 0 puis tend ensuite vers un score de 1, arrivant finalement au bout d'une cinquantaine de générations à une situation où la victoire est quasi systématique, laissant place à de rares nuls, tout en jouant uniquement sur des cases vides.

On peut alors considérer ce test succinct comme une réussite, puisque l'agent parvient à battre son adversaire. Cependant, il est possible d'exprimer une réserve sur le résultat : en effet, si l'on fait jouer le meilleur individu contre un humain, il aura tendance à perdre. Pourtant, au Tic-tac-toe, il existe une stratégie optimale permettant au pire des cas d'atteindre le match nul, que l'on soit premier ou second joueur. Le problème est ici que pour battre un joueur aléatoire, l'individu n'a pas eu besoin de développer d'importantes stratégies au jeu. Par conséquent, un individu pouvant battre un joueur aléatoire a été obtenu, mais il n'est pas pour autant un joueur parfait.

3.1.2 Arcade Learning Environment

De manière assez naturelle, la seconde mission a été de relier Gegelati à l'Arcade Learning Environment, permettant de jouer à des jeux Atari 2600. C'est en effet comme évoqué dans la section 2.1 la méthode utilisée par S. Kelly pour évaluer ses résultats.

Cette fois, l'application n'était pas à implémenter, le travail consistait principalement à comprendre la structure de la bibliothèque ALE afin de s'en servir et de la relier à

l'architecture des programmes utilisant Gegelati. Cela a été fait assez rapidement, les deux bibliothèques étant toutes deux claires et faciles à utiliser. Il y a eu cependant quelques ajustements et travaux à réaliser.

Par exemple, S. Kelly utilisait le mode SECAM permettant au jeu de n'être qu'en 8 couleurs. Une capture d'écran d'un jeu dans ce mode est disponible Figure **??**. Il utilisait à partir de ce mode une méthode pour réduire l'espace d'entrée. En effet, comme évoqué, l'idée est de diviser l'écran en carrés de 5x5. On construit alors un octet par carré, chacun des bits valant 1 ou 0 en fonction de la présence de l'une des 8 couleurs ou non. Cela permet de diviser le nombre des variables par 25 et ne fait pas perdre beaucoup d'informations : les formes dans les jeux Atari 2600 sont assez grandes et même en diminuant ainsi la résolution de la perception de l'écran, il reste possible de distinguer les éléments essentiels au bon déroulement du jeu.

Figure 3: Capture d'écran du jeu Frostbite

Une fois cela implémenté, des tests ont alors pu être mis en place. Des apprentissages ont été réalisés sur des jeux classiques tels que Frostbite, Fishing Derby, Ms. Pac-Man ou encore Asteroids. De bons scores ont été obtenus. Par exemple, le meilleur agent sur Frostbite atteint un score supérieur à 9000, dépassant largement le score typique d'un humain débutant qui approche 5000 points. On peut voir l'évolution du score de cet entraînement en fonction des générations sur la figure 4. L'agent parvient à réaliser des mouvements de précision, mais sa mauvaise compréhension du jeu lui fait parfois réaliser des actions l'amenant à perdre. Les résultats de S. Kelly ont pu être atteints, démontrant que Gegelati a correctement implémenté la notion des TPG.

