Programmation fonctionnelle de λman

Version 4ef3b21 – 12 novembre 2019

Contexte Nous sommes en l'an de grâce 4273, la conquête spatiale fait rage. Les humains étant incapables de résister aux vitesses quasi-luminiques, ils envoient des robots programmés en OCaml pour coloniser ces nouveaux eldorados. Leur mission est simple : récupérer un maximum d'arbres de Böhm, la matière première alimentant les vaisseaux spatiaux capables de se déplacer dans toute la galaxie et de découvrir de nouvelles planètes remplies de nouveaux arbres de Böhm [1].

Fonctionnement du projet Ces robots sont appelés des λmen. Votre projet consiste à programmer le module de prise de décisions de ces robots pour récolter un maximum d'arbres de Böhm.

Ce projet est décomposé en plusieurs tâches à effectuer en binôme. Les tâches 0 à 3 permettent d'avoir jusqu'à 12/20 et devraient pouvoir être réalisées en une dizaine d'heures.

Le projet sera évalué par une soutenance finale dont la date vous sera communiquée ultérieurement. Lors de cette soutenance, vous devrez nous prouver que vous êtes bien les auteurs du code source de votre projet.

Une attention particulière sera portée sur votre style de programmation. Votre code source devra en effet être écrit dans un style fonctionnel, ce qui suppose par exemple qu'il doit être globalement composé de petites fonctions générales, inductives et (observationnellement) pures. Le code source devra être indenté et écrit pour être compréhensible. Bien entendu, un projet qui ne compile pas obtiendra la note minimale.

Pour vous aider, chaque tâche est accompagnée d'un ensemble de tests.

Enfin, pour toute question relative au projet, utilisez la liste de diffusion disponible à l'URL suivante :

`https://listes.univ-paris-diderot.fr/sympa/info/pf5-2019`

L'inscription à cette liste de diffusion est obligatoire.

Tâche 0 : λman naît

La première étape du projet consiste à créer votre dépôt GIT à partir du code à trous que nous vous fournissons. Pour cela, l'un des deux membres du binôme doit "forker" le projet GitLab suivant en précisant que **votre dépôt doit être privé** :

`https://gaufre.informatique.univ-paris-diderot.fr/yrg/lambda-man`

Ensuite, vous devez rajouter le binôme et le compte `yrg` dans les membres du projet en leur donnant les droits de mainteneur.

Une fois que votre projet GitLab est configuré, clonez le dépôt GIT associé sur votre machine :

```
% git clone https://gaufre.informatique.univ-paris-diderot.fr/VOTRE-LOGIN/lambda-man
```

Pour développer votre projet, votre machine doit utiliser une distribution GNU/Linux et avoir OPAM 2 installé. Si c'est bien le cas, vous pouvez exécuter la commande suivante depuis la racine du dépôt :

```
% ./configure
```

Ce script va installer les dépendances du projet dans une installation OPAM dédiée à votre projet et stockée dans le répertoire `_opam` à la racine de votre dépôt. Si tout s'est bien passé, vous devriez obtenir le message :

```
Congratulations! Your project is ready for development.
```

La dernière étape de ce jalon consiste à enregistrer votre projet dans la liste des projets. Après avoir poussé sur votre dépôt la mise-à-jour du fichier `authors.json`, il suffit de lancer :

1. Un milliers d'années plus tard, les historiens appeleront "ère Ω", cette période de croissance frénétique autojustifiante.

```
% ./register
```

Vous devriez alors obtenir le message suivant, qui valide votre réussite de ce jalon :

```
Congratulations! Your lambda-man is registered.
Its name is XXXXX and its password is XXXXX.
```

Notez l'identifiant et le mot de passe de votre λman.

