MarioUS
Plombier génétique

IFT615
Intelligence artificielle

IFT630
Processus concurrents et parallélisme

Présenté par
Gabriel P. Girard – 07 173 738
Marc-Alexandre Côté – 07 166 997
Simon Renaud-Deputter – 07 149 640

Présenté à
Froduald Kabanza
Gabriel Girard

Université Sherbrooke
16 avril 2009

Problématique

Ce projet a pour but de nous familiariser avec les algorithmes génétiques parallélisés. Nous avons décidé d'en créer un qui apprendra à résoudre un niveau d'un jeu nommé : Super Mario Bros.

Super Mario Bros est un jeu de la compagnie Nintendo et est un des premiers à être développé pour leur console appelée : NES. Il est très simple d'apprendre le fonctionnement de ce jeu sorti en 1985. Le but du jeu est d'atteindre un drapeau qui désigne la fin du niveau. Pour y parvenir, le joueur devra, à l'aide d'une manette, éviter une multitude d'ennemis, sauter au-dessus des trous et sauter par-dessus des tuyaux. Le joueur est chronométré et doit terminer le niveau dans le temps qui lui est alloué. Il existe de nombreux niveaux que le joueur devra affronter afin de terminer le jeu.

Dans notre projet, plusieurs éléments ont été omis afin de nous faciliter la tâche (principalement par manque de temps). Les éléments qui diffèrent du jeu original sont : les briques incassables, l'absence de plusieurs types d'ennemis, l'absence du chronomètre et l'absence d'items. Afin de pouvoir suivre l'évolution de notre Mario, nous avons conçu un moteur de jeu 2D permettant de simuler Super Mario Bros.

L'objectif de notre projet est de trouver une suite d'actions permettant d'atteindre le drapeau tout en évitant les obstacles immobiles et les ennemis mobiles. Il doit aussi minimiser le nombre d'actions nécessaire pour parvenir au drapeau.

Instructions

*tous les paths sont par rapport au répertoire racine "projet/".

Notre projet est codé en C++ ainsi qu'en Java. Il est divisé en trois exécutables.

Générer les actions

Tout d'abord, il faut générer une liste d'action en utilisant cette commande à partir du répertoire "bin/":

```
"mpi/bin/mpiexec -l -n 5 sigma"
```

Les générations seront affichées à l'écran avec leur distance parcourue par rapport à la distance totale ainsi que leurs nombres d'actions correspondants. À chaque fois qu'une nouvelle liste d'actions possède un *fitness* plus haut que la meilleure liste d'actions, elle est sérialisée. Une fois que la distance parcourue est maximale et que le nombre d'actions converge, l'application termine. Ensuite, le répertoire "actions/" contiendra toutes les meilleures listes d'actions pour une partie du niveau.

Générer les logs

Deuxièmement, il faut générer un fichier ".log" si l'entrainement de l'IA n'a pas convergé (c.-à.-d. ne s'est pas terminé correctement). Pour ce faire, il faut utiliser cette commande à partir du répertoire "bin/":

```
"engine nomTableau nomFichierActions nomFichierLog"
nomTableau : Nom d'un tableau dans le dossier "levels/"
nomFichierActions : Nom d'un fichier d'actions dans "actions/"
nomFichierLog : Nom du fichier log à générer.

ex: engine levels/tableau1.lvl actions/action_généré.act logs/fichier_log.log
```

Visualiser le résultat

Finalement, le fichier est chargé par le visionneur pour visualiser le Mario génétique. Pour ce faire, il faut double cliquer sur "bin/viewer/viewer.jar". Ou exécuter par ligne de commande à partir du répertoire "bin/viewer/":

