Rapport de projet : CoreWar

Amand Henry Théo Sicot Etienne Bossu
Tom Rousée

12 avril 2024

Table des matières

1 Introduction

Le **Corewar** est un jeu de programmation créé en 1984 dans les universités américaines où deux programmes sont opposés pour prendre le contrôle d'une machine virtuelle appelée **MARS** (Memory Array Redcode Simulator). Ces programmes sont écrits dans un langage proche de l'assembleur appelé **RedCode**. Les programmes, appelés "Guerriers", ont pour objectif d'être les derniers à s'exécuter en faisant se terminer toutes les instances du programme adverse.

Au début les joueurs concevaient leurs programmes par eux mêmes mais avec la popularisation des **algorithmes génétiques**, beaucoup utilisent l'ordinateur pour essaier de trouver les meilleurs programmes possible et gagner la partie.

Ce projet était principalement tourné vers la création d'un algorithme génétique, cependant pour pouvoir le tester, il fallait créer un jeu de CoreWar complet. C'est donc ce que nous avons fait. Nous avons créé un jeu de CoreWar complet avec une machine MARS, un RedCode, un algorithme génétique et une interface graphique pour visualiser les parties.

Après avoir présenté les objectifs du projet, nous allons détailler le RedCode, la machine MARS, l'algorithme génétique et l'interface graphique que nous avons créé. Nous présenterons ensuite les résultats obtenus et conclurons sur ce que nous avons appris et les perspectives d'évolution que nous pourrions mettre en place.

1.1 Objectifs

L'objectif de ce projet était de créer un jeu de CoreWar complet. Pour cela nous avons dû identifier les points principaux qui composent notre projet et les implémenter. Nous avons donc décidé de découper notre projet en plusieurs parties :

1. Le RedCode
2. La machine MARS
3. Un algorithme génétique pour créer des guerriers
4. Une interface graphique pour afficher le déroulement des parties

Nous avions pour la réalisation de ce projet un peu plus de 3 mois. Durée de temps appropriée car elle nous a permis de bien réfléchir aux détails de la création du jeu et de le réaliser dans les temps sans avoir a délaisser les autres cours.

2 Le RedCode : Détails du Langage

Le **RedCode** est un langage proche de l'assembleur, uniquement utilisé par les joueurs de CoreWar et qui permet de programmer les guerriers. On utilise ce langage car par défaut il n'est pas compris par les ordinateurs. Cela évite que des personnes mal-intentionnées, ou simplement des erreurs, créent des bugs sur les pc des joueurs comme il n'y a pas risque qu'ils s'échappent de la machine

MARS. Au fur et à mesure des années ce langage a évolué avec l'ajout de nouvelles instructions mais pour ce projet nous utilserons une de ses premières versions : la norme icw88 (créee een 1988).

Dans cette version, le Redcode se compose de 11 instructions. Les instructions sont des commandes qui permettent de déplacer les guerriers, de modifier la mémoire, de sauter des instructions, etc. Les opérandes sont quant à eux un couple valeur-mode qui associe un entier et le mode d'adressage de cette valeur (Direct, Indirect, Immédiat et Pre-Decrement).

Voici les différentes instructions et leurs effets :

Instruction	**Description**
DAT A B	Supprime le processus en cours d'exécution de la file d'attente des processus
MOV A B	Déplace A dans B
ADD A B	Ajoute A à B
SUB A B	Soustrait A à B
JMP A B	Saute à A
JMZ A B	Saute à A si B est égal 0
JMN A B	Saute à A si B est différent de 0
CMP A B	Si A est égal à B, ignore l'instruction suivante
SLT A B	Si A est inférieur à B, ignore l'instruction suivante
DJN A B	Décrémente B ; Si B est différent de 0, saute à A
SPL A B	Place A dans la file d'attente des processus

Ensuite, pour définir sur quelle cellule l'instruction va s'éxécuter on retrouve 4 modes d'adressage différents. Ils sont représentés par des symboles et sont associés aux valeur A et B pour former des **Operandes**.