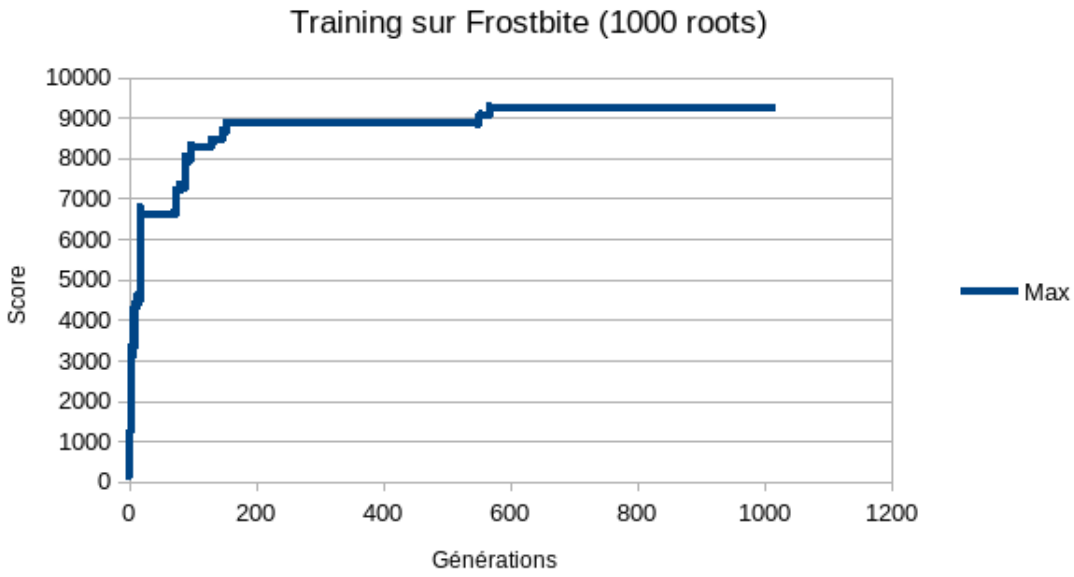

Figure 4: Evolution du score du meilleur individu sur Frostbite

Utiliser Gegelati avec l'ALE a permis de comparer les performances de Gegelati avec celles de l'implémentation de S. Kelly, en terme de temps notamment. La figure 5 est un exemple de graphique comparant le nombre de générations en fonction du temps dans le cas de l'implémentation de S. Kelly et dans le cas de Gegelati. Gegelati permettant le multithreading, on peut voir qu'une courbe représente le temps d'exécution d'une sélection par Gegelati avec un seul thread, et une autre avec 6. Si Gegelati, du fait de son architecture, a tendance à être un peu plus lente que l'implémentation de S. Kelly avec un seul thread, son multithreading lui permet d'être très rapide dans l'autre cas où l'on dispose de 6 coeurs.

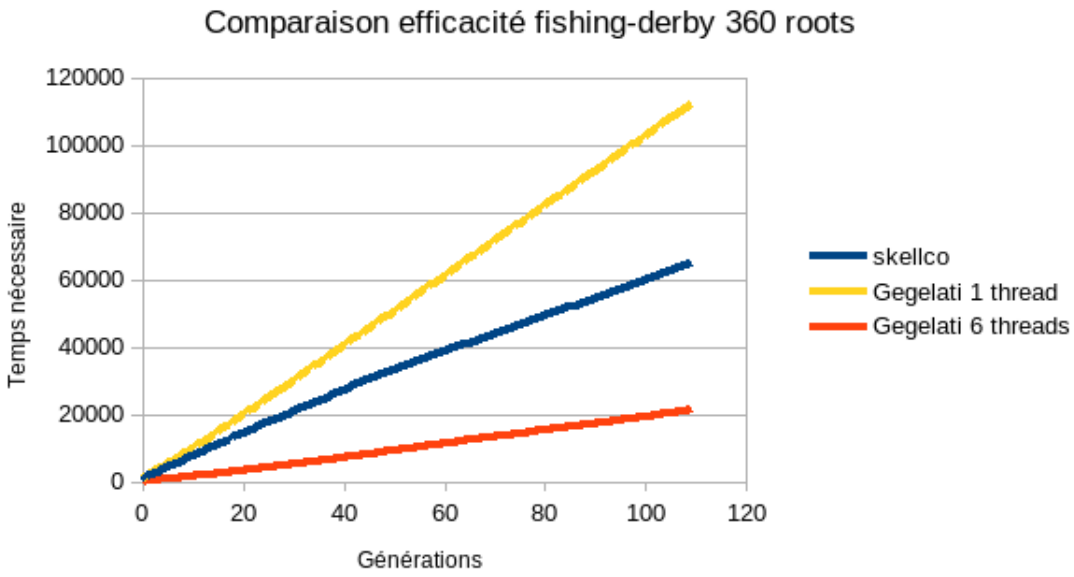

Figure 5: Comparatif de vitesses d'exécution lors d'un entraînement sur le jeu Fishing Derby

3.1.3 Armlearn

Une autre application avec laquelle Gegelati a travaillé est Armlearn [3]. Armlearn est une bibliothèque écrite par Gael Gendron, stagiaire à l'IETR en été 2019. Elle permet d'utiliser facilement au travers d'instructions de haut niveau un bras robotique composé de 6 servomoteurs (des moteurs pas à pas). L'équipe dispose d'un bras de ce type et j'ai pu m'en servir au cours de mon stage, mais les prochains paragraphes qui traitent de l'apprentissage concernent des simulations purement virtuelles qu'Armlearn permet également, dans le but de pouvoir réaliser des millions de mouvements de grande ampleur en quelques minutes ce qui ne serait évidemment pas possible avec le vrai bras.

