ÉCOLE POLYTECHNIQUE DE LOUVAIN

Année académique 2017-2018

Université Catholique de Louvain

LFSAB1402

Projet 1

Rapport BomberSnake Oz

Professeur : Van Roy Peter

Date de remise du rapport : 15 mai 2017

Noms et nomas des membres du groupe :

Nicolas Pire	4921-1200
Cyprien Janssens	5387-1600

1 Introduction

Dans le cadre de notre cours d'informatiques 2 donné par Peter Van Roy, il nous a été demandé de compléter un programme afin de permettre aux futurs utilisateurs de ce dernier de pouvoir jouer à BomberSnake-Oz, une version améliorée du jeu bien connu : Snake. Pour implémenter ce jeu, nous avions pour contrainte de n'utiliser que des constructions déclaratives.

2 Structure globale du jeu

Notre jeu se base sur un scénario qui contient des serpents ainsi que des stratégies. Le but de notre programme est de faire évoluer les serpents sur le plateau de jeu en fonction des instructions présentes dans la stratégie. Ainsi, notre fonction `DecodeStrategy` va lire la stratégie et envoyer des instructions à `Next` qui se chargera d'appliquer les instructions aux serpents afin de les faire bouger correctement et de leur appliquer les effets et bonus auxquels ils ont droit.

3 La fonction Next

Notre fonction `Next` se compose principalement d'une succession de constructions de type `if`. Elle reçoit en argument un serpent ainsi qu'une instruction, elle parcourt les conditions à la recherche d'une correspondance et renvoit directement le serpent qui doit être affiché au coup suivant. Afin de ne pas trop surcharger cette fonction, `Next` fait souvent appel à des fonctions supplémentaires telles que `AppendLists` ou `FlattenList`. Toutes ces sous-fonctions sont présentées et spécifiées dans notre fichier Code.oz.

4 La fonction DecodeStrategy

Notre fonction `DecodeStrategy` est composée d'une construction de type `case`. `DecodeStrategy` est de type récursive terminale, c'est à dire que la fonction fait appel à elle-même sans laisser un calcul en suspent, pour éviter d'utiliser trop de mémoire vive en même temps. Le `case` regarde si le premier élément de la liste passée en argument est une forme d'instruction compréhensible par le serpent (ces instructions sont sous la forme : `forward, turn(left), turn(right)`). Si oui, elle crée une liste avec, comme premier argument, l'appel à la fonction `Next` décrite ci-dessus. Si ce n'était pas une instruction, alors elle fait appel à une autre fonction, `DecodeRepeat`, qui est décrite dans notre fichier Code.oz.

Et enfin, `DecodeStrategy` s'appelle elle même pour analyser le deuxième élément de la liste passée en argument, et ainsi de suite jusqu'à la fin.

5 La complexité de nos fonctions

5.1 La fonction Next et ses sous-fonctions

`AppendLists` Cette fonction est récursive terminale et a donc une complexité de $O(n)$, avec n étant le nombre d'éléments dans la liste `L1`.

`DeleteLast` Cette fonction est récursive terminale et a donc une complexité de $O(n)$ avec n étant le nombre d'éléments dans la liste `L`.

`ReversedList` Cette fonction a une complexité de $O(n)$.

`FlattenList` Cette fonction est récursive terminale et a donc une complexité de $O(n)$, avec n étant le nombre d'éléments dans la liste `L`.

`InversedHead` Cette fonction est récursive terminale et a donc une complexité de $O(n)$ avec n étant le nombre d'éléments dans la liste `L`.

`ChangeDirection2` Cette fonction a une complexité de $O(1)$, n'effectuant que plusieurs `if`.

`ChangeDirection1` Cette fonction est récursive terminale et a donc une complexité de $O(n)$ avec n étant le nombre d'éléments dans la liste `L`.

`Next` Cette fonction n'étant composée que d'appels aux fonctions ci-dessus, `Next` a donc également une complexité de $O(n)$.

5.2 La fonction DecodeStrategy et ses sous-fonctions

`MultList` Cette fonction est récursive terminale et a donc une complexité de $O(n)$, avec n étant le nombre de fois que la liste doit être répétée, donc équivalent à l'argument `Times`.

`DecodeRepeat` Cette fonction, faisant appel une fois la fonction `MultList`, mais aussi à la fonction `DecodeStrategy` avec comme argument la liste d'instructions relatives au `repeat`, sa complexité est de $O(n^2)$.

`DecodeStrategy` Cette fonction, faisant une boucle avec `DecodeRepeat`, elle possède une complexité de $O(n^2)$.

6 Les extensions

— *Shrink* : Ce bonus permet de supprimer un élément du serpent. Ainsi, un serpent composé de 6 éléments ne sera plus composé que de 5 éléments après avoir bénéficié de cet effet. Attention, cet effet peut devenir mortel pour un serpent qui n'est plus composé que de deux éléments puisque s'il est composé de moins de deux éléments, le serpent meurt. Cette option peut donc être utilisée comme attaque vis à vis des adversaires et peut conduire à des parties beaucoup plus rythmées puisque un joueur devra en permanence faire attention à sa longueur pour éviter d'être tué par une trop grande succession de `Shrink` envoyée par les adversaires.

