INSA Rennes - département EII

Stage dans l'équipe VAADER du laboratoire IETR

Rapport de stage 4A

Génération de code pour une bibliothèque d'apprentissage par renforcement

Étudiant :
BOURGOIN Thomas
Tuteur INSA :
COUSIN Jean-Gabriel

Encadrants :
DESNOS Karol
DARDAILLON Mickaël
SOURBIER Nicolas

Stage du 17/05/2021 au 27/08/2021

4 octobre 2021

Table des matières

1 Introduction

Ces dernières années, l'essor d'intelligences artificielles basées sur des réseaux de neurones a été possible grâce à l'accessibilité de machines avec de grandes capacités de calcul. Paradoxalement, de plus en plus de secteurs veulent pouvoir utiliser des intelligences artificielles (IA) sur des systèmes limités en puissance. On peut, par exemple, citer le secteur de l'internet des objets. Il serait donc intéressant d'avoir une architecture d'intelligence artificielle plus légère utilisable sur un système limité en ressources. Une des solutions existantes est l'utilisation d'IA basées sur les Tangle Program Graph (TPG). Ces IA utilisent un apprentissage par renforcement pour apprendre à interagir avec leur environnement. Dans le cas d'un apprentissage par renforcement, l'agent (ici un TPG) reçoit des données de l'environnement et détermine une action à transmettre à l'environnement. L'environnement interprète l'action ce qui modifie son état. Il envoie ensuite les nouvelles données à l'agent. Ce dernier reçoit aussi une récompense qui lui permet de déterminer si l'action choisie était bonne. L'objectif de l'agent est de maximiser la récompense. La bibliothèque C++ GEGELATI permet l'apprentissage et l'inférence de TPG.

Ce stage vise à implémenter la génération de code C dans une bibliothèque pour accélérer l'exécution d'un TPG sur des cibles embarquées. Ce stage a été effectué dans l'équipe VAADER du laboratoire L'Institut d'Électronique et des Technologies du numéRique (IETR). Cette équipe travaille principalement sur le traitement d'images et vidéo. Ils travaillent aussi sur la conception d'algorithmes optimisés pour des systèmes embarqués. Dans un premier temps, nous définirons un TPG, nous présenterons la bibliothèque GEGELATI et les objectifs du stage. Puis, nous étudierons plusieurs représentations d'un TPG en C. Dans

un troisième temps, nous verrons les algorithmes qui ont étés ajoutés dans la bibliothèque pour générer les fichiers. Enfin, la dernière section présente les résultats obtenus lors de ce stage. Elle contient une comparaison entre le temps d'exécution d'un TPG en utilisant le code généré avec le temps nécessaire pour réaliser l'inférence avec les fonctions internes à la bibliothèque.

2 Définition d'un Tangle Program Graph et présentation de GEGELATI

Cette section vise à introduire le concept d'un TPG et à présenter la bibliothèque qui est modifiée durant le stage. La dernière sous-section énumère les objectifs du stage.

2.1 Définition d'un Tangle Program Graph

Le concept de Tangle Program Graph (TPG) est introduit par Kelly dans [2]. Un TPG est une architecture d'intelligence artificielle sous la forme d'un graphe dirigé. Ce graphe se constitue d'équipes, d'actions et de programmes. Les actions correspondent aux feuilles du graphe. Les équipes correspondent à tous les autres nœuds du graphe qui ont des arêtes en sortie. La racine du TPG est l'équipe qui n'a pas d'arêtes en entrée. Les arêtes du graphe sont constituées d'un programme et d'un nœud de destination. Un programme correspond à une suite d'opérations issues d'un jeu d'instructions défini par l'utilisateur. Il prend en entrée les données transmises par l'environnement et retourne une valeur. Pour déterminer le prochain nœud à visiter, on choisit l'arête avec la plus grande valeur qui n'a pas encore été sélectionnée. Une inférence ou une exécution correspond au parcours du TPG en partant de la racine jusqu'à une action. La Figure 1 montre un exemple de TPG simple avec la racine en bas du graphe.

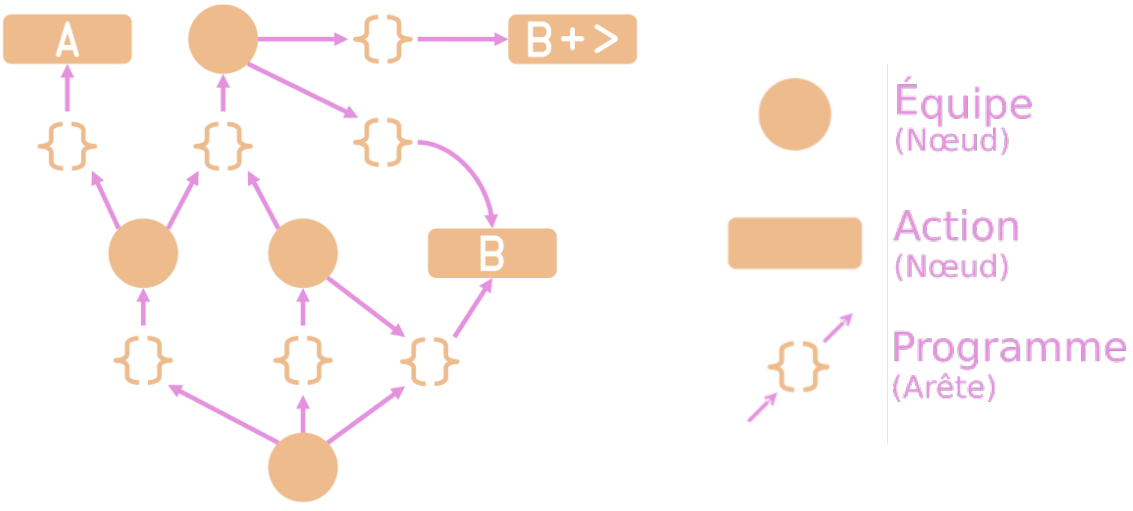


FIGURE 1 – Exemple de TPG avec une seule racine.