Voici les différents modes d'adressage et leurs effets :

Mode	**Description**
Direct	L'opérande est une adresse mémoire vers une autre cellule
Indirect	L'opérande est une adresse mémoire vers une autre adresse (Comme si on faisait 2 Direct)
Immédiat	L'opérande est une valeur
Pre-Decrement	Similaire à l'indirect mais on fait -1 sur la deuxième adresse avant d'aller chercher la cellule

Afin de pouvoir créer des guerriers et réaliser notre projet, il est nécessaire de bien comprendre le RedCode, ses instructions et modes d'adressage. Nos premières séances furent donc dédiées a la compréhension du fonctionnement de ces derniers. Une fois compris, nous avons pu créer notre machine MARS qui interprète ces instructions et les exécute.

3 Machine MARS : Mémoire et Déroulement de la partie

La **machine MARS** est la pièce maitresse des parties de CoreWar. C'est la machine virtuelle dans laquelle les guerriers vont s'exécuter. Elle gère la mémoire, interprète le RedCode et est responsable du déroulement des parties, sans elle pas de projet c'est donc par là que nous avons commencé notre travail.

3.1 Mémoire

Une des problèmatiques de cette partie était la **représentation de la mémoire**. Nous avons choisi d'utiliser une **liste doublement chainée** car la mémoire devait être circulaire et il facile de réaliser ce type de mémoire avec cette structure de données.

Dans cette liste chainée chaque cellule contient **une instruction et deux Operandes**. Pour rappel, les instructions sont les commandes que les guerriers vont exécuter et les Operandes contiennent les informations sur la méthode a appliquer pour exécuter l'instruction.

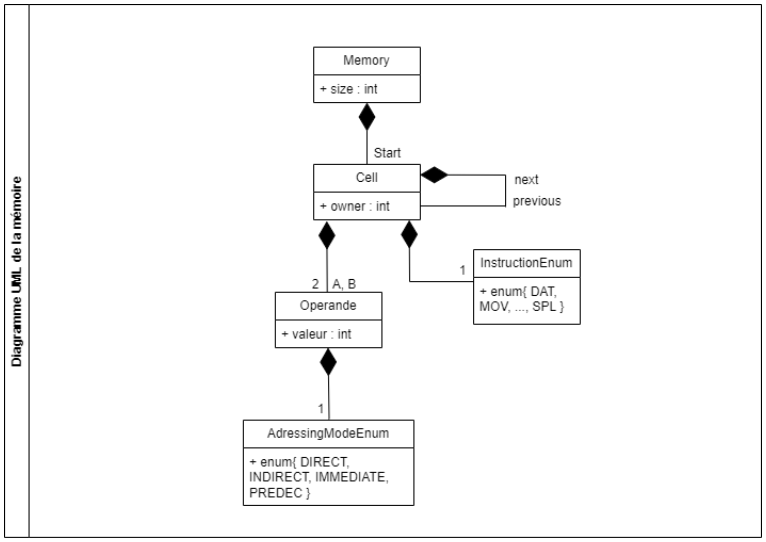


FIGURE 1 – Diagramme UML de la mémoire

3.2 Déroulement d'une partie

La machine MARS doit aussi gérer le déroulement des parties. Pour cela elle doit pouvoir exécuter les instructions des guerriers et gérer les processus.

A l'initialisation de la machine MARS, le RedCode est placé a l'index 0 de la mémoire pour le premier guerrier et au milieu de la mémoire pour le second. Un pointeur vers la première instruction de chacun de ses programmes est mit dans la file d'attente des processus. C'est toujours le premier guerrier qui commence

a jouer c'est pourquoi on effectue 2 fois un combat, une fois où le guerrier 1 est a l'index 0 et une fois où c'est le guerrier 2 qui y est dans l'algorithme génétique (voir 4.2).