— *Pas de murs* : Ce bonus fait en sorte que toutes les cases telle que `x=1, x=22, y=1` et `y=22` se comportent comme des `Teleport` liées à la case qui se situe à l'autre extrémité du plateau de jeu. De cette manière, les serpent peuvent traverser les murs. Cette extension fait en sorte que les cases qui bordent le plateau soient toutes considérées comme des `Teleport`. Il n'est donc pas possible pour un serpent de longer le bord du plateau puisque aussitôt que la tête touchera un des bord, il sera envoyé de l'autre côté.

7 Les difficultés rencontrées

La première difficulté face à laquelle nous étions confrontés était dans un premier temps la discrétion du langage Oz dans le monde informatique, il n'est effectivement pas assez répandu que pour pouvoir trouver une aide concrète sur internet. Mais la date du projet avançant, les tuteurs se rendant largement disponibles et les sujets sur le forum se multipliant, nous sommes rapidement parvenus à trouver toute l'aide dont nous avions besoin.

Une de nos difficultés a été l'implémentation de la fonction `DecodeStrategy`. Tout d'abord nous avions créé une fonction qui utilisait une cellule. Cette cellule contenait la liste à renvoyer à la fin de `DecodeStrategy`, et à chaque lecture de l'argument passé à cette fonction, nous réassignions la cellule à une nouvelle liste contenant la précédente liste suivie de l'instruction décodée par `DecodeStrategy` (ou `DecodeRepeat`). Puisque seules les constructions déclaratives étaient autorisées, nous nous sommes ensuite penchés sur une alternative à l'utilisation de `Cell` (qui ne sont pas déclaratives). Nous avons donc dû repartir de (presque) zéro pour créer un `DecodeStrategy` complètement opérationnel.

Pour finir, nous avons également eu du mal à gérer l'interaction entre les différents effets. Nous nous sommes rendu compte en testant notre jeu que lorsqu'un serpent venait de finir de traverser un mur et que le bout de sa queue était encore sur une case du bord de type `Teleport` Si ce serpent prenait ensuite un effet `Revert`, alors la tète de se dernier finissait hors du plateau et il mourrait.

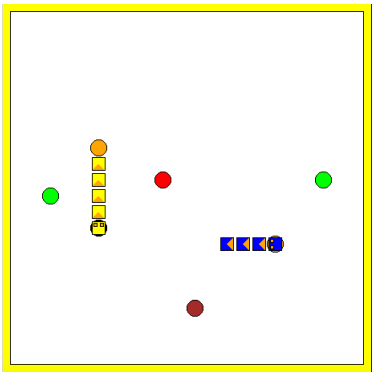

FIGURE 1 – Ici, le serpeut bleu va continuer en ignorant le `teleport` sur lequel il est puisque le serpent jaune lui envoie un effet `Shrink`

Nous sommes parvenus à fixer cette erreur (comme montré dans le scénario de démonstration lorsque le serpent bleu mange le `Revert` brun) grâce à un appel à une autre fonction qui contient une structure if supplémentaire. Ce bug peut parvenir à d'autres moments (nous en parlerons dans nos limites mais il est bon de savoir qu'il peut facilement être solutionné de la sorte).

8 Limitations du jeu

Dans l'implémentation de notre jeu, il n'est pas possible pour un serpent de recevoir plus d'un effet à la fois. Cela ne devrait pas poser problème puisque tous nos effets et toutes nos extensions s'appliquent directement sur le serpent afin qu'au coup suivant, l'effet du serpent soit à nouveau soit `nil` si le serpent ne subit pas d'autre effet, soit n'importe quel autre effet. Un lèger bug peut toutefois survenir dans le cas où un serpent prend un bonus et qu'un autre serpent lui envoie un autre bonus au même moment. Dans ce cas, seul un des deux bonus sera appliqué au serpent. Le bonus appliqué sera celui qui figure le plus haut dans la liste suivante :

1. Grow
2. Revert
3. Shrink
4. Teleport

Puisque toutes les cases situées en `x=1`, `x=22`, `y=1` et `y=22` se comportent comme des cases teleport, il n'est normalement pas possible qu'un serpent pénétrant sur une de ces cases puisse recevoir deux effets. Toutefois, cela peut arriver dans le cas où un serpent qui s'apprète à "traverser" un mur reçoit un bonus qui lui est envoyé par un adversaire. Si le bonus qui lui est envoyé est un `Grow` ou un `Shrink`, alors le serpent ne traversera pas le mur et mourra en

sortant du plateau de jeu. Ces erreurs peuvent être facilement corrigées de la même manière que nous avons corrigé ce problème avec le `Revert`. Ces erreurs étant suffisamment peu signifiantes, nous avons décidé de ne pas toutes les fixer de manière à ne pas surcharger (inutilement) notre code.
En dernier lieu, notre implémentation se faisant indépendamment des bombes, nous n'avons pas jugé utile de les inclure dans notre scénario de démonstration ce qui a pour malheureux effet de dénaturer le nom de notre projet. Cela dit, rien n'empêche qui que ce soit de créer un nouveau scénario dans lequel les bombes sont présentes pour plus de jeu.

9 Conclusion

En n'utilisant que des constructions déclaratives, nous sommes parvenus à réaliser la mission. À savoir implémenter correctement les fonctions `Next` et `DeclareStrategy` afin que les serpents répondent correctement aux instructions envoyées. Nous avons également intégré deux extensions à la version de base (`Shrink` et l'absence de murs). Seules quelques petites exceptions mineures ne sont pas toutes traitées par notre fonction `Next` mais dans la très grande majorité des scénarios, cela ne nuit pas à l'expérience de jeu.