2.2 Présentation de GEGELATI

GEGELATI est une bibliothèque C++ open source introduite par Desnos et al. [1]. Elle permet de réaliser l'apprentissage et l'exécution de TPGs. Les paramètres d'apprentissage sont stockés dans un fichier JSON. GEGELATI permet aussi d'importer et d'exporter un

TPG dans un fichier au format dot. Les principales contraintes de développement sont d'avoir un code parallèle et déterministe. Puisque GEGELATI est développée en C++, la bibliothèque peut être déployée sur de nombreuses architectures.

Desnos et al. montrent aussi dans [1] que plus les programmes du TPG ont un jeu d'instructions riche, plus les TPG appris seront bons pour interagir avec leur environnement. C'est pourquoi, GEGELATI propose à l'utilisateur de créer son propre jeu d'instructions. Pour cela, le développeur pourra utiliser les lambdas instructions du C++. La Figure 2 montre l'interaction entre GEGELATI, le TPG et l'environnement. On peut voir sur cette figure que GEGELATI sert d'intermédiaire et permet la communication entre le TPG et l'environnement. La bibliothèque utilise des mécanismes d'abstraction afin de pouvoir manipuler le type (entier, flottant, double précision...) et la structure (tableau ou matrice) des données transmises par l'environnement. Ces mécanismes utilisent une arborescence complexe de classes afin de représenter plusieurs structures de données. Le type de la source de données est défini à l'aide d'un template. Les opérandes d'une instruction utilisent rarement l'ensemble des données fournies par l'environnement. C'est pourquoi, il est nécessaire de pouvoir extraire des données fournies par l'environnement pour ensuite réaliser des calculs avec. L'extraction des données nécessite de vérifier que le type de la donnée fournie par l'environnement est compatible avec le type de l'opérande qui souhaite l'utiliser. Si les types sont compatibles, on peut alors calculer l'adresse où se trouve la donnée et l'extraire. Ces opérations doivent être réalisées lors de l'inférence du TPG car des mutations peuvent modifier de nombreux paramètres dont la source de données et l'adresse utilisée.

FIGURE 2 – Interaction entre GEGELATI, le TPG et l'environnement.

Les mécanismes d'abstraction sont nécessaires pour l'apprentissage et afin d'avoir une bibliothèque générique. Cependant ces structures de données ne sont plus indispensables lorsque l'on souhaite déployer le TPG après l'apprentissage. Ces structures ajoutent de la complexité lors de l'inférence du TPG sur des architectures embarquées. C'est pourquoi, en les supprimant, on peut espérer obtenir de meilleures performances lors de l'inférence surtout dans le cas d'une architecture limitée en ressources.

2.3 Objectifs du stage

Le principal objectif de la génération de code est d'accélérer l'inférence du TPG lorsque l'apprentissage est terminé et que sa structure est figée. Elle sera très intéressante dans le cas où la cible qui exécute le TPG est limitée en ressource de calcul ou en mémoire. On peut aussi espérer des gains en performances sur des cibles plus puissantes telles que des ordinateurs de bureau en x86. Le deuxième intérêt de la génération de code est de pouvoir réaliser l'inférence sans avoir besoin de déployer GEGELATI sur la cible embarquée. En effet, le code généré doit être capable de représenter et d'exécuter le TPG en s'affranchissant des structures de données de la bibliothèque. Grâce à cela, l'exécutable n'aura pas besoin de GEGELATI ce qui réduira son empreinte mémoire sur la cible.

Un objectif secondaire de la génération de code est de pouvoir l'utiliser lors de l'apprentissage. Si la génération de code est suffisamment rapide et si elle offre de réels gains en performance sur des cibles x86, alors on pourrait envisager de l'utiliser pour réaliser les inférences lors de l'apprentissage du TPG. Dans ce cas là, il faudra comparer le temps nécessaire pour générer les fichiers, les compiler et les exécuter au temps requis pour faire l'inférence déjà implémentée dans la bibliothèque.

Un autre objectif du stage était une contrainte sur l'implémentation de la génération du code dans GEGELATI. En effet, mes encadrants souhaitent que la génération du code soit ajoutée sous la forme d'un "module" dans la bibliothèque. La bibliothèque doit pouvoir être installée avec ou sans la génération de code en fonction de ce que l'utilisateur décide. La principale raison derrière cette contrainte est de pouvoir réduire sa taille si l'utilisateur n'a pas besoin de générer de code. On peut imaginer que l'utilisateur souhaite apprendre son TPG sur le système embarqué directement et qu'il n'aura jamais besoin de générer de code. Pour éviter de surcharger la mémoire programme de son système, le développeur peut donc compiler la bibliothèque sans la génération de code et réduire l'empreinte mémoire de son exécutable.

La Figure 3 montre comment la génération de code sera utilisée dans la bibliothèque. Lorsque le TPG a fini d'apprendre à interagir avec son environnement, il sera possible de générer un code C qui contiendra la représentation du TPG. Ce code C possédera aussi les fonctions requises pour exécuter le TPG ce qui lui permettra d'interagir directement avec l'environnement sans avoir besoin de GEGELATI. Le langage C a été choisi par mes encadrants car c'est un langage couramment utilisé pour les systèmes embarqués pour sa capacité à gérer la mémoire et sa portabilité.