Intermezzo : Présentation du code source

Le projet est formé de modules. Vous êtes invités à lire leur code source mais il n'est pas nécessaire de tout comprendre pour réaliser le projet : pour certains modules, comprendre uniquement l'interface d'utilisation suffit ; d'autres modules peuvent tout simplement être ignorés. On précise le rôle et le statut de l'ensemble des modules dans la liste suivante :

- `LambdaDriver` : module d'entrée de l'exécutable.
  (Ni à modifier, ni à comprendre.)
- `LambdaMan` : client du jeu.
  (Ni à modifier, ni à comprendre.)
- `LambdaServer` : serveur du jeu.
  (Ni à modifier, ni à comprendre.)
- `Communication` : couche de communication client-serveur.
  (Ni à modifier, ni à comprendre.)
- `Game` : boucle du jeu.
  (Ne pas modifier mais à lire pour comprendre les règles du jeu.)
- `Visualizer` : visualisateur du jeu.
  (À modifier dans la tâche 6.)
- `World` : représentation du monde.
  (Ne pas modifier mais comprendre son interface pour l'utiliser.)
- `Space` : représentation des ensembles de polygones.
  (Ne pas modifier mais comprendre son interface pour l'utiliser.)
- `PriorityQueue` : structure de données de file de priorité.
  (Ne pas modifier mais comprendre son interface pour l'utiliser.)
- `Graph` : structure de données de graphe.
  (Ne pas modifier mais comprendre son interface pour l'utiliser.)
- `Ext` : ensemble de fonctions utiles.
  (Ne pas modifier mais à utiliser éventuellement.)
- `WorldGenerator` : générateur de mondes.
  (À modifier pour la tâche 5.)
- `Decision` : prise de décision des robots.
  (À modifier pour les tâches 1, 2, 3 et 4.)

Il est conseillé d'explorer le code source du projet avant de commencer à programmer. En effet, le code contient beaucoup de commentaires qui expliquent les règles du jeu et son principe de fonctionnement. Vous pouvez aussi le tester dès maintenant. Pour cela, il faut d'abord le compiler avec la commande :

```
% make
```

Cela devrait produire un exécutable nommé `lambda` à la racine de votre dépôt. Cet exécutable est à la fois le serveur et le client de jeu : on détermine son rôle en fonction de l'argument passé en ligne de commande. Typiquement, la commande :

```
% ./lambda server -s 0.1 -v -w tests/00001-simple.json './lambda man -n 1 -v'
```

lance un serveur en utilisant le monde de tests `00001-simple.json` distribué avec les sources. Entre guillemets simples, on a donné au serveur la commande à lancer pour exécuter le client : ici, il se relance avec l'argument `man`. L'option `-n 1` du client indique qu'il ne doit prendre en charge qu'un seul robot. Les options `-v` permettent de visualiser le monde du point de vue du robot et du serveur. Enfin, l'option `-s 0.1` permet de ralentir le serveur pour avoir le temps d'observer la partie quand elle se déroule trop rapidement : ici on indique au serveur d'attendre 100ms entre chaque tour.

Si le monde contenait plusieurs équipes, il faudrait rajouter d'autres commandes pour décrire comment appeler les clients de chaque équipe. Si le monde avait plus d'un robot par équipe, il faudrait modifier l'argument de l'option `-n` passée au client.

En attendant de tester ces appels sophistiqués, la commande précédente doit déjà fonctionnée. Par contre, vous devez avoir noté qu'il fuit lâchement ! C'est donc le moment de commencer la tâche 1.

Tâche 1 : λman s'entraine dans le désert

Ce premier jalon sert à vous familiariser avec le code source du projet en vous attaquant à une version simplifiée du problème : votre robot est dans le désert et peut se déplacer vers les arbres de Böhm à récolter sans se poser de question car on n'y trouve aucune bouche de l'enfer.

Vous devez donc mettre à jour les fonctions `Decision.plan` et `Decision.next_action` pour vous assurer que le robot récupère l'ensemble des branches du monde. N'oubliez pas de ramener ses branches dans votre vaisseau spatial pour obtenir le bonus.

Ne prenez pas trop de temps pour répondre : le serveur vous laisse 100ms, pas une de plus !

Vous pouvez tester votre réponse sur les tests de la forme `tests/xxxxx-simple.json`.

Tâche 2 : λman évite les bouches de l'enfer

Nous rentrons maintenant dans le vif du sujet : votre robot est maintenant dans des contrées terribles où il ne faut pas tomber dans les bouches de l'enfer : on ne peut plus se contenter de se diriger en ligne droite vers les arbres de Böhm car il faut maintenant trouver un chemin "sûr".

