Core War

Sujet proposé par Jean-Christophe Filliâtre
mailto:Jean-Christophe.Filliatre@lri.fr

Difficulté : dans la moyenne (⋆⋆)

Version 2 (10 mars 2011)

URL de suivi : `http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/10-11/INF431/core_war/`

Le but de ce projet est de réaliser un simulateur pour le jeu Core War.

1 Core War

Le jeu Core War est apparu dans un article d'Alexander Dewdney de mai 1984 publié dans le *Scientific American*. Le principe de ce jeu est extrêmement simple. Deux programmes informatiques sont chargés dans la mémoire d'une machine virtuelle et s'y affrontent. Ils exécutent chacun à tour de rôle une instruction. Le premier programme qui exécute une instruction illégale est éliminé. Tout l'intérêt du jeu vient du fait qu'un programme peut écrire n'importe où dans la mémoire et peut donc provoquer l'élimination de l'autre joueur en écrivant une instruction illégale dans le code de ce dernier. Un joueur peut également se déplacer et se dupliquer dans la mémoire, pour échapper à son adversaire.

Mémoire. La machine virtuelle est constituée d'une mémoire et d'une structure contenant l'état de chaque joueur. La mémoire est constituée de M mots. Elle est *circulaire*, c'est-à-dire que les adresses mémoire sont toujours calculées modulo M. Plus généralement, toute l'arithmétique de la machine virtuelle est réalisée modulo M. Dans ce projet, on fixe la taille de la mémoire à $M = 800$. Chaque mot mémoire est décomposé en trois parties : une opération d'une part et deux opérandes A et B d'autre part. La mémoire ressemble donc à ceci :

adresse	opération O	opérande A	opérande B
000			
001			
⋮			
799			

Les deux opérandes A et B sont des entiers, compris entre 0 et $M-1$. L'opération spécifie l'instruction qui doit être exécutée et les modes d'adressage qui doivent être utilisés pour interpréter les deux opérandes. Les instructions sont au nombre de 10. On les présente rapidement ici (le sens précis des opérandes sera spécifié en détail plus loin).

- `MOV` : copie A vers B
- `ADD` : ajoute A à B
- `SUB` : soustrait A de B
- `JMP` : saute à l'adresse A
- `JMZ` : saute à l'adresse A si B est nul
- `JMN` : saute à l'adresse A si B est non nul
- `CMP` : si $A = B$, saute l'instruction suivante
- `SLT` : si $A < B$, saute l'instruction suivante
- `DJN` : décrémente A ; puis saute à l'adresse A si B n'est pas nul
- `SPL` : crée un nouveau processus démarrant à l'adresse A

À cela s'ajoute une onzième instruction, `DAT`, qui n'est pas une instruction à proprement parler, mais peut servir à stocker des données dans ses opérandes A et B. C'est la tentative d'exécution de `DAT` qui provoque l'élimination d'un programme.

Adressage de la mémoire. On a déjà expliqué que les adresses étaient calculées modulo M. D'autre part, les adresses sont toujours *relatives* à celle de l'instruction qui est exécutée. En particulier, un programme ne connaît pas l'adresse absolue où il se trouve et il ne peut pas la calculer. Il raisonne toujours relativement à sa position.

Il existe quatre modes d'adressage : immédiat, direct, indirect et décrément. Chaque instruction spécifie le mode d'adressage de ses opérandes. La signification des modes d'adressage est détaillée dans le paragraphe suivant.

Sémantique. On décrit ici formellement l'exécution d'une instruction. La mémoire est représentée par trois tableaux O, A et B. Soit p l'adresse de l'instruction à exécuter. Soient m_A et m_B les modes des opérandes A et B de l'opération $O[p]$. On commence par évaluer l'opérande A, de la manière suivante :

si $m_A =$ immédiat **alors**
  $a \leftarrow 0$
sinon
  $a \leftarrow A[p]$
  si $m_A \neq$ direct **alors**
    si $m_A =$ décrément **alors**
      $B[p+a] \leftarrow B[p+a] - 1$
    $a \leftarrow a + B[p+a]$

On notera que cette évaluation peut avoir modifié le contenu de la mémoire. Puis on évalue l'opérande b de la même façon :

si $m_B =$ immédiat **alors**
  $b \leftarrow 0$
sinon

$b \leftarrow B[p]$
si $m_B \neq$ direct **alors**
  si $m_B =$ décrément **alors**
    $B[p+b] \leftarrow B[p+b] - 1$
  $b \leftarrow b + B[p+b]$

Là encore, le contenu de la mémoire peut avoir été modifié. Enfin on exécute l'instruction. L'exécution d'une instruction est détaillée figure 1. La valeur de p est modifiée pour indiquer l'instruction suivante à exécuter.

