Projet Corewars

Isabelle Angeletti – Xavier Maupeu

13 Décembre 2013

Première partie

Présentation du Projet

1 Définition et cadre du projet

Nous avons réalisé le projet Corewars dans le cadre du cours d'Algorithmie en première année d'Informatique à l'Enseirb-Matmeca. Ce projet avait pour objectif de mettre en pratique les différentes connaissances obtenues au cours du premier semestre.

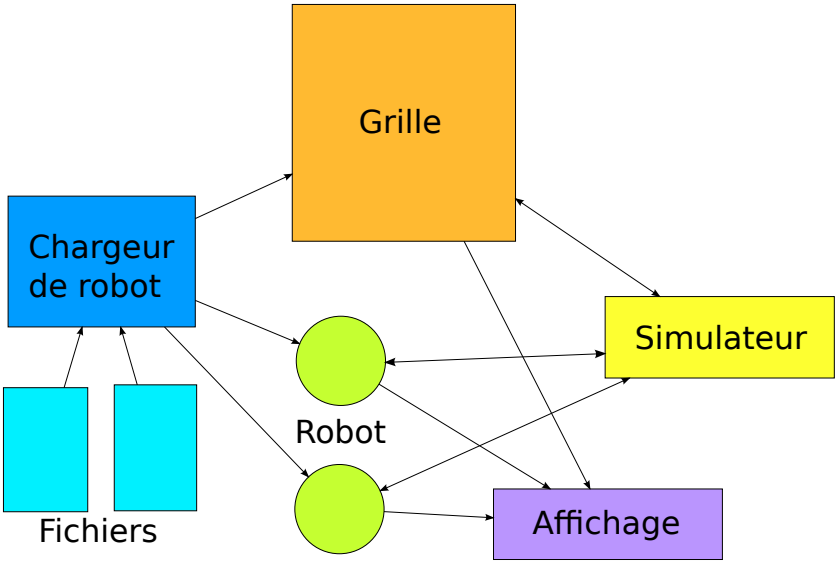
Le projet Corewars consiste à implémenter une zone de mémoire où deux robots constitués d'un code assembleur (le *Redcode*) peuvent s'affronter. Les robots sont chargés et placés de manière aléatoire sur la grille de mémoire. Un robot peut se dupliquer ce qui lui permet de créer un nouveau fil d'exécution. Celui-ci est détruit lorsqu'il exécute une instruction invalide. Un robot a gagné lorsque son adversaire n'a plus de fil d'execution. Un fil d'éxcution reprèsente l'exécution d'un ensemble d'instructions du langage machine.

2 Analyse du projet

Le projet ne comporte pas de grosse difficulté algorithmique, les principales difficultés sont :

- la réalisation d'un grille capable de contenir les instructions
- la conversion du code redcode stocké dans un fichier texte en une suite d'instruction stocké sur la grille
- la création d'un controleur permettant de simuler le code et de déterminer si un robot exécute une instruction invalide
- l'affichage de la grille ainsi que des 5 dernières instructions de chaque robot

L'un de nos premiers objectifs a été de réaliser un schéma récapitulant toutes les intéractions entre les différents modules de notre programme.



2.1 Le Redcode

La version 1984 du Redcode que nous utilisons comporte un jeu d'instruction très réduit :

- *DAT B* : indique que la zone mémoire comporte une donnée de valeur B
- *MOV A B* : déplace l'opérande A vers B
- *ADD A B* : ajoute les opérandes A et B et place le résultat dans B
- *SUB A B* : soustrait l'opérande A avec B et place le résultat dans B
- *JMP B* : saute à l'instruction indiqué par l'addresse relative donnée par l'opérande B
- *JMZ A B* : si l'opérande A vaut 0, saute à l'instruction indiquée par l'addresse relative B
- *DJZ A B* : décrémente l'opérande A puis si A vaut 0, saute à l'instruction indiquée par l'addresse relative B
- *CMP A B* : si les opérandes A et B sont différentes, saute l'instruction suivante

Il faut aussi rajouter l'instruction *SPL B* qui permet de créer un nouveau fil d'exécution commençant à l'addresse relative donnée par l'opérande B. L'instruction SPL va dupliquer le processus qui l'exécute.