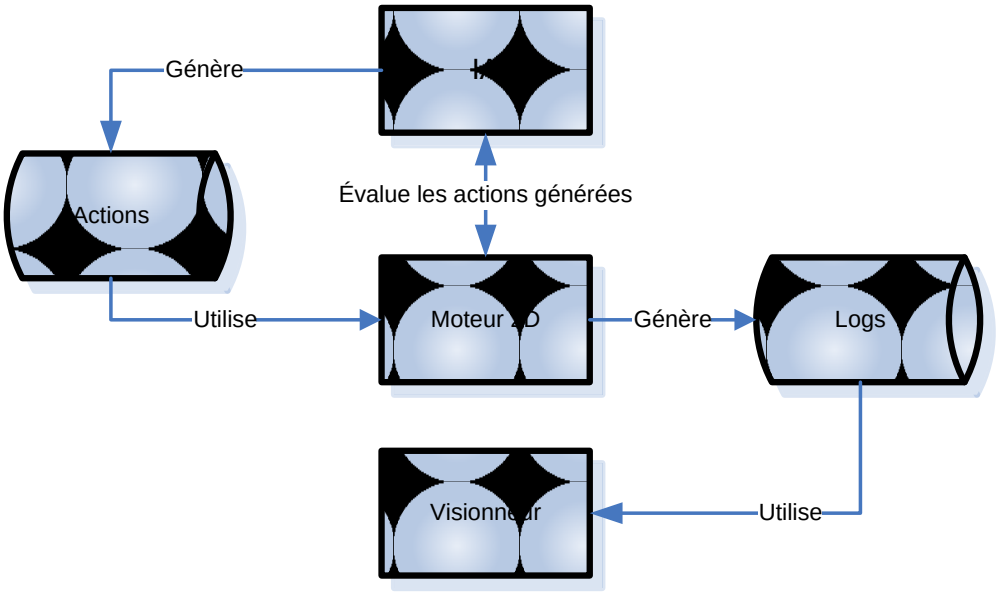
```
"java –jar viewer.jar nomFichierLog.log"
nomFichierLog.log : Nom du fichier log à visionner.
```

Ensuite, cliquer sur "Fichier->Ouvrir log".

Mise en oeuvre

Ce type de problèmes nécessite d'immenses graphes afin de représenter tous les états possibles. De plus, ces graphes doivent être dynamiques puisque les ennemis peuvent bouger dans le niveau. Pour parvenir à trouver une solution efficace dans un délai acceptable, notre équipe a décidé d'opter pour l'utilisation d'un algorithme génétique.

Le projet est divisé en trois parties (programmes) distinctes, soit : un moteur 2D, un visionneur de logs et une intelligence artificielle dirigeant Mario.



Moteur 2D

Le moteur est utilisé par l'algorithme génétique afin d'évaluer une liste d'actions. Les valeurs de retour sont : la distance parcourue, la distance totale et le nombre d'actions utilisées. Il sert aussi à générer un log contenant la position de tous les objets de tous les frames (image d'une animation). À chaque frame, le moteur lit une action et l'exécute. Le moteur est complètement indépendant de l'affichage graphique ce qui permet d'accélérer les calculs.

Visionneur

Un visionneur est utilisé pour visualiser la progression de notre Mario génétique. Il permet aussi de visionner des niveaux et d'ajuster la vitesse de visionnement. Il commence par charger l'image contenant tous les sprites (images d'un bonhomme). Ensuite, il charge le niveau, qui sera toujours gardé en mémoire. Finalement, il charge les frames puis les fait jouer une après l'autre ce qui crée une animation.

IA

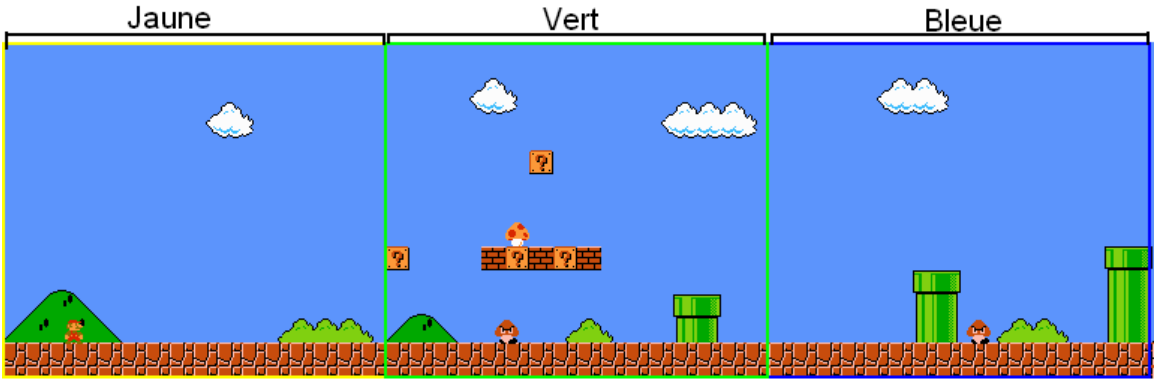
Notre intelligence artificielle utilise un algorithme génétique afin de trouver le chemin le plus efficace pour terminer un niveau.