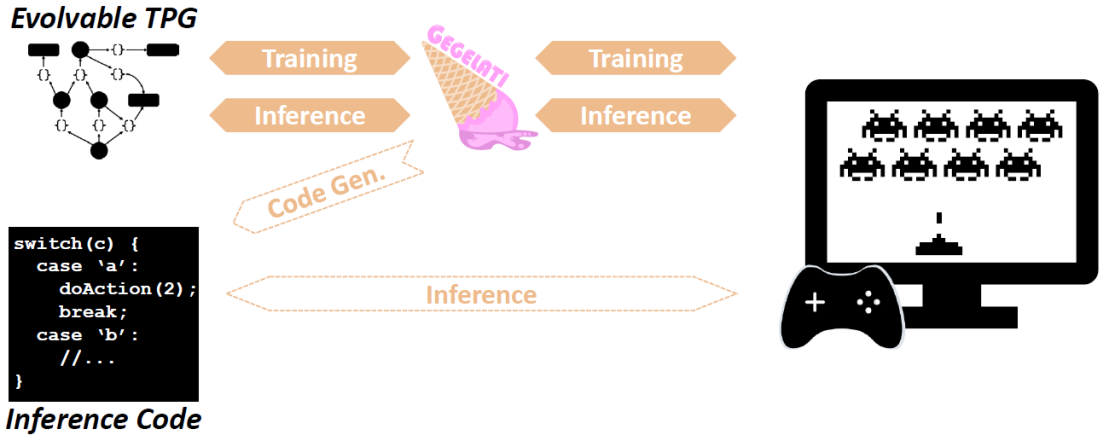


FIGURE 3 – Interaction de la génération de code dans le fonctionnement de GEGELATI. Lorsque l'apprentissage est terminé, du code peut être généré pour interagir avec l'environnement et réaliser l'inférence du TPG.

Le code inclus dans le projet GEGELATI doit satisfaire des contraintes en termes de qualité. La bibliothèque possède des tests unitaires avec une couverture de code de 100%. Le projet utilise l'intégration continue pour vérifier la compatibilité de la bibliothèque pour trois compilateurs. Les compilateurs testés sont Clang pour Mac, GCC pour Linux et MSVC pour Windows. Le code produit lors de ce stage doit donc être couvert à 100%. De plus, la génération de code et ses tests doivent pouvoir être exécutés sur les trois systèmes d'exploitation avec les compilateurs énoncés.

3 Étude de la représentation d'un TPG en C

Cette section présente la méthodologie utilisée pour représenter un TPG et satisfaire les objectifs du stage précédemment énoncés. Plusieurs solutions sont proposées, elles sont ensuite comparées pour déterminer laquelle sera la plus adéquate. La dernière sous-section introduit la structure et le contenu des fichiers qui seront générés.

3.1 Contenu des éléments d'un TPG

Avant d'étudier des représentations de TPG, il faut commencer par déterminer le contenu des éléments du TPG. Une équipe peut être vue comme un ensemble d'arêtes sortantes. Une arête doit contenir un programme, un nœud de destination et une variable pour déterminer si elle a déjà été visitée. La destination est soit une équipe soit une action. Dans GEGELATI, les actions renvoyées par l'agent sont représentées par des entiers. Il faudra donc sauvegarder cet entier et pouvoir le transmettre à l'environnement lorsque l'inférence est terminée. Il faut aussi que les programmes des arêtes du TPG puissent avoir accès aux données transmises par l'environnement. Au cours d'une réunion avec mes encadrants, il a été décidé que ces données seraient transmises grâce à des variables globales.

3.2 Représentation des éléments d'un TPG

Maintenant que l'on a défini ce que doit contenir chaque élément d'un TPG, on peut proposer plusieurs structures de données pour les représenter. Les structures de données que nous avons essayées peuvent être découpées en deux catégories différentes. Nous avons utilisé le TPG de la Figure 4 pour comparer les représentations. Il nous a servi de témoin pour vérifier la faisabilité de chaque solution et il nous a permis d'avoir un aperçu du code à générer.

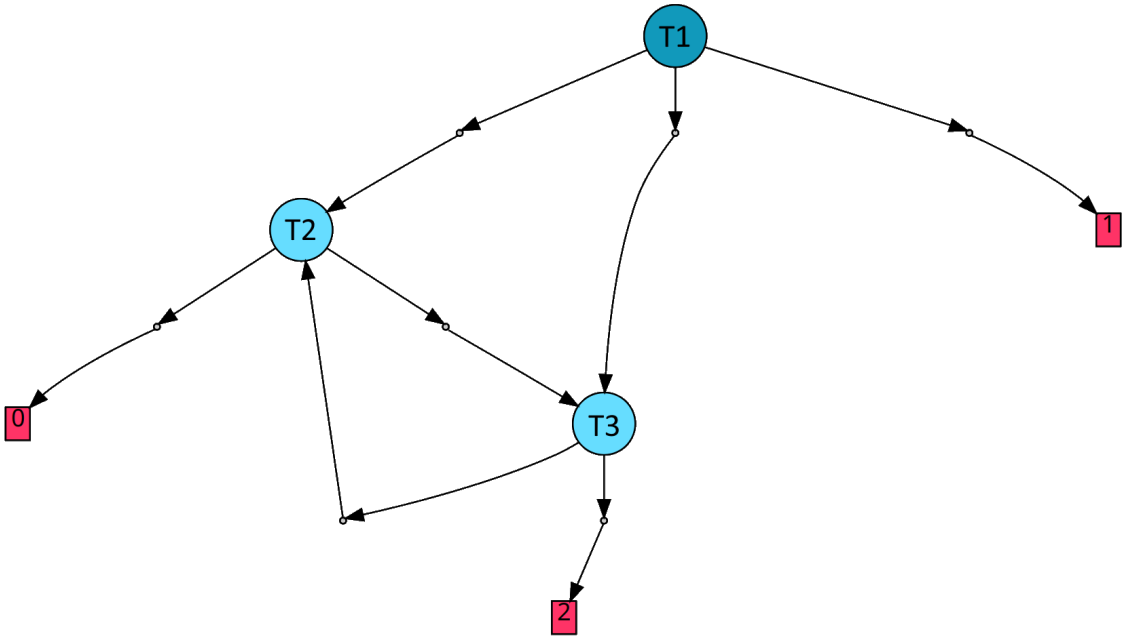


FIGURE 4 – TPG utilisé pour tester les différentes représentations.