Pour cela, comme l'expliquent les commentaires du module `Decision`, il faut se construire une carte puis déterminer un chemin efficace pour se rendre au prochain arbre. Comme le robot découvre le monde progressivement, il faut faire attention à vérifier que le chemin suivi est toujours valide.

Vous devez donc mettre à jour les fonctions `Decision.visibility_graph`, `Decision.shortest_path`, `Decision.plan` et `Decision.next_action` pour réaliser cette tâche.

Vous pouvez tester votre réponse sur les tests de la forme `tests/xxxxx-hell.json`.

Tâche 3 : λman traverse les champs de la souffrance

Si les bouches de l'enfer détruisent votre robot à coup sûr, les champs de la souffrance peuvent le torturer très cruellement ! Quand votre robot traverse ces champs, sa vitesse est modifiée : cela influence donc la valeur d'un chemin s'il croise l'un de ces champs ! Saurez-vous en profiter ?

Vous devez donc mettre à jour les fonctions `Decision.visibility_graph`, `Decision.shortest_path`, `Decision.plan` et `Decision.next_action` pour réaliser cette tâche.

Vous pouvez tester votre réponse sur les tests de la forme `tests/xxxxx-hell-and-suffering.json`.

Tâche 4 : λman travaille en équipe

Un seul robot ne peut récolter seul tous les arbres de la planète. C'est pour cela que les humains envoient plusieurs robots à la fois. Pour collaborer, les robots doivent communiquer mais sans réseau de communication, leur seule solution consiste à laisser des microcodes sur le sol. Grâce à ces microcodes, les robots peuvent se répartir les arbres et aussi partager des "bons plans".

Vous devez donc mettre à jour les fonctions `Decision.visibility_graph`, `Decision.shortest_path`, `Decision.plan` et `Decision.next_action` pour réaliser cette tâche.

Vous pouvez tester votre réponse sur les tests de la forme `tests/xxxxx-pp-team.json`. Attention, `pp` correspond au nombre de robots attendus par le niveau : passer ce nombre au client grâce à l'option `-n` pour tester votre programme. Vous devez chercher à réduire le nombre de tours nécessaires à la résolution de niveau.

Tâche 5 : À votre tour de créer des mondes

À ce stade, vos robots sont déjà capables de résoudre les mondes de tests que nous vous avons fournis. Seulement, ces mondes sont très simples et on peut en imaginer des plus complexes.

Vous devez mettre à jour le module `WorldGenerator` pour lui faire produire des mondes plus compliqués.

Vous pouvez soumettre vos mondes les plus intéressants en les copiant dans un répertoire `myworld` et en les poussant sur votre `git`. Si votre robot est capable de résoudre un de ces mondes alors ce monde sera utilisé par la compétition finale.

Tâche 6 : Visualisation

Le visualisateur n'est pas très esthétique. N'hésitez pas à l'améliorer si vous le souhaitez ! Ces améliorations vous apporteront des points bonus.

Tâche 7 : À la conquête de l'Univers !

Vos robots sont maintenant prêts pour la compétition : celle-ci se déroulera sur les mondes proposés par vous et vos camarades.

Voici quelques éléments supplémentaires pour vous permettre de prendre l'avantage.

— L'algorithme naïf de calcul du graphe de visibilité est de complexité cubique. C'est une catastrophe sur les graphes très gros ! Pour réussir à survivre dans les grands mondes, il va falloir améliorer cette situation. Vous pourriez par exemple construire ce graphe incrémentalement en mettant à jour le graphe à chaque nouvelle information plutôt que de le reconstruire à chaque tour. Vous pourriez aussi implémenter l'algorithme de Lee pour calculer le graphe de visibilité en temps $O(n^2 logn)$.

— L'algorithme de Dijkstra n'est pas le seul algorithme pour calculer un plus court chemin. Essayez d'en explorer d'autres comme $A^\star$ par exemple.

— Les algorithmes de colonies de fourmis permettent de profiter d'une communication "à même le sol" pour marquer les chemins avantageux. N'hésitez pas à vous en inspirer !