En général chaque programme exécute une instruction chacun a son tour. Mais certaines instructions comme SPL ou DAT permettent respectivement d'ajouter des processus dans la file d'attente ou d'en supprimer. En altérant la file on a donc des tours ou plutôt que de jouer p1 puis p2 on jouera p1-p1-p2. Ces instructions permettent des stratégies plus complexes et des parties plus intéressantes.

On décide que la partie est terminée lorsque tout les processus dans la file d'attente sont au même programme. Cela signifie que le programme adverse n'a plus de processus en attente et que le programme en cours d'exécution est le dernier. C'est donc lui qui a gagné la partie.

4 Algorithme Génétique

Après avoir créer la machine MARS nous avons pu développer l'algorithme génétique. L'objectif étant d'avoir un algorithme qui crée des programmes en RedCode qui sont des meilleurs guerriers que les précédent. On peut dissocier le développement de cet algorithme en deux grandes parties : Le Fonctionnement de l'algo en lui même et ensuite les différentes méthodes afin de l'entrainer efficacement.

4.1 Fonctionnement

4.1.1 Création de guerriers aléatoires : les seeds

Nous avons opté pour un algorithme génétique dans lequel on retrouve une grande partie d'aléatoire. Cela permet idéalement de ne pas rester bloqué sur certains programmes et d'avoir un guerrier qui est bon contre la majeure partie de ses adversaires en explorant un maximum de possibilités.

Pour cela il a d'abord fallu pouvoir générer un guerrier aléatoirement. Nous avons alors travailler sur un système de génération de **seeds** aléatoires. Ces seeds sont des nombres générés aléatoirement que l'on peut convertir en une ligne de RedCode.

Exemple : *la ligne de RedCode "ADD @10, <5" est représenté par la ligne 02101005, 02 étant le code de l'instruction ADD, le premier 10 les modes d'adressage, puis enfin 10 et 05 les valeurs de A et B.*

Une fois que l'on a généré une ligne aléatoirement il suffit tout simplement de la placer dans une **"seedline"** un ArrayList de Seed qui nous permet de représenter un guerrier. On peut alors créer un guerrier aléatoirement a partir de rien. Et c'est en générant plusieurs de ces guerriers que l'on peut créer une population.

4.1.2 L'amélioration des populations

Une fois que l'on a généré une population il faut pouvoir l'améliorer. Pour ce faire nous avons travaillé sur un système de reproduction et de mutation.

La **reproduction** consiste à prendre deux guerriers de la population et à les croiser pour créer un nouveau guerrier. Pour cela on prend une partie de la seedline d'un guerrier et on la remplace par une partie de la seedline de l'autre guerrier. Cela permet de créer un nouveau guerrier qui a des caractéristiques des deux guerriers de départ. On choisit un pivot aléatoirement dans la plus petite graine et on échange les parties de la seedline avant ou après ce pivot (décidé a Pile ou Face).

Afin que le système de reproduction soit efficace nous avons choisi de faire en sorte que le guerrier le plus performant de la population se reproduise avec tout les autres guerriers. Cela permet de garder les caractéristiques du guerrier le plus performant et de les diffuser dans toute la population afin d'en obtenir une nouvelle.

Pour déterminer le guerrier le plus performant il a fallu réfléchir a des méthodes de scoring. Nous avons choisi de faire s'affronter les guerriers deux par deux et de compter le nombre de victoires de chacun. Le guerrier qui a le plus de victoires est alors le guerrier le plus performant. Aussi un guerrier qui s'est suicidé dans les 32 premiers cycles ne rapporte pas de point au gagnant mais en fait perdre un. Cela permet de pénaliser les guerriers qui se suicident trop rapidement sans récompenser celui qui gagne a cause d'eux.

La **mutation** est un autre moyen d'améliorer les guerriers. Elle consiste à prendre un guerrier et à changer une partie de sa seedline aléatoirement. Cela permet de créer des guerriers qui sont différents de ceux de la population et qui peuvent être meilleurs.