La première catégorie contient des solutions où l'on utilise des structures de données pour

représenter le graphe d'un TPG. Ensuite pour réaliser l'inférence, on utilise des fonctions qui permettent de parcourir le graphe. Nous avons testé des solutions qui utilisent des listes chaînées et des tableaux dynamiques pour représenter le TPG. Cependant le fait qu'elles utilisent de l'allocation dynamique les a rendues non viables pour des systèmes embarqués. De plus, la structure du TPG est fixée, il n'y a pas besoin d'avoir une structure de données capable de modifier sa représentation. Nous avons donc cherché des structures de données statiques pour éviter des problèmes liés à l'allocation de mémoire sur des systèmes limités en ressources. La structure Compressed Sparse Row (CSR) présentée par T.Kelly dans [3] permet de représenter un graphe avec deux tableaux statiques.

La deuxième catégorie contient des solutions où le graphe du TPG est directement représenté par le code source généré. Chaque nœud du TPG possède sa propre fonction. Pour réaliser l'inférence avec ces solutions, il faut exécuter la fonction qui représente la racine du TPG. Ces méthodes se basent sur l'utilisation de pointeurs de fonction pour parcourir le graphe. L'avantage de ces méthodes est que l'utilisateur du code généré n'a pas besoin d'instancier le TPG. En effet, lorsqu'il voudra lier le TPG à l'environnement, il aura seulement besoin d'ajouter le code généré à son projet et appeler la fonction qui réalise l'inférence. De plus, ces méthodes évitent de devoir créer des fonctions qui sont seulement utilisées pour l'instanciation du TPG.

Les quelques avantages énoncés plus haut nous ont fait choisir une solution issue de la seconde catégorie. Cependant, le choix était difficile car on ne pouvait pas mesurer rapidement si la méthode CSR était plus efficace qu'une solution avec des pointeurs de fonction. Il aurait fallu faire deux générations de code pour comparer les performances sur des TPGs plus gros.

3.3 Utilisation de pointeurs de fonctions

Puisque la génération de code va utiliser des fonctions pour représenter le TPG, cette sous partie va présenter plus en détails les prototypes de fonctions et les structures de données utilisées. Une fonction représentant une équipe contient un tableau statique de ses arêtes sortantes. Si la fonction représente un nœud qui est une action, alors il n'y a pas de tableau puisque les actions sont les feuilles du graphe.

La première solution que nous avons envisagée utilisait les prototypes de fonction de la Figure 5.

```
int T1(){... return prochainNoeud();}
int A1(){return 1;}
```

FIGURE 5 – Définitions des fonctions des nœuds qui retournent l'action à transmettre à l'environnement.

Avec cette solution, si le nœud est une équipe, alors il exécute la fonction du prochain nœud. Si le nœud est une action, alors la fonction renvoie un entier. L'inférence correspond donc à exécuter la racine du graphe qui appellera les nœuds suivant jusqu'à une action. La Figure 7 (a) montre la pile d'appel pour cette solution. On observe très clairement que la pile peut vite devenir très grande si le graphe du TPG est profond. Cette solution n'est donc pas envisageable sur des systèmes embarqués.

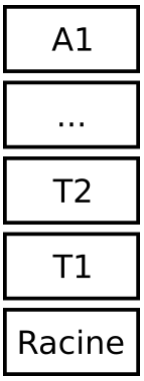
Pour régler ce problème de taille de pile d'appel, une solution est de retourner le pointeur de fonction du prochain nœud à exécuter au lieu de l'exécuter. L'action à transmettre à

l'environnement est donc passé par pointeur dans les arguments des fonctions des nœuds. Le type de retour des fonctions des nœuds doit être capable de retourner un pointeur sur une fonction de nœud. Mais pour déclarer un pointeur de fonction, il faut connaître le type de retour de la fonction pointée. On obtient donc une boucle infinie. Le seul moyen pour la briser est d'utiliser un pointeur générique : **void***. On obtient donc les définitions de la Figure 6.

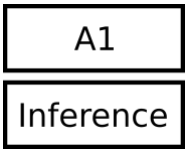
```
void* T1(int* act){... return ptrNoeud;}
void* A1(int* act){*act = 1; return NULL;}
```

FIGURE 6 – Définitions des fonctions des nœuds qui retournent le pointeur de la prochaine fonction à exécuter.

Avec cette solution, il faut une méthode qui permet de gérer l'inférence en exécutant les fonctions de chaque nœud. La Figure 7 (b) montre la pile lorsque l'on exécute la fonction du dernier nœud (l'action). On observe sur cette figure que seulement la fonction du nœud de l'action est dans la pile. En effet, entre chaque appel de fonction de nœud, la fonction précédente est dépilée avant d'exécuter la suivante. La pile d'appels est donc bornée indépendamment du nombre de nœuds exécutés.



(a) 1ère solution : L'équipe exécute la fonction du prochain nœud jusqu'à atteindre une action.



(b) 2ème solution : l'équipe retourne le pointeur sur la fonction à exécuter ensuite. La fonction Inference s'occupe d'appeler les fonctions des nœuds.

FIGURE 7 – Pile d'appels pour les deux solutions avec des pointeurs de fonction lorsque l'on exécute le dernier nœud du TPG.