Processus. Chaque joueur n'exécute pas un mais plusieurs programmes, appelés ici *processus*. Un processus n'est rien d'autre que l'adresse de l'instruction qu'il doit exécuter. Les processus d'un joueur sont stockés dans une *file*. Initialement, la file d'un joueur ne contient qu'un seul processus, égal à l'adresse où le code du joueur a été chargé. À chaque tour où un joueur doit jouer, le premier processus de sa file est extrait ; appelons p son adresse. L'instruction située à l'adresse p est examinée et trois cas de figure se présentent :

- instruction `DAT` : le processus est éliminé ; s'il n'y a plus de processus dans la file du joueur, celui-ci a perdu la partie.
- instruction `SPL` : le processus est remplacé par deux processus, l'un correspondant à l'adresse $p+1$ et l'autre à l'adresse spécifiée par A. Le processus correspondant à $p+1$ est remis dans la file *avant* celui correspondant au saut. Si le nombre maximal de processus est atteint, cette instruction ne fait rien *i.e.* un seul processus est remis dans la file, correspondant à $p+1$.
- toute autre instruction est exécutée puis le processus est remis à la fin de la file, avec l'adresse correspondant à sa prochaine instruction ($p+1$ ou celle résultant d'un saut, le cas échéant).

Les joueurs jouant à tour de rôle, il est important de noter que les processus d'un joueur s'exécutent d'autant plus lentement qu'ils sont nombreux.

Affrontement. L'affrontement de deux joueurs consiste en un nombre pair de matchs entre les deux joueurs, chaque joueur étant le premier à jouer dans la moitié des matchs. Chaque match consiste à charger les programmes respectifs des deux joueurs à des adresses choisies aléatoirement, en garantissant seulement que les deux programmes ne se chevauchent pas. Le reste de la mémoire est initialisé par `DAT #0, #0`. Tant qu'aucun des joueurs n'est éliminé, on exécute leurs processus à tour de rôle. Si le nombre maximal de tours est atteint, il y a match nul. Dans ce projet, on prendra les paramètres suivants :

paramètre	valeur
taille de la mémoire M	800
taille maximale d'un programme	100
nombre maximal de processus d'un joueur	200
nombre maximal de tours	1 000 000
nombre de matchs dans un affrontement	40

selon l'instruction $O[p]$ **:**
`DAT` :
  rien à faire
`MOV` :
  si $m_A =$ immédiat **alors** $B[p+b] \leftarrow A[p]$
  sinon $O[p+b] \leftarrow O[p+a]$; $A[p+b] \leftarrow A[p+a]$; $B[p+b] \leftarrow B[p+a]$
  $p \leftarrow p+1$
`ADD` :
  si $m_A =$ immédiat **alors** $B[p+b] \leftarrow B[p+b] + A[p]$
  sinon $A[p+b] \leftarrow A[p+b] + A[p+a]$; $B[p+b] \leftarrow B[p+b] + B[p+a]$
  $p \leftarrow p+1$
`SUB` :
  si $m_A =$ immédiat **alors** $B[p+b] \leftarrow B[p+b] - A[p]$
  sinon $A[p+b] \leftarrow A[p+b] - A[p+a]$; $B[p+b] \leftarrow B[p+b] - B[p+a]$
  $p \leftarrow p+1$
`JMP` :
  $p \leftarrow p+a$
`JMZ` :
  si $B[p+b] = 0$ **alors** $p \leftarrow p+a$ **sinon** $p \leftarrow p+1$
`JMN` :
  si $B[p+b] \neq 0$ **alors** $p \leftarrow p+a$ **sinon** $p \leftarrow p+1$
`CMP` :
  si $m_A =$ immédiat **alors** $t \leftarrow A[p] = B[p+b]$
  sinon $t \leftarrow O[p+b] = O[p+a] \wedge A[p+b] = A[p+a] \wedge B[p+b] = B[p+a]$
  si t est vrai **alors** $p \leftarrow p+2$ **sinon** $p \leftarrow p+1$
`SLT` :
  si $m_A =$ immédiat **alors** $t \leftarrow A[p] < B[p+b]$ **sinon** $t \leftarrow B[p+a] < B[p+b]$
  si t est vrai **alors** $p \leftarrow p+2$ **sinon** $p \leftarrow p+1$
`DJN` :
  si $m_B =$ immédiat **alors** $v \leftarrow B[p]$ **sinon** $B[p+b] \leftarrow B[p+b] - 1$; $v \leftarrow B[p+b]$
  si $v \neq 0$ **alors** $p \leftarrow p+a$ **sinon** $p \leftarrow p+1$
`SPL` :
  $p_1 \leftarrow p+1$
  $p_2 \leftarrow p+a$
  (voir le paragraphe intitulé **Processus**)

FIG. 1 – Exécution d'une instruction.