Les 4 premiers moteurs du bras disposent de 4096 crans, les deux derniers étant plus réduits (1024 et 512). Cela lui permet de couvrir une grande distance : mesurant environ 50 centimètres tendu, les positions que le bras peut théoriquement atteindre se situent dans un espace proche d'une demi-sphère de rayon 50 cm. L'espace réellement atteignable au cours des simulations a été estimé en générant 10 millions de points atteignables aléatoirement, une capture du graphe 3D obtenu sur R est visible dans la Figure 6.

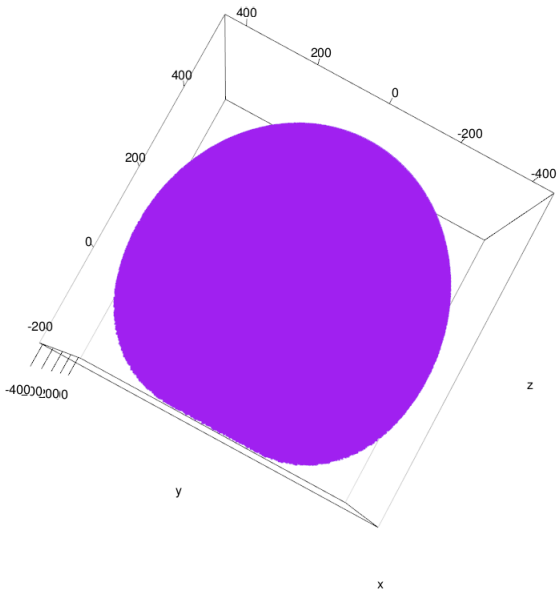

Figure 6: Volume 3D formé par les 10 millions des points atteignables par le bras robotique.

Le problème auquel devait répondre Gegelati est le suivant : étant donné la position de la tête du bras robotique dans un repère cartésien et une position cible dans ce même repère, sachant que l'on ne peut agir que sur les 6 moteurs qui ne peuvent recevoir qu'une consigne d'angle, quelles instructions donner pour que la tête atteigne la position cible ? Ce n'est pas une question triviale. Un système d'équations permet de résoudre algébriquement le problème, seulement, en situation réelle, il peut arriver que l'un des moteurs soit grippé ou encore qu'il y ait un obstacle sur le terrain. Dans ce cas, la modélisation algébrique devient très complexe. C'est là qu'intervient l'apprentissage : une intelligence artificielle peut apprendre à bien guider le bras malgré ces défauts.

Cette fois, il fallait réaliser une connexion entre la bibliothèque Armlearn et Gegelati. Ces deux bibliothèques fonctionnant en demandant à l'utilisateur de réaliser un héritage (en programmation orientée objet), un objet en double héritage a été créé pour bénéficier au mieux des deux bibliothèques. Une fois cela fait, il a fallu penser à des méthodes d'évaluation. Gael Gendron ayant déjà réalisé un apprentissage, mais par réseaux de neurones, il a au départ suffi d'utiliser ses méthodes. Il s'agissait de définir une ou plusieurs positions cibles, et d'entraîner Gegelati à les atteindre à tour de rôle. Cela donnait de bons résultats, néanmoins, après apprentissage, l'agent obtenu était relativement incapable de généraliser : il parvenait rarement à approcher de manière satisfaisante une cible qu'il n'avait pas rencontrée durant son apprentissage.

Une idée a alors été de générer des cibles aléatoires après s'être entrainé quelque chose comme 40 générations sur des cibles définies "à la main" pour avoir une difficulté croissante. On avait ainsi par exemple 10 générations sur une cible tout près de la position initiale, 10 sur une position plus loin, 10 sur une de l'autre côté, et 10 sur une cible bien plus loin, avant d'embrayer sur des cibles aléatoires (pouvant elles aussi rester les mêmes pendant 10 générations). Cela a donné de bons résultats et, finalement, en réduisant petit à petit les générations passées sur les cibles choisies "à la main" leur relative inutilité a pu être mise

en évidence. Plaçant dès le départ des cibles aléatoires, les résultats étaient tout aussi bons, j'ai donc complètement supprimé la notion de cibles choisies "à la main".