Pour représenter une arête du TPG, il suffit de créer un **struct** qui regroupe les trois informations évoquées dans la sous-section 3.1. La variable qui indique si l'on a déjà suivi cette arête est un entier. Le nœud de destination de l'arête est stocké sous la forme d'un pointeur sur la fonction du nœud de destination. La valeur retournée par le programme de l'arête est un **double**. Ce programme est représenté par une fonction qui a accès aux variables globales des données de l'environnement et n'a pas besoin d'arguments. Le dernier élément de l'arête est donc un pointeur sur la fonction qui correspond à son programme. Le code de la Figure 8 permet de représenter une arête d'un TPG.

3.4 Contenu des fichiers générés

La sous-section 3.3 a présenté les structures de données et les prototypes des fonctions qui sont utilisées dans le code généré. Le code généré est réparti en 4 fichiers différents. Une première paire de fichiers avec un fichier source et son fichier d'en-tête qui contient

```
typedef struct Edge {
    int visited;
    void* (*ptr_vertex)(int* action);
    double (*ptr_prog)(void);
}Edge;
```

FIGURE 8 – Struct contenant les informations d'une arête.

les fonctions représentant les nœuds du TPG. Ces fichiers contiennent aussi les fonctions nécessaires pour réaliser l'inférence du TPG et réinitialiser les variables indiquant si une arête a été visitée. Le fichier d'en-tête contient aussi la définition de la structure d'une arête. La seconde paire de fichiers avec un fichier source et son en-tête contient la définition et la déclaration des fonctions représentant les programmes des arêtes du TPG.

Le code généré peut avoir besoin d'accéder à des fonctions ou des constantes définies dans d'autres fichiers d'en-tête. Pour éviter que l'utilisateur doive modifier les fichiers générés, le fichier d'en-tête *externHeader.h* est inclus dans les fichiers générés. L'utilisateur de la bibliothèque peut ajouter les fichiers d'en-tête nécessaires à la compilation du code généré dans ce fichier. Comme il n'est pas édité par la génération de code, les modifications de l'utilisateur sont conservées même si le code est de nouveau généré. Grâce à cela l'utilisateur n'a pas besoin d'inclure manuellement les fichiers d'en-tête après chaque génération de code.

4 Implémentation de la génération de code dans GEGELATI

La section précédente a introduit les structures de données qui seront utilisées pour représenter un TPG. Elle a aussi défini le contenu des 4 fichiers à générer. Dans cette section, nous allons voir le code et les algorithmes ajoutés à la bibliothèque qui permettent de générer les fichiers. Ce code respecte les contraintes de développement présentées dans la sous-section 2.3. L'une des contraintes de développement est de pouvoir compiler la bibliothèque sans la génération de code. Pour cela, le code et les tests spécifiques à la génération sont encadrés par des directives **#ifdef** CODE_GENERATION.

4.1 Représentation des instructions

GEGELATI permet à son utilisateur de créer ses propres instructions à l'aide des lambdas instructions incluses dans la norme C++. Les opérandes de ces instructions peuvent être des tableaux ou des matrices. Grâce à cela, l'utilisateur peut, par exemple, créer des instructions qui appliquent un filtre sur une image ou réalisent des convolutions. La Figure 9 donne plusieurs exemples de lambda instructions qui peuvent être utilisées dans un jeu d'instructions. La première fonction est une addition. Le deuxième exemple permet de calculer un modulo en utilisant cmath. La troisième lambda instruction calcul le déterminant d'une matrice 2×2.

Afin de savoir comment une instruction doit être générée, nous utilisons une chaîne de caractères pour stocker un format. Le format correspond à du code C qui permet de réaliser la même opération que l'instruction définie avec la lambda instruction. Les opérandes de l'instruction sont représentés par le caractère $ suivis d'un nombre. Le résultat de l'opération est représenté par les caractères $0. Le format étant du code C où l'on remplace

```
[](double a, double b)->double {return a + b; }
[](double a, double b)->double {
    if (b != 0.0) { return fmod(a, b);}
    else { return  DBL_MIN; }}
[](const double a[2][2]) -> double {
    return a[0][0] * a[1][1] - a[0][1] * a[1][0]; }
```

FIGURE 9 – Exemples de lambdas instructions qui peuvent être utilisées dans les programmes d'un TPG.

les variables par des $, l'utilisateur peut utiliser des opérateurs ternaires, appeler des fonctions, utiliser des constantes d'une autre bibliothèque, etc. La Figure 10 donne un exemple de format pour chaque lambda instruction de la Figure 9.

```
$0 = $1 + $2;
$0 = (($2) != 0.0) ? fmod($1, $2) : DBL_MIN ;
$0 = $1[0][0] * $1[1][1] - $1[0][1] * $1[1][0];
```

FIGURE 10 – Format permettant de générer les instructions de la Figure 9.

4.2 Génération d'un programme d'une arête de TPG

En utilisant la chaîne de caractères de la sous-section 4.1, il possible de créer une fonction qui reproduit le comportement d'un programme d'une arête. Afin de pouvoir gérer les opérandes qui sont des tableaux ou des matrices, le code généré utilise des variables intermédiaires. Elles permettent de séparer la déclaration et l'instanciation des opérandes de leur utilisation dans l'instruction. Pour chaque instruction d'un programme, les opérandes sont déclarés et instanciés avec les variables qui accèdent aux données de l'environnement. Puis, grâce à une expression régulière, les caractères $ suivis d'un nombre sont remplacés par les opérandes adéquates. Chaque programme possède une file de registres représentée par un tableau de **double**. La valeur de retour du programme correspond au registre 0. La Figure 11 montre un exemple d'une instruction générée. Cette instruction calcul le déterminant d'une matrice 2×2 extrait d'une matrice 5×5 (in2). Le résultat de ce calcul est ensuite stocké dans le registre 2. La Figure 12 montre la position de la matrice 2×2 dans la source de données in2.

```
double op0[2][2] = {{in2[5], in2[6]}, {in2[10], in2[11]}};
reg[2] = op0[0][0] *  op0[1][1] - op0[0][1] * op0[1][0];
```

FIGURE 11 – Code généré permettant de calculer le déterminant d'une matrice 2×2 extrait d'une matrice 5×5.