Pour que la mutation soit efficace il a fallu déterminer un taux de mutation. A chaque reproduction le guerrier obtenu a une chance sur 3 de muter. Cela permet de ne pas trop modifier les guerriers et de ne pas perdre les caractéristiques des guerriers de départ.

4 types de mutations ont été implémentées :
- **Modified** : Modifie une des 4 données de la seed aléatoirement (Instruction, Modes, Valeur de A, Valeur de B), si c'est une instruction on change aussi le mode d'adressage en fonction pour éviter les erreurs.
- **Additional** : On ajoute une ligne de RedCode aléatoire dans le guerrier.
- **Removal** : On supprime une ligne de RedCode aléatoire du guerrier.
- **Swap** : On intervertit deux lignes du guerrier entre elles aléatoirement.

Enfin, pour que l'algorithme génétique soit efficace il a fallu déterminer un nombre de générations. Nous avons choisi de faire 100 . Cela permet de bien faire évoluer les guerriers et de ne pas rester bloqué sur des guerriers qui ne sont pas performants.

Voici un exemple de code pour la génération d'un nouveau guerrier :

```
generateChild(Seed parent1, Seed parent2) {
    Seed enfant = new Seed();
```

```
        // Effectuer le croisement pour generer les
            instructions de l'enfant
        enfant = crossover(parent1, parent2);

        // Appliquer la mutation a l'enfant
        mutation(enfant);

        // Renvoyer l'enfant
        return enfant;
    }
```

4.2 Entrainement

Une fois notre algorithme génétique créé il a fallu l'entrainer. Pour cela nous avons du générer une population aléatoire puis y faire s'affronter les guerriers et les faire évoluer en conséquence. Cette opération doit être répété plusieurs miliers de fois pour que les guerriers soient performants. Afin de réduire le temps d'entrainement nous avons fait évoluer nos méthodes et développé 3 façons d'entrainer nos guerriers et de vérifier que l'algo fonctionne ont été implémentées.

4.2.1 Le CheatCode DAT

Il nous fallait d'abord une méthode très rapide et très simple qui nous permettrait de vérifier que l'algorithme marche. Pour cela nous avons décidé d'une méthode très simple : le CheatCode.

Le CheatCode est une ligne qui permet de gagner la partie en un seul coup. En effet nous avions décidé que si un guerrier avait une ligne dont l'instruction était DAT alors il gagnait automatiquement la partie. Cela permettait de vérifier que l'algorithme marchait et que les guerriers trouveraient cette ligne et la garderaient dans les générations suivantes.

Au départ lorsque l'on générait une population aléatoire il y avait forcément très peu de guerriers qui contenaient une instruction DAT mais après a peine quelques générations on pouvait voir que les guerriers gardaient cette instruction et que la population était de plus en plus performante. L'algorithme fonctionnait donc bien !

4.2.2 Entrainement d'une population aléatoire

Une fois que l'on a vérifier que l'algorithme marche il faut donc trouver un moyen d'entrainer les guerriers réellement. Et oui, dans une partie de CoreWar pas de Cheatcode. La première méthode qui nous est venue en tête était de générer une population aléatoire et de faire s'affronter chacun des guerriers de cette population. Le guerrier qui avait le plus de victoire était le plus performant et on pouvait alors le faire se reproduire avec les autres guerriers de la population.

Cette méthode est celle qui nous permettrait d'avoir le guerrier le plus précis et meilleur contre le plus d'adversaire. Cependant elle est aussi la plus longue. 100 guerriers qui participent chacun a 100 combats a chaque génération, c'est quand même 10000 combats a répéter plusieurs centaines si ce n'est milliers de fois. C'est quand nous avons pris conscience du temps que cela allait prendre de générer des programmes performants que nous avons décidé de mettre en place l'utilisation des **threads** pour accélérer le processus. En effet, grâce aux threads on peut faire se dérouler plusieurs combats simulatanément en affectant une machine par thread et donc réduire drastiquement le temps nécessaire pour l'entrainement. Mais le temps restant long nous avons également mis en place un moyen de sauvegarder la progression. On exporte la dernière population et ses scores dans des fichiers binaires et on les importe lorsque l'on relance le programme. Cela permet de ne pas perdre de temps a reentrainer une population déjà entrainée.