Nous avons tout d'abord décidé de coder une instruction sur 32 bits, mais lorsque que nous avons voulu rajouter l'instruction *SPL* nous avons dû passer à 64 bits.

Chaque opérande A et B comporte un mode d'addressage sur 2 bits qui peut être immédiat, relatif ou indirect. Le nombre en lui-même est codé sur 12 bits et le 12^{eme} bit correspond au signe (un bit à 1 indique un nombre négatif).

Opcode	Mode pour A	Mode pour B	Signe A	Valeur A	Signe B	Valeur B
5 bits	2 bits	2 bits	1 bit	11 bits	1 bit	11 bits

Un des problèmes rencontrés lors de la réalisation du projet était la comprehénsion des différentes instructions. En effet, il fallait comprendre le fonctionnement de chaque instruction pour pouvoir les implémenter et vérifier ensuite si notre simulateur exécutait bien l'instruction.

3 Conception de la solution algorithmique

3.1 La gestion des différents fils d'exécution

Nous avons choisi d'utiliser une liste chainée pour stocker les différents fils d'exécutions. En effet nous avons besoin d'en créer et d'en supprimer au cours de l'exécution du robot.

De plus, pour faciliter le debuggage, chaque fil doit pouvoir garder une trace des 5 instructions précédemment exécutées. On utilise pour cela un tableau de 5 cases et après avoir exécuté chaque instruction on décale les instructions contenues dans le tableau et on met l'intruction récente au début.

Pour que le combat soit equitable, pour chaque tour et pour chaque robot on prend le fil d'éxécution suivant dans le liste. De cette manière un robot qui a un seul fil d'exécution va être très rapide.

Deuxième partie

Réalisation du projet

4 Organisation du projet

Au niveau de l'organisation du projet, nous avons déterminé trois fonctionnalités principales :

- Le chargement des robots
  - Lecture des fichiers
  - Conversion d'une instruction
- L'exécution des instructions
- L'affichage des robots et de la grille

Chacune de ses fonctionnalités est divisée en sous-fonctions.
Nous avons chacun travaillé sur un des deux modules principaux que sont le chargement des robots et l'éxécution des instructions. Pour l'affichage de la grille et des robots, nous avons tous les deux travaillé sur cette fonctionnalité.

Nous n'avons eu aucun problème au niveau de l'organisation car nous savions chacun sur quel module travailler. Nous partagions tout ce que nous faisions régulièrement, et nous nous aidions mutuellement lorsque l'un de nous rencontrait des difficultés.

5 Structure pour la mémoire

5.1 La grille

Le rôle principale de la grille est de contenir les différentes instructions utilisées par les robots. Comme la taille de la grille est constante et que l'on veut avoir un accès par index, on peut utiliser un tableau statique. Nous avons choisi d'utiliser un tableau de structure où la structure contient une instruction et un entier indiquant quel robot a écrit cette instruction. Cela nous permet lors de l'affichage de la grille de montrer les portions de grille modifiées par les robots.

5.2 Robots et fils d'exécutions

Voici la structure en liste chaînée permettant de stocker un fil d'exécution. Un robot est donc constitué d'une *id* qui peut valoir 1 ou 2, d'une liste de fil d'exécution et d'un pointeur vers le fil d'exécution courant : celui qui est train de s'exécuter.

```
typedef struct process
{
    // if the bot execute a wrong instruction
    bool abort;
    // position of the current instruction
    int position;
    // some instructions have to store their value
    int answer_instruction;
    // latest instructions
    uint64_t last_instruction[5];
    struct process* next_process;
} process;

typedef struct
{
    int id;
    process* list_process;
    process* current_process;
} bot;
```

6 Implémentation des différents modules

6.1 L'affichage

Nous avons choisi de représenter la mémoire par une grille remplie de zéro. Pour afficher cette grille dans le terminal, nous avons utilisé la bibliothèque nCurses qui permet de donner un peu de style au terminal. Cette librarie permet par exemple d'afficher de la couleur dans le terminal.
Nous avons ensuite représenté l'exécution des deux robots dans la mémoire par les chiffres 1 et 2 sur la grille. Dans des soucis de visibilité, nous avons attribué à chaque robot une couleur différente.