4.3 Génération d'un TPG

La génération du code permettant de représenter un TPG se fait en générant chacun de ses nœuds un par un. Si le nœud est une équipe, alors on crée un tableau statique avec les arêtes en sortie de l'équipe. Il suffit ensuite d'écrire l'appel de la fonction qui permet d'exécuter une équipe. Lors de la génération du tableau d'arête, on vérifie si la fonction qui correspond au programme de chaque arête existe. Si ce n'est pas le cas, la fonction est ajoutée dans les fichiers qui contiennent les programmes du TPG. Une fonction

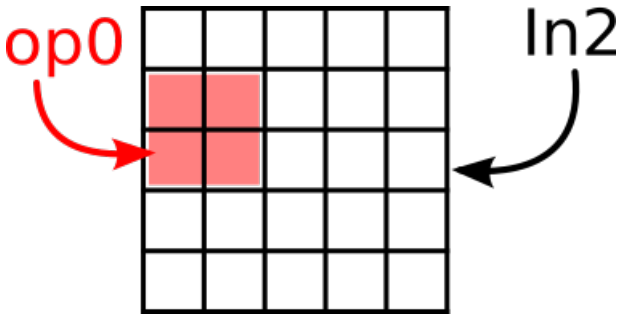


FIGURE 12 – Schéma de l'extraction à partir de la source in2 de l'opérande op0 de la Figure 11.

représentant une équipe a un code similaire à la Figure 13. Si le nœud est une action, alors la fonction modifie la valeur pointée par l'argument action et renvoie NULL car il n'y a pas de prochain nœud.

```
void* T1(int* action){
    static Edge e[] = {
        {0,P1,T2},
        {0,P2,A1}
    };
    int nbEdge = 2;
    return executeTeam(e, nbEdge);
}
```

FIGURE 13 – Exemple de code généré représentant l'équipe T1.

Lors de l'implémentation de la génération des TPGs, nous avons réutilisé certains algorithmes déjà présents dans la bibliothèque. Cependant, il n'était pas logique de faire hériter les classes de la génération de code des classes qui contenaient ces algorithmes. Pour éviter de dupliquer du code, nous avons créé des classes abstraites qui regroupent les algorithmes génériques. Puis, on hérite les classes spécifiques à la génération de code de ces classes abstraites. La Figure 14 montre le diagramme UML résultant de cette modification.

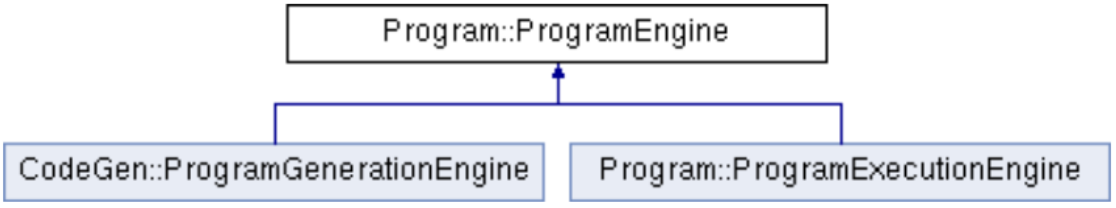


FIGURE 14 – Diagramme UML avec une classe abstraite qui regroupe les algorithmes utiles à plusieurs classes.

4.4 Qualité du code

Pour tester la génération de code pendant le développement, nous avons créé des TPGs basiques avec les fonctions de la bibliothèque. La Figure 15 montre une partie des TPGs utilisés. Ils nous ont permis de vérifier que les TPGs et les programmes étaient générés correctement. Chaque TPG vérifie une fonctionnalité précise. Ces TPGs ont ensuite été

intégrés dans les tests unitaires de la bibliothèque. Pour vérifier que le code généré est correct, on utilise des scripts shell et bash qui nous permettent de compiler le code. Il est ensuite possible d'exécuter le code généré avec des valeurs prédéfinies dans la routine de test. Si le TPG généré peut renvoyer plusieurs actions en fonction des données transmises, on utilise un fichier CSV afin de toutes les tester. Ces tests unitaires permettent de vérifier les fonctionnalités basiques des TPGs générés.

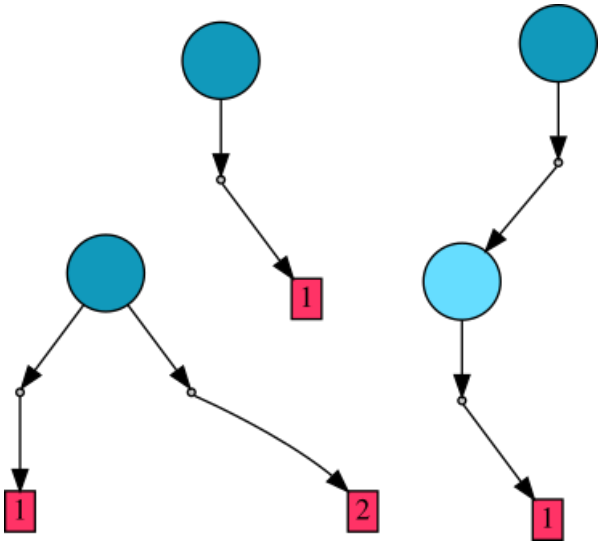


FIGURE 15 – Exemples de TPGs utilisés pour les tests unitaires de la génération de code.