Une fois le temps d'entrainement réduit nous avons aussi eu besoin de moyen de visualiser l'amélioration de nos guerriers. Pour cela nous avons décidé de générer des graphiques nous affichant l'évolution des scores de nos guerriers.

Comme on peut le voir, la courbe des premiers graphiques ne montrent pas d'évolution probante. Cela est du au fait que avec notre méthode d'affichage et d'entrainement nous n'avons pas un référentiel de comparaison. En effet, un programme qui a un score de 120 a la première génération n'est pas forcément meilleur qu'un programme qui a un score de 100 a la 10ème génération. Un a battu des programmes plus faibles que l'autre. C'est pour cela qu'il nous fallait revoir notre méthode de scoring et d'entrainement pour avoir des résultats plus pertinants a vous montrer.

4.2.3 Le Challenger

Pour avoir un référentiel de comparaison nous avons décidé de créer un guerrier qui serait notre référence. Ce guerrier, appelé le **Challenger**, est un guerrier qui est toujours le même. Pour le graphique en dessous nous générons 100 population et le challenger est le meilleur programme de la centième population. Mais cela peut être un programme généré aléatoirement ou un programme trouver sur internet. L'objectif est d'avoir un programme fixe qui permet de voir l'évolution de nos guerriers face a un programme performant.

Chaque guerrier de notre population est donc confronté au Challenger et on peut alors voir si il est performant ou non. Cela permet de savoir si notre algorithme génétique est efficace et si nos guerriers sont de plus performants.

Cependant si on ne fait seulement des combats contre le Challenger et que l'on ne fait pas s'affronter les guerriers entre eux on n'obtiendrait un programme uniquement performant face au Challenger avec très peu d'adaptabilité face à d'autres guerriers. Les guerriers ont donc deux scores : un score face au Challenger et un score face aux autres guerriers de la population. Cela permet de savoir si un guerrier est performant ou non.

Le score du challenger est celui que l'on affiche dans les graphiques. Cela permet de voir l'évolution de nos guerriers face à un guerrier performant et de savoir si notre algorithme génétique est efficace. L'autre score sert toujours a