Tout d'abord notre ADN est composé d'une suite d'actions. Une action est représentée par un « char » où chaque bit correspond à un bouton enfoncé. Cela nous permet d'avoir des compositions d'actions.

Un bon fitness pour notre algorithme signifie que la distance parcourue est maximisée et que le nombre d'actions est minimisé. La distance parcourue se calcule par une sommation de chaque pixel à partir du début du tableau jusqu'à la position finale (en X) du joueur [ex. Position finale = 6; Distance parcourue = 1 + 2 + 3 + 4 + 5 + 6 = 21]. Cela permet d'avantager les individus qui se sont rendus plus loin.

Nous utilisons le principe nommé *King of the Hill* pour résoudre notre optimisation. Son fonctionnement consiste à garder le meilleur individu d'une génération afin d'entrainer les prochains individus des prochaines générations. Cet élu n'est changé que s'il y en a un plus fort (dans notre cas, celui qui s'est rendu plus loin ou qui est plus rapide).

Après l'avoir longuement testé, nous avons décidé d'améliorer notre algorithme génétique en divisant un niveau en plusieurs tranches (environ un écran de large). Notre IA commence par optimiser les deux premières tranches (donc les deux premiers écrans). On arrête l'optimisation lorsque l'élu d'une génération ne s'est pas fait battre (rapport entre distances parcourues et le nombre d'actions utilisées) pendant un nombre de générations prédéfini (présentement 1000 générations). Ensuite, on décale d'une tranche et on optimise sur les deux prochaines. L'algo continue jusqu'à ce que toutes les tranches soient optimisées.



Chaque fenêtre représente une tranche. L'algorithme commence par optimiser la fenêtre jaune et la fenêtre verte. Par la suite, il va optimiser la fenêtre verte et la fenêtre bleue et ainsi de suite.

Cheminement IA

Au cours de notre projet, nous avons essayé de modifier plusieurs paramètres de notre algo afin de trouver les bonnes valeurs.

Première version :

- Principe : King of the Hill
- Utilisait toutes les actions de bases.
- Le niveau ne contenait pas d'ennemis.
- Résultat : notre Mario ne réussissait pas à terminer le niveau.
- Temps pour générer une solution : temps infini (il fallait l'arrêter à main).
- Nombre d'actions : environ 2000.

Deuxième version :

- Principe : King of the Hill
- Utilisait seulement deux actions : **Courir-Sauter-Avant et Courir-Avant**.
- Le niveau contenait des ennemis.
- Résultat : notre Mario réussissait à terminer le niveau.
- Temps pour générer une solution : temps infini (il fallait l'arrêter à main).
- Nombre d'actions : environ 600.

Troisième version :

- Principe : King of the Hill et **Fenêtre coulissante**
- Utilisait seulement deux actions : Courir-Sauter-Avant et Courir-Avant.
- Le niveau contenait des ennemis.
- Résultat : notre Mario réussissait à terminer le niveau.
- Temps pour générer une solution : environ 15 min.
- Nombre d'actions : 456 (minimum = 454).

Quatrième version :

- Principe : King of the Hill et Fenêtre coulissante
- Utilisait **toutes les actions de bases**.
- Le niveau contenait des ennemis.
- Résultat : notre Mario réussissait à terminer le niveau.
- Temps pour générer une solution : environ 2 heures.
- Nombre d'actions : 780.

Nous avons décidé de paralléliser notre algorithme en utilisant le principe Maître-Travailleur dans le but d'accélérer la convergence vers une solution. Nous avons implémenté ce code en MPI (C++), puisqu'il nous permet de distribuer facilement et rapidement des calculs sur une grappe d'ordinateurs.