Nous avons aussi mis en place des tests avec des TPGs plus complexes afin de comparer l'inférence en utilisant le code C++ de la bibliothèque et le code généré. Pour obtenir ces TPGs, nous avons pu utiliser les exemples des jeux stick-game, tic-tac-toe et pendulum du dépôt gegelati-apps [1]. GEGELATI permet d'importer des TPGs au format .dot si l'on connait le jeu d'instructions, la taille et le type des données transmises par l'environnement. L'ensemble de ces informations sont présentes dans le code source des exemples, nous avons donc pu importer des TPGs déjà appris. La préparation du test consiste à importer un TPG au format .dot puis utiliser la génération de code. Les fichiers générés sont ensuite compilés afin de créer un exécutable qui prend en argument les données de l'environnement. Le test se constitue d'une succession d'inférences jusqu'à atteindre la fin d'une partie du jeu. A chaque fois qu'une inférence est nécessaire, le code généré est lancé avec les données de l'environnement d'une part et le TPG est exécuté avec les fonctions internes à GEGELATI d'autre part. Les résultats des deux inférences sont comparés. Si ils sont différents, alors le test s'arrête. Sinon, on envoie l'action à l'environnement qui transmet de nouvelles données et on recommence. Ce test est important car il assure à l'utilisateur que l'exécution d'un TPG avec le code généré sera identique à l'exécution avec les fonctions de la bibliothèque.

La Figure 16 montre les résultats de l'intégration continue. On peut voir que les 3 compilateurs GCC, Clang et MSVC sont capables de compiler la bibliothèque. Cette figure montre aussi le taux de couverture des nouvelles lignes de code qui est de 100%.

1. `https://github.com/gegelati/gegelati-apps`

FIGURE 16 – Résultats de l'intégration continue sur GCC, Clang et MSVC et de la couverture de test.

5 Résultats

Cette dernière section résume les principales fonctionnalités de la génération de code. Elle contient aussi une comparaison des performances entre l'inférence avec les fonctions de la bibliothèque et l'inférence avec le code généré.

5.1 Fonctionnalités de la génération de code

La génération de code est capable de produire 4 fichiers contenant le code C équivalent d'un TPG. Il est possible d'importer un TPG et de générer son code C si l'on connaît la taille et le type des informations transmises par l'environnement et le jeu d'instruction que ses programmes utilisent. La compilation des fichiers nécessite un fichier d'en-tête supplémentaire qui permet à l'utilisateur d'inclure ses propres fichiers d'en-tête. Grâce à ce fichier, il est possible de générer des programmes issus d'arêtes d'un TPG qui utilisent des fonctions définies par l'utilisateur ou qui proviennent d'une autre bibliothèque, par exemple math.h. La génération des programmes des arêtes du TPG prend en charge les instructions plus complexes qui utilisent des tableaux ou des matrices.

Il est possible de lier le TPG généré à l'environnement d'apprentissage. Pour cela, il suffit de faire pointer les variables globales sur les données de l'environnement qui doivent être transmises aux programmes du TPG. Cela nous a permis de compléter les exemples stick-game, tic-tac-toe et pendulum en ajoutant la génération d'un TPG et l'interaction de ce TPG avec son environnement.

5.2 Mesure des performances

La possibilité de faire interagir le code généré avec son environnement nous a permis de mesurer les performances du code généré. Pour cela, nous avons comparé le temps nécessaire pour l'inférence d'un TPG en utilisant les fonctions de la bibliothèque avec le temps mis pour exécuter le code généré. Nous avons utilisé la bibliothèque ctime pour réaliser les relevés. Les mesures ont été réalisées avec l'exemple de stick-game de gegelati-apps. Comme une inférence est très rapide indépendamment des méthodes, il est nécessaire de faire énormément d'inférences pour exhiber une différence de temps de calcul. Les mesures ont été effectuées en réalisant 1 000 000 de parties stick game. Cela représente 9 000 000 d'inférences au total. Les mesures ont été répétées 10 fois et nous avons pris la moyenne des 10 temps. Les tests ont été réalisés sur deux architectures différentes : une machine virtuelle Linux avec un processeur I7-6700HQ cadencé à 3.10GHz et sur un processeur Mediatek Helio P20 cadencé à 2.30GHz basé sur un processeur ARM cortex A53.

Les résultats sont visibles sur le diagramme de la Figure 17. Le diagramme utilise une échelle logarithmique pour afficher les résultats du tableau 1. Pour les deux architectures

testées, les mesures indiquent que l'inférence avec le code généré est environ 45 fois plus rapide que l'inférence avec GEGELATI.