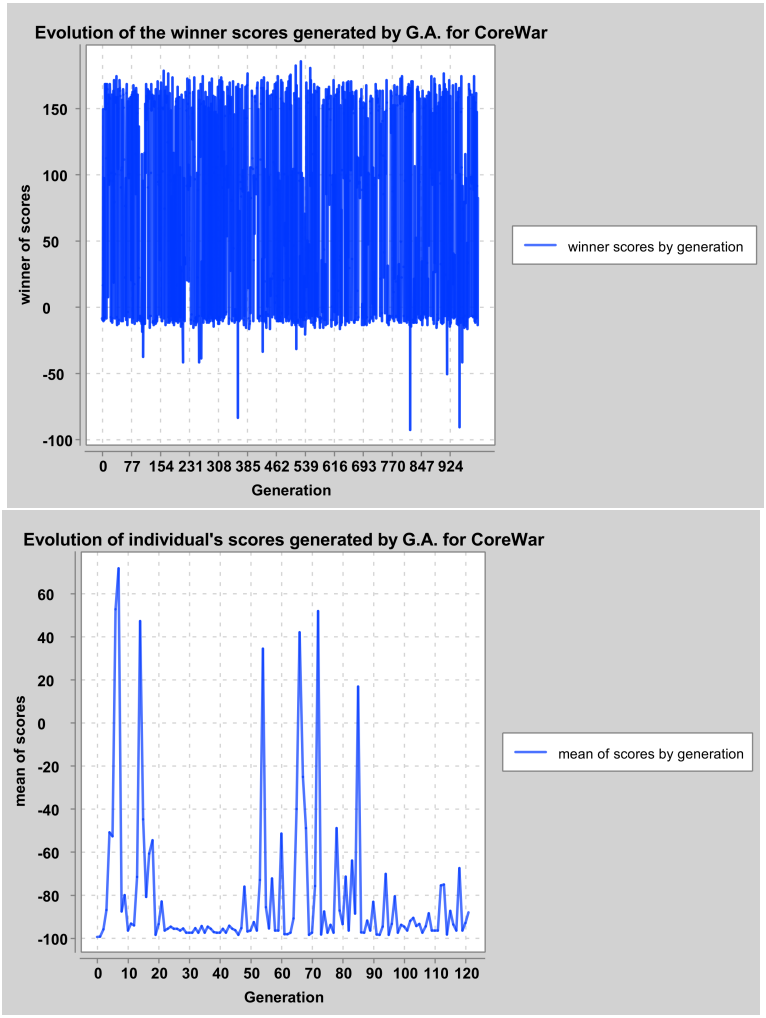
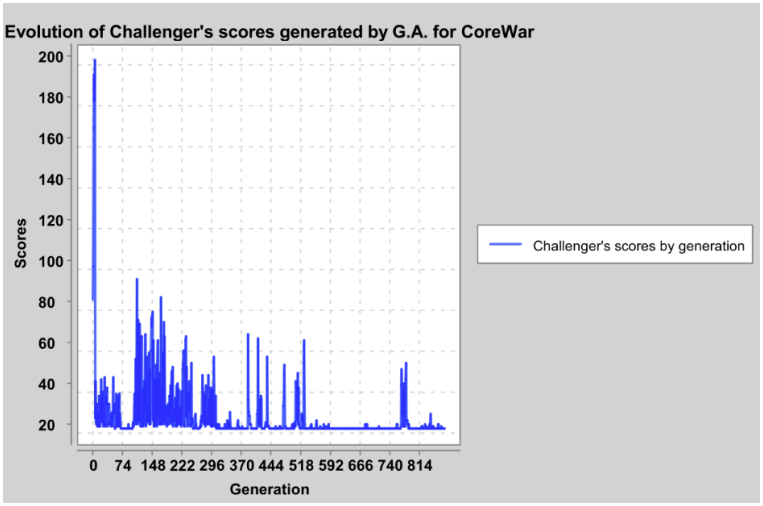


FIGURE 2 – Premiers graphiques obtenus en utilisant le score des gagnants

déterminer le guerrier le plus performant de la population pour l'amélioration de cette dernière.

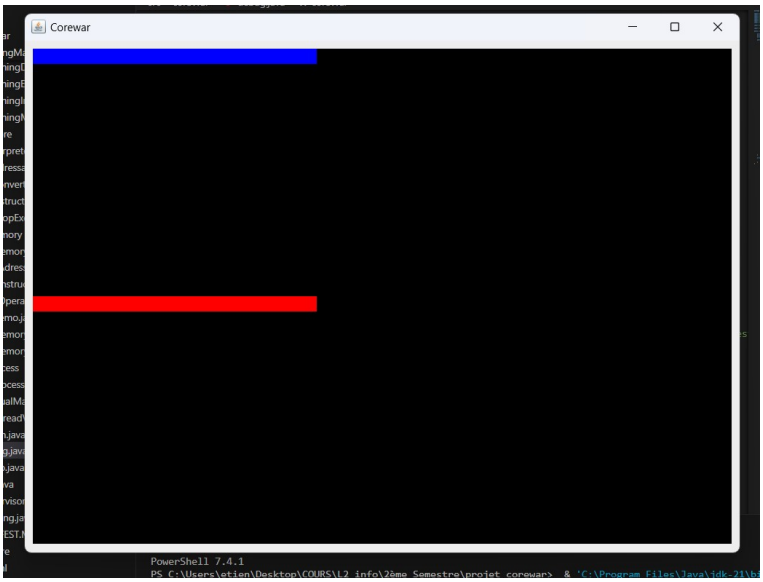
On voit sur le graphique obtenu avec cette méthode que la tendance globale est descendante c'est donc bien que les algorithmes s'améliore face au Challenger. Cependant on voit aussi que les scores sont très variables. Cela est du au fait qu'une population dans laquelle la plupart des programmes se suicident ne donnent que très peu de point au Challenger car pour rappel, si un programme se suicide on ne donne pas de point au gagnant, on en enlève un au perdant. L'idée est de voir la tendance des pics de score qui sont ceux d'une génération ou peu, voir aucun programme ne se suicident. On voit bien qu'ils descendent c'est donc bien que le Challenger perd de plus en plus de combat et est donc confronté a des algorithmes de plus en plus performants.



