TP 2 mai 2024

Langages Formels – Projet Analyse Syntaxique

Le projet tourne autour du problème des prisonniers. Deux individus se rencontrent et ils ont un choix indépendant de tricher ou d'être honnête. Si les deux sont honnêtes, ils gagnent 3 points chacun. Si les deux trichent, ils gagnent 1 point chacun. Si une personne triche et l'autre est honnête, le tricheur gagne 5 points et l'autre 0 points. Ici, on considère des *interactions* qui consistent de plusieurs itérations du jeu, où chaque individu peut avoir une stratégie en fonction des actions précédentes. Par exemple, la stratégie *donnant-donnant* commence par être honnête, puis répète la réponse précédente de l'autre joueur ; une stratégie *rancunière* commence par être honnête, mais une fois trompé, le joueur va toujours tricher.

Ces stratégies peuvent se décrire dans un langage de programmation très simple, voici un exemple (un exemple plus complet est fourni sur la page web du cours) :

```
Strategy Nice (3)
        return Honest

Strategy TitForTat
        if last == Undef return Honest else return last

Strategy Grim
        def decision = Honest
        if last == Cheat decision = Cheat;
        return decision

Strategy Rand
        if Random(2) < 1 return Cheat else return Honest

Constants
 duration = 10
 rewardHH = 3 rewardHC = 0
 rewardCH = 5 rewardCC = 1
```

Toute stratégie est censée renvoyer une valeur `Cheat` ou `Honest`, en fonction de son état actuel. Toute stratégie possède une variable implicite `last` dont la valeur est initialement `Undef` puis contient la dernière décision de l'autre joueur. On peut déclarer d'autres variables de type entier avec le mot-clé `def`. Les valeurs `Undef`, `Cheat` et `Honest` sont des entiers symboliques. L'expression `Rand(n)` donne une valeur entière entre 0 et $n-1$. Ainsi, `TitForTat` représente un joueur donnant-donnant, `Grim` un joueur rancunier etc. Un stratégie peut avoir un poids optionnel (3 dans `Nice`), qui sera expliqué plus tard. Si aucun poids n'est donné, il est 1.

La section `Constants` donne le nombre de jeux dans une interaction (`duration`) et les recompenses en fonction des choix des joueurs.

1 Objectifs

En fonction de vos ambitions, vous pouvez implémenter soit un projet de base, soit un projet plus intéressant ; la notation prendra bien sûr compte de l'objectif que vous avez choisi. Les choix sont ainsi :

— Projet de base : Faire une analyse syntaxique d'un fichier, le transformer en un arbre syntaxique et afficher l'arbre syntaxique sur l'écran.
— Implémentation des stratégies : Simuler une seule interaction entre deux individus avec leurs stratégies.
— Simulation d'une population : Gérer toute une population d'individus qui se multiplient en fonction de leur récompenses.

Vous pouvez travailler **en groupes de deux** et utiliser les fichiers du TP (notamment `langlex.l` et `lang.y`) comme base ; ces fichiers contiennent déjà quelques structures utiles, mais il faudra les adapter.

Modalités de rendu. La date de rendu sera fixé ultérieurment (en fonction de vos retours à ce sujet). Envoyez un mail à `schwoon@lmf.cnrs.fr` avec un seul archive (format zip ou tgz) qui contient les fichiers sources ainsi qu'un readme avec toutes les explications qui vous paraissent utiles, et surtout quel objectif vous avez choisi. Si vous travaillez en groupes de deux, la personne qui envoye le mail mettra son parténaire en copie.

2 Analyse syntaxique

L'analyse syntaxique doit être implémenté par tous les groupes, mais l'affichage de l'arbre syntaxique n'est exigé que pour les projets de base.

La syntaxe et sémantique du langage pour les stratégies ne sont pas fixées formellement – il est à vous de ce faire en créant une grammaire. Votre grammaire devrait traiter au moins les fichiers fournis en exemple, mais vous pouvez y apporter des améliorations (à spécifier dans votre rapport). Vous devez aussi traiter le problème du "dangling else" (une branche `else` appartient au dernier `if` qui n'a pas encore de `else` correspondant).

Dans la section de constants il est permis d'en omettre certains ; dans ce cas votre programme va leur affecter des valeurs standard.

Votre grammaire ne doit comporter aucun conflit de type "shift-reduce" ou "reduce-reduce" ; tous ces conflits sont à éliminer avant le rendu.

Tâches pour le projet de base :

— Tester si un fichier donné est syntaxiquement correct.
— Créer un arbre syntaxique qui représente les stratégies et les constants en mémoire.
— Reproduire les stratégies et constants sur l'écran, de façon sémantiquement équivalente. À noter que cet affichage être réalisé à partir de l'arbre syntaxque, notamment, il n'est pas permis de réaliser cet affichage *pendant* l'analyse syntaxique.

3 Implémentation des stratégies

Réaliser un programme qui prend en compte un fichier qui décrit de différentes stratégies et permet d'en choisir deux (pas nécessairement distinctes). Votre programme simule alors une interaction entre deux individus avec ces stratégies, affiche leurs décisions dans chaque tour et compte le nombre de points qu'ils gagnent.

À votre choix, votre programme peut permettre de choisir le fichier et les stratégies sur la ligne de commande pour en réaliser une seule interaction, ou de faire plusieurs interactions entre différentes strategies avec une interface interactive.

4 Simulation d'une population

Il s'agit de gérer une population d'individus. Pour ce faire, on va ajouter des constants supplémentaires :

```
Constants
 initial = 1000
 meetings = 10000   intervals = 100
 life = 10    spawn = 180    mutation = 10
```

Ici, `initial` est la taille initiale de la population. Le stratégie de chaque individu est choisie aléatoirement en fonction des poids (p.ex. une strategie avec poids 3 est choisie trois fois plus souvent qu'une avec poids 1). Ainsi, un poids de 0 sert pour désactiver une stratégie pour une simulation.

Si `meetings` égale m et `intervals` égale n, une simulation consiste de $m \cdot n$ interactions, avec des statistiques sur la population affichées tous les m tours. Dans chaque interaction, deux individus sont choisis aléatoirement dans la population. Un individu meurt après `life` interactions. Pour tous les `spawn` points gagnés, un autre individu est créé. À priori ce nouveau-né possède la même stratégie que son progéniteur, mais avec une probabilité de `mutation` pourcent il choisit une autre stratégie aléatoirement.

Les informations affichées sur la population incluent la taille actuelle de la population, la distribution des stratégies, et le nombre de points moyen gagné par les individus pendant leurs vies.

Note : Afin d'éviter une explosion de la population, on peut aussi adapter dynamiquement le seuil de points pour créer un nouvel individu ; p.ex. multiplier `spawn` par un facteur taille de la population actuelle divisée par taille de population initiale.

5 Améliorations diverses

Cette partie est optionnel, mais vous pouvez étendre le nombre de stratégies (et étendre le langage si nécessaire), p.ex. en vous inspirant des sources suivantes :

— `https://www.cristal.univ-lille.fr/~jdelahay/pls/242.pdf`
— `https://plato.stanford.edu/entries/prisoner-dilemma/strategy-table.html`