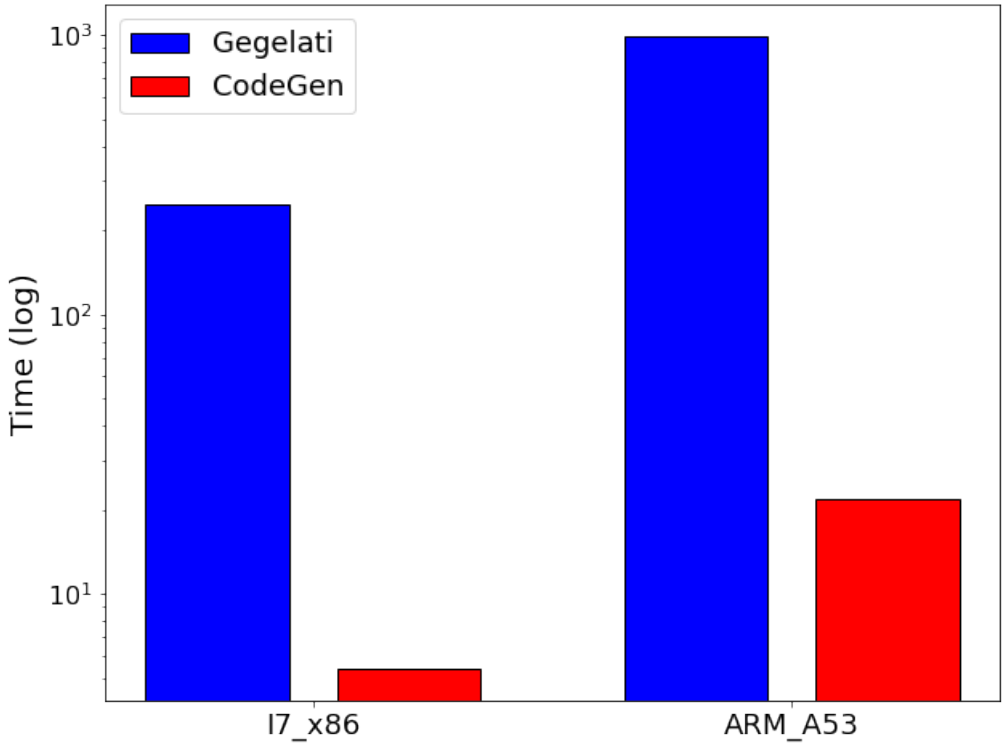


FIGURE 17 – Graphique des temps d'inférences en secondes pour 1 000 000 de parties de stick-game avec échelle logarithmique.

	GEGELATI	code généré	gain
I7-6700HQ	246 ≈ 4 min	5.4	45.5×
cortex A53	993 ≈ 16.5 min	22	45.1×

TABLE 1 – Comparaison des performances des méthodes d'inférences en secondes pour 1 000 000 parties de stick-game.

6 Conclusion

Pour conclure, la génération du code satisfait les objectifs du stage. Elle permet de générer automatiquement le code C équivalent d'un TPG. La représentation du TPG avec des pointeurs de fonction a été choisie suite à l'étude de diverses solutions. Les modifications apportées à la bibliothèque GEGELATI ont été faites en respectant les contraintes de développement. Les nouvelles lignes de code sont couvertes à 100% et peuvent être compilées avec les 3 compilateurs supportés par la bibliothèque. De plus, la compilation du code spécifique à la génération du TPG est désactivable grâce à une variable dans le cmake. Les mesures de performances indiquent que l'inférence avec le code généré est 45 fois plus rapide que l'inférence avec la bibliothèque. Les prochains objectifs sont de réaliser plus de mesures et de proposer des optimisations du code généré. Quelques idées ont déjà été émises comme paralléliser l'exécution des programmes des arêtes ou les regrouper pour utiliser les caractéristiques d'un processeur superscalaire.

Acronymes

CSR Compressed Sparse Row. 6

IA intelligence artificielle. 1

TPG Tangle Program Graph. 1–15

Glossaire

équipe nœud d'un TPG comportant une ou plusieurs arêtes en sorties. 2, 5–7, 9, 10

action Feuille d'un TPG. 1, 2, 5–7, 10, 11

Références

[1] Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, and Maxime Pelcat. Gegelati : Lightweight artificial intelligence through generic and evolvable tangled program graphs. In *Workshop on Design and Architectures for Signal and Image Processing (14th edition)*, pages 35–43, 2021.

[2] Stephen Kelly. Scaling genetic programming to challenging reinforcement tasks through emergent modularity. 2018.

[3] Terence Kelly. Programming workbench compressedd sparse row format for representing graphs. *Usenix*, Winter 2020 :76–82, 2020.

Thomas BOURGOIN

Laboratoire IETR, Équipe Vaader
Rennes (35700)

Mai à Août 2021

Génération de code pour une bibliothèque d'apprentissage par renforcement

Mots clés : TPG, GEGELATI, génération de code, pointeur de fonction

Les TPGs sont une architecture d'intelligence artificielle légère comparée à des réseaux de neurones classiques. Cette propriété les rend intéressants pour des systèmes embarqués. La bibliothèque GEGELATI permet l'apprentissage et l'inférence de TPGs en C++. Elle utilise des structures de données abstraites afin de pouvoir réaliser l'apprentissage du TPG mais elles ajoutent de la complexité lors de l'inférence. Cependant, elles ne sont plus nécessaires lorsque la structure du TPG est figée. Ce stage vise à ajouter un module de génération de code dans la bibliothèque GEGELATI. Le code est généré en C et doit s'affranchir des structures de données de la bibliothèque afin d'obtenir des meilleures performances. Tout d'abord plusieurs représentations de TPGs en C sont étudiées. La représentation avec des pointeurs de fonction a été choisie pour sa simplicité d'utilisation. Puis, les algorithmes ajoutés à la bibliothèque qui permettent de générer les fichiers sont présentés. Enfin, des comparaisons de performances indiquent que le code généré est 45 fois plus rapide que l'inférence avec la bibliothèque.

Code generation for a Reinforcement Learning Library

Keyword : TPG, GEGELATI, code generation, function pointer

A TPG is an artificial intelligence architecture lighter than conventional neural networks. This property makes them interesting for embedded systems. The GEGELATI library provides a framework to learn and execute a TPG in C++. It needs abstract data structures to learn a TPG but they add overhead during the inference. These data structures are not required when the graph of the TPG is set. The goal of this internship is to add code generation in the GEGELATI library. The C generated code must be independent of the data structure of the library to improve performance. At first, several representations of a TPG in C are studied. A representation with function pointers is chosen for its ease of use. Then the algorithms added to the library that generates the files are presented. Eventually, some performance metrics are given. They indicate that the generated code is approximately 45 times faster than the code inside the library.