Ensuite, suite à des dicussions durant les réunions hebdomadaires, j'ai réduit petit à petit le nombre de générations passées sur chaque cible, jusqu'à atteindre 1, ce qui semblait plus rapide que 10. L'étape suivante a été, dans cette même logique, d'utiliser 2 cibles pour une seule génération : chaque noeud racine était testé sur les 2 cibles. Il semble qu'au lieu d'évaluer le bras sur sa capacité à atteindre des positions déterminées, l'évaluer sur sa capacité à rejoindre n'importe quelle position de l'espace est plus efficace, par conséquent j'ai porté le nombre de cibles par génération à 100 pour permettre qu'elles soient relativement représentatives de l'espace tout en évitant un temps de calcul trop long.

Cela nécessitait donc 100 simulations par individu à chaque génération, et puisque l'on a 1000 noeuds racines et 1000 actions par simulation, cela représentait au total 100 000 000 de décisions prises par le TPG à chaque génération. Néanmoins, en moins d'une journée de bons résultats ont été atteints. L'évolution des distances moyennes aux 100 cibles en fonction des générations est visible sur la figure 7. Les distances sont exprimées en unités arbitraires, 10 unités valant environ 1 cm. Le bras parvient donc en moyenne, dans les dernières générations, à s'approcher à une distance de la cible de l'ordre du demi-centimètre.

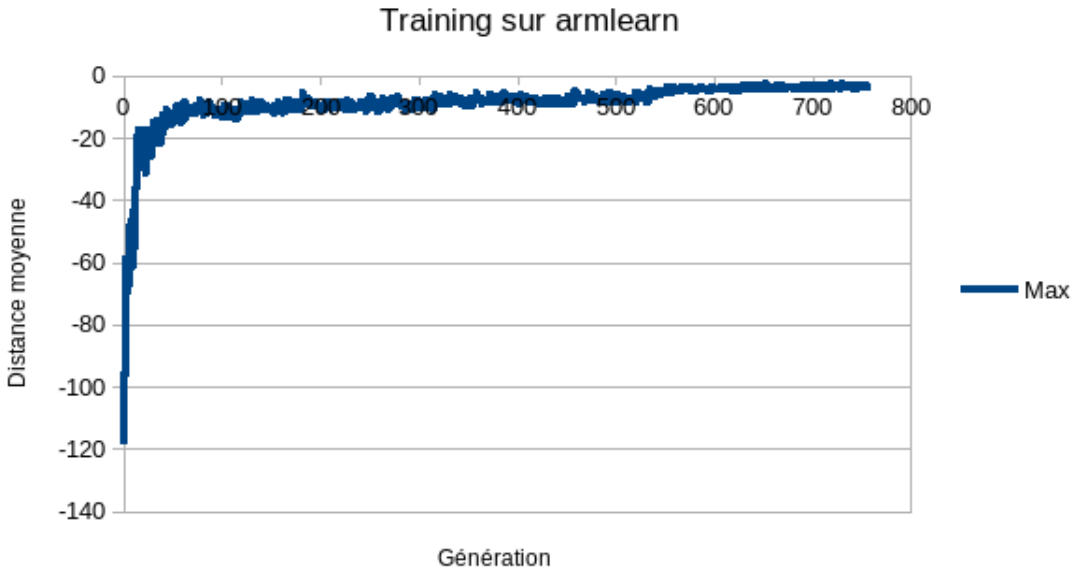

Figure 7: Evolution du score du meilleur individu sur Armlearn

Une fois un bon individu obtenu, 1000 cibles ont été générées au hasard afin d'évaluer celui-ci. La position atteinte par le bras robotique pour chaque cible a été conservée, permettant des statistiques et analyses. Selon le critère défini par Gael Gendron, à savoir que la tête du bras robotique doit s'approcher à une distance inférieure à 3,52 de la cible, près de 850 des 1000 cibles peuvent être considérées comme totalement résolues. Cependant, parmi les autres, on trouve des positions pour lesquelles le TPG ne parvient pas du tout à s'approcher convenablement. Il peut lui arriver de conserver par exemple une distance de

100 entre le bras et la cible. Après analyse, il semble qu'il s'agisse de points problématiques car pour les atteindre, l'agent a tendance à bloquer un moteur. En effet, la bibliothèque Armlearn propose de tronquer la consigne donnée à un moteur lorsque celle-ci dépasse les bornes. Par conséquent, si le TPG a placé le bras de telle manière à ce que l'un de ses moteurs soit presque au bout de course, il est possible qu'il se bloque à cette position.

