Projet PComp

Un projet par David Galichet, Julien Sagot, Nicolas Cailloux, et Baptiste Fontaine dont les contraintes étaient :

- Utiliser Scheme
- Utiliser un maximum de bibliothèques

Approche

La seconde contrainte n'est pas dérangeante, car il est naturel d'utiliser le plus de bibliothèques possibles pour ne pas réinventer la roue. La première en revanche nous a poussé à réaliser un système très modulaire car nous ne voulions pas coder tout le programme en Scheme.

Implémentation

Le cœur de notre projet est écrit en Go, choisi pour ses capacités de bas niveau (à la C), son typage fort et sa facilité à lancer des threads légers, sur lesquels nous reviendrons après. Ce cœur est organisé en plusieurs modules : le premier (`api/io.go`) est chargé de la communication avec le serveur distant. Il implémente toutes les manipulations de bas niveau comme la gestion transparente des cookies et de l'encodage des requêtes. Sur celui-ci s'appuie un client de haut niveau pour l'API (`api/client.go`) qui se charge de construire les requêtes à l'API et interpréter ses réponses. Nous utilisons ici les structs de Go pour représenter toutes les entités. Un jeu est représenté par la struct `Game`, son statut par `GameStatus`, les infos sur l'API par `APIInfos`, etc.

Nous avons donc été confrontés au problème de l'interface entre Go et Scheme. Plutôt que de tenter d'appeler du code Scheme via Go ou inversement, nous utilisons des programmes complètement séparés qui communiquent entre eux. Au dessus du client écrit en Go, nous utilisons un serveur de jeu local (`api/server.go`) qui se charge de créer ou rejoindre une partie existante, et lance une ou plusieurs intelligence(s) artificielle(s) sous forme de programme externe. L'un d'eux est écrit en Scheme (`ai/scout.scm`) mais tous les langages sont supportés. Nous avons ainsi une IA en Ruby (`ai/ant.rb`) et une en Shell (`ai/randombot.sh`). Le serveur de jeu local utilise la notion d'acteurs (`api/actors.go`) pour lancer les IA et communiquer avec elles. Lors du lancement, il crée un acteur par IA, et ouvre deux canaux de communication avec cet acteur, l'un pour lui envoyer des messages et l'autre pour en recevoir. L'acteur lance son programme d'IA, relaie les messages qu'il reçoit du serveur sur l'entrée standard du programme et récupère sa sortie standard pour la transmettre au serveur. Ces messages suivent un protocol précis mais simple (documenté dans `docs/ai_protocol.md`). Les programmes d'IA reçoivent les informations de chaque tour et répondent par une commande. Le serveur local combine ces commandes et les envoie au serveur distant. Il maintient les informations de la

carte et les envoie à chaque tour aux IA, ce qui permet d'avoir des programmes d'IA très légers puisque sans état.

Ce principe d'acteurs externes a été généralisé pour pouvoir brancher n'importe quel plugin sur le serveur de jeu. Nous avons par exemple une interface graphique écrite en OCaml (`gui/monitor.ml`) qui reçois les informations de chaque tour et dessine la carte.

Ces acteurs fonctionnent en parallèle du serveur local, et celui-ci n'a pas d'état externe (pas de fichier de cookies par exemple) donc il est possible de lancer plusieurs serveurs locaux en même temps.

Tout le code Go précédemment décrit est exposé sous la forme d'une bibliothèque, utilisée par le programme `antroid.go` situé à la racine du répertoire. Celui-ci permet de lancer un serveur de jeu et d'utiliser tous les appels à l'API pour créer une partie, afficher la liste des parties visibles, rejoindre ou créer une partie, etc.

Tests

Les couches les plus basses (et donc les plus stables) du code Go disposent de tests unitaires. Seul 30% du code est couvert par les tests mais les parties les plus haut niveau sont les moins stables donc ce serait une perte de temps d'écrire des tests pour du code qui change tout le temps. Sur les 2100 lignes de code Go, près de 1000 servent exclusivement aux tests unitaires.

Documentation

Le code est commenté, le fichier `README.md` explique comment mettre en place son environnement de travail, et le fichier `docs/hacking.md` décrit l'organisation du code. Tout, à l'exception de ce rapport, est écrit en anglais. L'ensemble du code se veut clair et respecte les conventions du langage dans lequel il est écrit. Il est découpé en plusieurs fichiers, chacun ne contenant du code que pour une partie distincte.