5 Interface Graphique

L'interface a été faite avec le package **swing**, déjà inclus dans java. Elle permet de visualiser une partie et ne se lance que si on lui demande. On ralentit le temps entre deux instructions quand elle est appelée sinon la partie se termine trop rapidement pour que ce soit lisible. Elle transforme la mémoire en une grille. Cette grille s'adapte à toute les tailles de mémoire pour d'éventuelles parties sur une mémoire a taille réduite. Elle est composée de cases qui peuvent être de 3 couleurs différentes : Bleu, Rouge et Noir. (le Bleu et le Rouge pour chacun des guerriers et le Noir pour les cases qui n'appartiennent encore a personne).

Pour actualiser l'affichage plusieurs méthodes ont été créées. Au départ la méthode consistait a prendre une nouvelle grille et à remplacer l'ancienne par la nouvelle. Mais pour éviter d'avoir trop de latence nous avons du créer une méthode qui actualise uniquement les cases qui ont changé de couleur. Chaque case est gérée comme un élément indépendant et peut être actualisée indépendamment des autres. Cela a permis d'optimiser l'affichage.



6 Conclusion : Résultats et évolutions

Nous pouvons considérer que les résultats sont satisfaisants. Nous avons réussi à créer un jeu de CoreWar complet avec une machine MARS qui interprète le RedCode et gère les parties. Aussi nous avons pu développer un Algorithme génétique qui fonctionne et qui est capable de créer des guerriers performants.

Cependant il reste des points à améliorer. L'algorithme génétique pourrait être amélioré pour être plus performant. Une perspective d'évolution serait de faire en sorte que le challenger soit une population test et non un guerrier unique. Cela permettrait de voir si nos guerriers sont performants face à une population plus clairement. Il pourrait être intéressant aussi de tester de nouveaux types de mutations et d'algorithme pour voir si certains sont plus performants que d'autres.

Une de nos principales contraintes fut aussi la contrainte matérielle. Bien que nous ayons fait un gros travail d'optimisation via l'utilisationd des threads, générer des milliers de population prend du temps et nécessite d'avoir un ordinateur suffisament puissant a sa disposition pour le laisser tourner et générer un maximum de programme. Il serait donc intéressant de voir si l'on atteint un guerrier ultime, meilleur que tout les autres, après un certain temps de génération.

Enfin, il serait intéressant de tester notre algorithme génétique sur d'autres jeux de programmation pour voir si il est performant et si il peut être utilisé pour créer des programmes performants dans d'autres domaines.

Ces possibilités d'amélioration ne réduit en rien notre satisfaction d'avoir réussi à créer un jeu de Corewar complet et de voir que notre algorithme génétique fonctionne. Nous avons appris beaucoup de choses durant ce projet au niveau des algorithmes génétiques ou encore du fonctionnement des machines. Le RedCode étant un langage proche de l'assembleur ce fut aussi une bonne

introduction a ce type de langage. Nous avons aussi progresser sur les méthodes de travail en groupe et de collaboration. C'est donc un projet que nous avons beaucoup apprécié et qui nous a permis de beaucoup progresser.