Il est à noter que cet entraînement et ces résultats n'ont été effectués que sur une portion de l'espace, 1/4 environs (seulement les coordonnées positives), néanmoins tout porte à croire qu'ils pourraient sans mal se généraliser.

Pour visualiser les positions atteintes, les positions cibles, la trajectoire du bras ou autre un script R a été utilisé. Il est disponible sur l'espace owncloud.

Pour finir, une fois ce résultat obtenu, le TPG a pu être utilisé directement sur le bras robotisé dont dispose l'équipe. Une séquence de mouvements lui permettant de saisir un objet et de le déplacer a été construite manuellement, saisissant dans un programme un ensemble de coordonnées de positions. Ensuite, le TPG a calculé les instructions à donner à ses moteurs en fonction de cette consigne, avant de finalement mettre en pratique cette trajectoire. Le bras est parvenu à réaliser le déplacement indiqué sans trop d'écart visible, ce qui en fait une preuve sommaire mais concrète du bon apprentissage du TPG.

3.2 Améliorations

Des améliorations diverses de la bibliothèque Gegelati ont été effectuées. Certaines étaient relativement mineures mais trois d'entre elles peuvent être évoquées. Afin de les réaliser, il était nécessaire de respecter un protocole. En effet, il fallait d'abord travailler sans affecter la version principale de l'application (en créant une autre branche git), couvrir les nouvelles lignes de code avec des tests unitaires, les formater comme le reste du programme et modifier un Changelog enregistrant les changements. Ensuite, il fallait faire une demande pour fusionner ces modifications avec la version principale (une "pull request"). Comme évoqué, des tests notamment de construction et de qualité étaient ensuite automatiquement déclenchés. Pour finir, Karol Desnos relisait le code avant son intégration.

3.2.1 Parseur

Tout d'abord, afin de réaliser des apprentissages, le fonctionnement de Gegelati est régi par divers paramètres. En effet, le nombre de noeuds racines du graphe, les probabilités de mutation ou encore le nombre de générations sont autant de valeurs que l'utilisateur doit définir avant le début de l'entrainement. Il peut y avoir une quinzaine de paramètres. Auparavant, ces valeurs étaient définies dans le programme avec lequel l'utilisateur appelle Gegelati. Désormais, un parseur a été créé et permet de déléguer ces paramètres dans un fichier à part, au format JSON. Ainsi, le programme principal de l'utilisateur ne dédie qu'une ligne aux paramètres, ce qui peut le clarifier grandement.

3.2.2 Logger

Par ailleurs, un système de log a également été ajouté. Auparavant, pour obtenir des statistiques sur son apprentissage (numéro, de génération en cours, score du meilleur individu à chaque génération...), l'utilisateur devait prévoir dans son programme principal de traiter les résultats au fur et à mesure de l'apprentissage. Désormais, Gegelati permet, via un système de callback, d'afficher automatiquement des éléments. En effet, durant son exécution, Gegelati appelle automatiquement à certains moments de son exécution d'éventuels loggers.

L'utilisateur peut prévenir Gegelati qu'il souhaite utiliser un logger pré-écrit, mais il peut également définir le sien en créant un logger héritant d'une classe spécifique de Gegelati. Là aussi, l'amélioration permet à l'utilisateur de ne dédier que deux ou trois lignes de son programme principal aux loggers, alors qu'avant cela pouvait représenter typiquement une quinzaine de lignes.