Pour faciliter le débugage des robots et permettre de suivre son exécution, nous affichons les 5 dernières instructions sur le côté de la grille. Lorsque le robot exécute une instruction invalide, une phrase indiquant qu'il a crashé apparaît à côté de son numéro.
Enfin, le numéro du tour qui est en train d'être joué et le score des deux robots sont aussi inscrits sur le côté de la grille. Lorsqu'un des robots atteint un score égal à 5, il a gagné. Pour indiquer à l'utilisateur quel robot a gagné, une ligne s'affiche en magenta sous le score des deux robots.

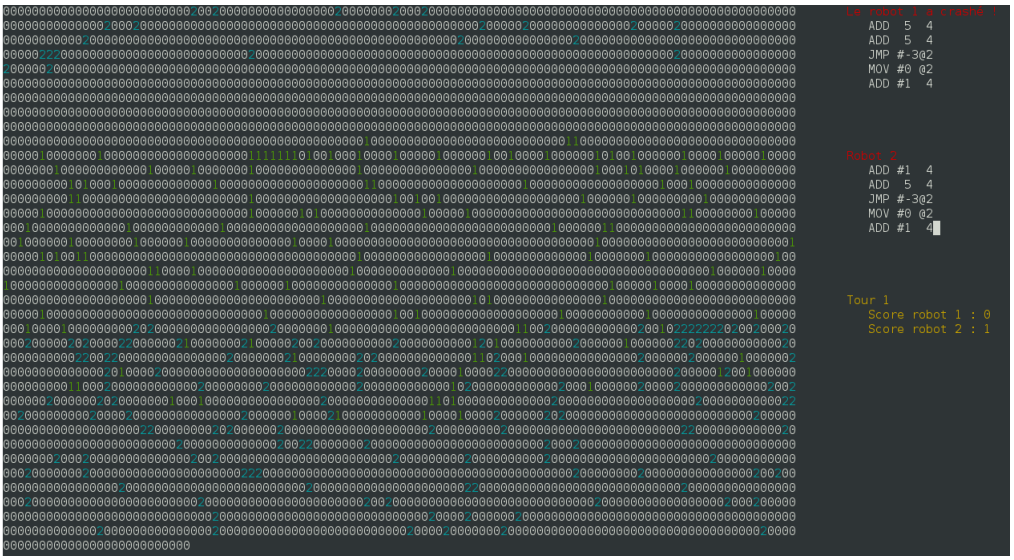


FIGURE 1 – Rendu final de CoreWar

6.2 Chargement des robots

Por charger un robot, il faut récupérer toutes ses instructions. Les différentes instructions du robot sont stockés dans un fichier. Pour les récupérer, nous commençons par récupérer et traiter la première ligne du fichier puis nous continuons jusqu'à ce que nous arrivions à la fin du fichier.

Une instruction est de la forme ADD #7 @-120. Lorsque nous la récupérons, c'est une chaîne de caractère. Pour que notre programme puisse l'éxcuter nous devons la convertir. Pour cela, nous la séparons en trois parties.