2 Le langage Redcode

Les programmes s'affrontant dans Core War sont écrits dans un langage source appelé Redcode, de type assembleur. Pour ce projet, on simplifie à l'extrême ce langage. Un source Redcode est constitué de lignes qui sont soit des commentaires, soit des opérations.

$$\textit{ligne} \quad ::= \quad \textit{commentaire} \mid \textit{opération}$$

Un commentaire est introduit par le caractère ; et le reste de la ligne est alors ignoré. Une opération est une ligne contenant une étiquette optionnelle, une instruction, une opérande, une seconde opérande optionnelle, et enfin un commentaire optionnel.

$$\begin{array}{lcl} \textit{opération} & ::= & \textit{étiquette}^? \ \textit{instruction opérande commentaire}^? \\ \textit{opération} & ::= & \textit{étiquette}^? \ \textit{instruction opérande} \ \texttt{,} \ \textit{opérande commentaire}^? \end{array}$$

Une étiquette est un identificateur, formé de caractères alphanumériques. Les onze instructions s'écrivent en majuscules.

$$\textit{instruction} ::= \texttt{DAT} \mid \texttt{MOV} \mid \texttt{ADD} \mid \texttt{SUB} \mid \texttt{JMP} \mid \texttt{JMZ} \mid \texttt{JMN} \mid \texttt{CMP} \mid \texttt{SLT} \mid \texttt{DJN} \mid \texttt{SPL}$$

Une opérande est un mode optionnel suivi d'une valeur.

$$\textit{opérande} ::= \textit{mode}^? \ \textit{valeur}$$

Un mode d'adressage est indiqué par le caractère `#` (mode immédiat), `@` (indirect) ou `<` (indirect avec pré-décrémentation). L'absence de mode signifie le mode direct.

$$\textit{mode} ::= \texttt{\#} \mid \texttt{@} \mid \texttt{<}$$

Une *valeur* est une constante entière, éventuellement précédée d'un signe `-`, ou une étiquette.

$$\textit{valeur} ::= \textit{entier} \mid \texttt{-}\,\textit{entier} \mid \textit{étiquette}$$

Lorsqu'une seule opérande est donnée dans une opération, il s'agit de l'opérande A et l'opérande B vaut alors `#0`.

Les étiquettes sont une facilité pour écrire des adresses (toujours relatives à l'instruction courante) de manière symbolique. Les étiquettes de deux instructions différentes doivent être différentes. Une étiquette apparaissant dans une opérande doit être l'étiquette d'une instruction du programme. La valeur d'une étiquette dans une opérande est la distance relative à l'instruction portant cette étiquette.

3 Travail demandé

Le but de ce projet est de réaliser deux outils, en paratageant autant de code que possible entre ces deux outils. Ces deux outils sont les suivants :

1. un *arbitre* qui prend en argument N joueurs, sous la forme de N fichiers contenant leur source Redcode, organise un tournoi où chaque joueur affronte tous les autres selon les règles données ci-dessus, puis affiche le score final de chaque joueur ;

2. une *interface graphique* qui permet de visualiser l'exécution de un ou deux joueurs, dans le but de mettre au point un programme.

Les objectifs de ces deux programmes ne sont pas les mêmes. L'arbitre doit être le plus efficace possible, afin qu'un tournoi puisse être effectué dans un temps raisonnable. Un tournoi entre 5 joueurs représentent 400 matchs, soit potentiellement 400 millions de tours. Le temps d'exécution d'un tel tournoi de devrait pas excéder quelques minutes.

L'interface graphique ne doit pas nécessairement être aussi efficace. En revanche, elle doit permettre une mise au point agréable d'un programme, ou encore de suivre facilement le combat entre deux programmes. Cela inclut une exécution pas-à-pas, une visualisation du contenu de la mémoire, de l'emplacement de chaque processus, etc.

Des exemples de programmes Redcode seront donnés sur la page de suivi du projet [1]. Mais il est évidemment possible (et encouragé) d'écrire quelques autres programmes, en particulier pour les comparer avec ceux d'autres élèves ayant choisi ce projet.

Références

[1] Page de suivi du projet. `http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/10-11/INF431/core_war/`