3.2.3 Adversarial

Pour finir, la plus ambitieuse amélioration à laquelle mon stage a apporté une contribution est "l'adversarial". Si l'on se remémorer l'application du Tic-tac-toe, Gegelati réussissait à développer un agent capable de battre un adversaire aléatoire. Cela dit, contre un humain, cet agent n'était pas performant. En effet, battre un joueur aléatoire ne nécessite pas une stratégie poussée, et le TPG n'en avait pas développée une sans raison. Une idée pourrait être de lui faire affronter une intelligence artificielle plus complexe qu'un joueur aléatoire, mais cela demande à l'utilisateur de programmer des adversaires intelligents pour faire face au TPG. Cela peut sembler parfois absurde, puisque la raison pour laquelle l'utilisateur utilise Gegelati peut être qu'il n'a pas encore d'intelligence artificielle convenable pour son problème. L'idée alors mise en pratique a été de permettre aux noeuds racines du TPG de s'affronter entre eux. D'abord, tous les noeuds racines de chaque génération s'affrontaient entre eux et ceux obtenant les meilleurs scores étaient sélectionnés. Le principe est le suivant par exemple dans un 1v1 :

1. Pour chaque noeud racine, choisir un autre noeud racine aléatoirement (ça peut être lui-même)
2. Créer un "Job" avec ces deux noeuds
3. Retourner à l'étape 1 avec le même noeud racine pour atteindre le nombre de "Job" par individu voulu, avant de passer à une nouvelle racine
4. Pour chaque Job, l'évaluer nbIterationsParJob fois
5. Fusionner les scores obtenus pour avoir un score par noeud racine

Mais, chaque génération étant en grande partie composée d'individus peu performants, cela n'était pas pertinent. En effet, en mettant que dans une application comme le Tic-tac-toe, 5% des noeuds racines soient préservés et parmi les 95% autres 90% sont vraiment très mauvais, dans un 1v1 chaque root a alors 90% de chance d'affronter un très mauvais. Si l'on réliase 10 combats par noeud racine, cela représente tout de même 35% de chances de n'affronter que des très mauvais, et quand bien même 1 ou 2 bons noeuds racines sont affrontés statistiquement cela ne contribue pas beaucoup au score.Intuitivement, on peut se dire que si ce taux de 90% de mauvais ne change pas, par récurrence il y a peu de chance que l'on ait une amélioration au fil des générations.

L'idée que des noeuds racines s'affrontent systématiquement eux-mêmes a été évoquée. Ceci dit, cela pose des problèmes : si un noeud se bat lui-même, est-ce que cela signifie qu'il est bon ou mauvais ? Au final, un très bon noeud peut très bien se battre lui-même et un mauvais aussi. De même, les matchs pourraient être systématiques nuls.

Pour obtenir de bons résultats, un système de champions a été créé, permettant aux noeuds racines de chaque génération d'affronter les noeuds ayant survécu à la génération passée, preuve de leur performance. Cela fonctionne ainsi, par exemple en 1v1 :

1. Préparer une liste de listes de champions. Dans le cas du 1v1, chaque liste de champions n'aura qu'un champion mais en 1v1v1 par exemple il y en aura deux. La taille de la liste de listes sera calculée en fonction des paramètres définissant le nombre d'itérations voulues par noeud racine.
2. Pour chaque noeud racine, on l'associe à chacune des listes de champions de la liste de listes. On place le noeud à tous les endroits possibles de la liste de champions pour qu'elle ait une taille correspondant au nombre d'agents par simulation : dans le cas du 1v1, en supposant que l'on ait une liste de champions A et un noeud racine à évaluer R, on crée l'association R-A puis A-R. On recommence pour chaque liste de champion A de la liste de listes.
3. Créer un job avec ces associations
4. Pour chaque Job, l'évaluer nbIterationsParJob fois
5. Fusionner les scores obtenus pour avoir un score par noeud racine

Le but est qu'un noeud racine soit évalué aussi bien en premier joueur, qu'en second ou plus en fonction de l'environnement.

Cela a été implémenté, néanmoins les résultats sur le Tic-tac-toe n'ont pas été excellents. En effet, les meilleurs agents d'une génération donnée atteignaient facilement tous le même comportement, et ceux de la génération suivante développaient simplement un comportement spécifiquement bon contre les champions, puisque ceux-ci étaient semblables. Cependant, une autre application, le "stick-game" qui est une implémentation du jeu des bâtonnets de Fort Boyard, a été résolue grâce à ce mode. En effet, là où face à un joueur aléatoire le TPG parvenait à adopter un comportement légèrement intéressant au bout d'une centaine de générations, il a pu adopter la solution optimale en 45 générations avec l'adversarial. Cela se constate graphiquement sur la figure 8, en effet on y voit un score élevé qui s'immobilise à 0,5 au bout d'un moment. Cela signifie que plus aucun nouvel individu n'arrive à battre les champions, un score de 0,5 indiquant un match nul. Le fait que près de 400 générations s'écoulent sans éloignement de la courbe de 0,5 tend à montrer que la situation est stable et que Gegelati ne peut fournir de meilleur résultat.