Une fois que nous avons récupéré la première partie correspond au code de l'opération, nous convertissons cette chaîne de caractère en un entier compris entre 0 et 8. Nous regardons ensuite si cette opération comprend une ou plusieures opérandes. Par exemple, les opérations SPL, DAT et JMP ne possédent qu'une seule opérande. Si l'opération accepte deux composantes A et B, nous allons traiter deux autres parties de notre instruction.
Le traitement effectué pour récupérer la valeur et le mode de l'opérande se fait en parcourant la châine de caractère. Suivant les symboles lu, on peut déterminer le mode, si le nombre est négatif et la valeur de l'opérande. L'extrait de code ci-dessous montre le traitement effectué pour récupérer le mode et les valeurs des opérandes.

```
uint64_t get_mode_and_value(int f_begin, char* f_instruction, char*
    f_mode, int* f_end)
{
    char _argument[20];
    int _negatif = 0;
    int k = 0;
    int j;

    *f_mode = ' ';

    for (j = f_begin; f_instruction[j] != ' ' && f_instruction[j] != '\0'
        ; j++)
    {
        // a number is read
        if (f_instruction[j] >= '0' && f_instruction[j] <= '9')
        {
            _argument[k] = f_instruction[j];
            k++;
        }
      // a minus is read: number is negative
        else if (f_instruction[j] == '-')
        {
            _negatif = 1;
        }
      // "#" read: mode is IMMEDIATE
        else if (f_instruction[j] == '#')
        {
            *f_mode = '#';
        }
      // "@" read: mode is INDIRECT
        else if (f_instruction[j] == '@')
        {
            *f_mode = '@';
        }
    }
    _argument[k] = '\0';
    *f_end = j + 1;

    if (_negatif == 0)
    {
        return atoi(_argument);
    }
    else
    {
        return atoi(_argument) + (1 << 11);
    }
}
```

Après avoir récupéré le code de l'opération, les modes et les valeurs des deux opérandes il faut les convertir. Pour cela, nous faisons un décalage bit à bit.

Fields	Encoded integer
Opération = op	$op << 28$
Mode A = modA	$modA << 26$
Mode A = modB	$modB << 24$
A	$A << 12$
B	B

Lorsque tous les éléments constituant une instruction sont encodés, nous les additionnons. Pour les instructions SPL, DAT et JMP, nous ajoutons uniquement les valeurs du code de l'opération, du mode B et de B. Nous avons alors notre instruction codée sur 64 bits prête à être interprétée par le simulateur.

6.3 Le simulateur

Nous avons fait plusieurs fonctions permettant de récupérer l'opcode de l'opération, les modes et les opérandes à partir d'une instruction encodée.
Lorsque ces différentes valeurs sont récupérées, notre simulateur va analyser le code de l'opération et réaliser le traitement correspondant à l'instruction.

Par exemple pour l'instruction ADD A B, le simulateur récupère l'opérande A et l'opérande B, fait l'addition, stocke le resultat dans une variable situé dans le fil d'exécution et écris une instruction DAT à l'emplacement mémoire indiqué par B.

7 Problèmes de mise en œuvre

Le sujet ne spécifiait pas l'implémentation des nombres, nous avons donc choisi d'avoir des nombres négatifs et pour que cela reste simple a coder, les nombres sont sur onze bits et le douzième indique le signe.
Pour l'instruction SPL, nous avons du changer totalement la structure des robots et implémenter les fils d'exécution.

NCurses crée des artefacts graphiques lorsque la fonction refresh() est appelé trop souvent. Nous avons réussi a les diminuer en faisant des refresh() toutes les 50 instructions.

8 Tests de validité

Au fur et à mesure du développement, nous avons créée et récupéré des robots sur internet afin de vérifier le bon fonctionnement des instructions. Particulièrement pour l'instruction SPL qui a été délicate à débugger.
Pour faciliter le debuggage nous avons rajouté la possibilité de rajouter l'argument *step* lors du lancement du programme pour exécuter chaque instruction étape par étape.

9 Conclusion

Nous avons réalisé toutes les fonctionnalités qui nous ont été demandé. Notre programme est fonctionnel : il permet de faire s'affronter différents robots.
Ce projet a été très intéressant a été réalisé. Il nous a permis de découvrir le jeu CoreWar ainsi que le langage RedCode. Au premier abord, ce projet nous a paru compliqué à mettre en place, mais au final nous avons réussi à le réaliser sans trop de difficulté.