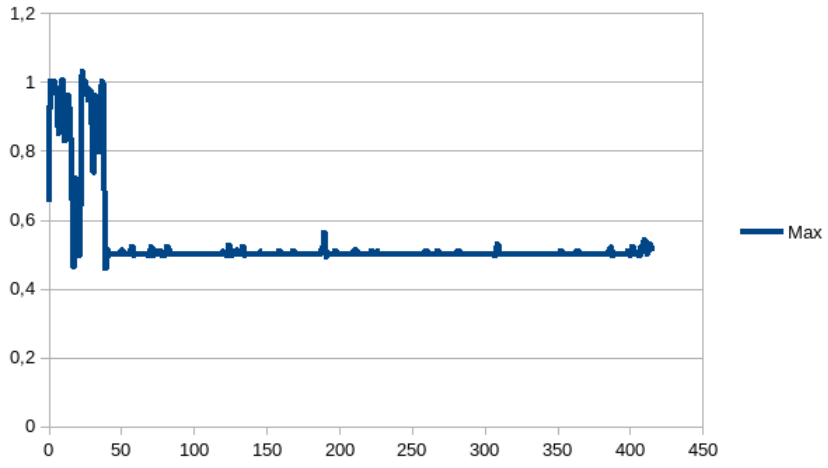

Figure 8: Evolution du score du meilleur individu sur stick-game

4 Conclusion

Ce stage de 3 mois a permis la réalisation de plusieurs éléments. Les 3 applications avec lesquelles j'ai pu utiliser Gegelati seront je pense utiles, en particulier l'ALE et Armlearn. La première permet d'effectuer des comparaisons avec le programme de référence de S. Kelly et la seconde semble des plus prometteuses, Gegelati fournissant de bons résultats et l'application ayant une utilité concrète. Par ailleurs, les améliorations et pistes d'améliorations que j'ai réalisées ont une réelle utilité dans l'utilisation de la bibliothèque. En conséquence, je pense avoir, bien que modestement, contribué à l'avancée de la bibliothèque.

De mon côté, j'ai pu acquérir une expérience et des connaissances nouvelles. Tout d'abord, la notion de TPG me semblant prometteuse, je suis satisfait d'y avoir été initié. En effet, cette technique possédant des avantages uniques, il est possible qu'un jour je sois amené à bénéficier de cette expérience. Il en va de même pour les algorithmes génétiques, notion avec laquelle je possédais déjà une expérience mais sur laquelle j'ai pu réfléchir abondamment durant mon stage. De manière plus générale cela m'a permis de me familiariser avec une technique d'apprentissage artificiel et, par conséquent, avec le concept d'apprentissage artificiel lui-même.

Par ailleurs, réaliser un parseur et un logger m'a ajouté une expérience utile dans ce domaine, ces deux éléments étant extrêmement fréquents au sein des projets informatiques d'envergure.

Utiliser git au quotidien, réalisant des branches plus ou moins complexes, tout en travaillant les tests unitaires de la bibliothèque et son intégration continue m'a aussi donné une expérience en génie logiciel.

Le dernier apport technique auquel on peut penser est le langage lui-même : ayant réalisé très peu de C++ durant ma formation, j'ai pu au travers de ce stage m'y familiariser de façon importante. J'ai notamment découvert de nombreuses astuces et obtenu de bons réflexes dans le langage lui-même, mais aussi dans l'utilitaire "cmake" qui permet leur construction et dans l'usage de la bibliothèque standard.

Outre tous ces gains, mon approche du monde de la recherche m'a permis de découvrir ce milieu et de bien mieux le comprendre. La situation sanitaire n'a pas rendu l'organisation d'équipe facilement observable mais l'aperçu que j'en ai eu m'a permis de mieux l'appréhender.

5 Références

[1] S. Kelly. "Scaling genetic programming to challenging reinforcement tasks through emergent modularity". PhD thesis. Dalhousie University, 2018.

[2] M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[3] G. Gendron. *armlearn*. `https://github.com/ggendro/armlearn`. 2019.