ACT - Algorithmes et Complexité

Ce dépôt correspond au TP2 de ACT "*TP n ◦ 2 : Programmation dynamique*".

Jayjaywantee KOODUN et Thi-Ngoc-Anh TRAN

M1S1 2018/2019

Sommaire :

I. Contenu

Ce TP contient les fichiers suivants :

- `HexapawnV1.hs`
- `HexapawnV2.hs`
- `HexapawnV3.hs`
- `Makefile`
- `README.md`
- `Setup.hs`

II. Introduction

Nous proposons une résolution du problème de Hexapawn avec le langage Haskell.

Nous avons commencé par définir le jeu comme suit :

- le joueur par `Player` qui est `Black` ou `White`
- un pion par `Piece` qui est `B` ou `W`
- la carte du jeu par `Board`, `Board` contient des cases `Square` qui consiste chacun d'une Position `Pos` et du pion qui se trouve dessus
- une configuration par `GameState` qui représente l'état du jeu avec le `Board` et le `Player` dont c'est le tour

Nous avons proposé une première version naive dans le fichier `HexapawnV1.hs` avec la fonction `evaluateGameState`. Cette version naive ne passe pas les tests 6 et 7 de la plateforme. Ce fichier contient également une version mémoizée avec la fonction `evaluateGameStateMemo` qui ne passe pas les tests 6 et 7 sur la plateforme non plus.

Le fichier `HexapawnV2.hs` contient une version de mémoization avec une liste pour sauvegarder la valeur d'une configuration qu'on a déjà rencontré. Ce version ne passe pas les tests 6 et 7 sur la plateforme non plus.

Le fichier `HexapawnV3.hs` contient une version avec mémoization qui passent tous les tests sur la plateforme, et cette version est implémentée avec la structure `Map` du langage Haskell pour mémoriser les configurations.

La structure `Map` de Haskell fonctionne comme une table de hachage mais son implémentation a été fait par un arbre binaire AVL ainsi les méthodes pour inserer (`insert`) et récuperer (`lookup`) des valeurs dans le Map qu'on utilise ont une compléxité en *O(log n)*.

III. Exécution

1. Pour compiler et tester la version naive `HexapawnV1.hs` :

```
$ make hexv1

$ ./Main < Tests/Config3x4_1
$ ./Main < Tests/Config3x4_minus2
$ ./Main < Tests/Config4x4_0
$ ./Main < Tests/Config4x4_11
$ ./Main < Tests/Config5x5_3
$ ./Main < Tests/Config5x5_minus2
```

2. Pour compiler et tester la version avec memoization `HexapawnV2.hs` :

```
$ make hexv2

$ ./HexapawnV2 < Tests/Config3x4_1
$ ./HexapawnV2 < Tests/Config3x4_minus2
$ ./HexapawnV2 < Tests/Config4x4_0
$ ./HexapawnV2 < Tests/Config4x4_11
$ ./HexapawnV2 < Tests/Config5x5_3
$ ./HexapawnV2 < Tests/Config5x5_minus2
```

3. Pour compiler et tester la version avec memoization avec consommation de moins de mémoire `HexapawnV3.hs`:

```
$ make hexv3

$ ./HexapawnV3 < Tests/Config3x4_1
$ ./HexapawnV3 < Tests/Config3x4_minus2
$ ./HexapawnV3 < Tests/Config4x4_0
$ ./HexapawnV3 < Tests/Config4x4_11
$ ./HexapawnV3 < Tests/Config5x5_3
```

```
$ ./HexapawnV3 < Tests/Config5x5_minus2
```

IV. Compte rendu

Question 1 :

La fonction d'évaluation pour une configuration :

- Si tous ses successeurs ont des valeurs positives, sa valeur est *l'opposé du (maximum des valeurs positives + 1)*.
- Sinon, sa valeur est *l'opposé du (maximum des valeurs négatives - 1)*.

Question 2 :

Algorithme des versions naive (Version 1) et mémoization (Versions 2 et 3)

2.1. Algorithme version naive dans le fichier `HexapawnV1.hs`:

Entrée : hauteur et largeur de la carte du jeu, et la configuration du jeu (configuration étant définie par un board contenant les pions ainsi que le joueur dont c'est le tour)

1. Générez l’arbre de toutes les configurations possibles du jeu jusqu'au niveau où le jeu est terminé. Le niveau supérieur (le premier tour par le joueur blanc) s’appelle MAX (comme c’est maintenant au tour d'un pion blanc de se déplacer), le niveau suivant s’appelle MIN (celui ou c'est le tour du joueur noir), le niveau suivant devient MAX, et ainsi de suite.

2. Appliquez la fonction d'évaluation à tous les configurations terminaux (feuilles) pour obtenir des valeurs de qualité.

3. Utilisez ces valeurs pour déterminer les valeurs à attribuer aux parents immédiats avec la fonction d'évaluation.

4. Continuez à propager les valeurs en remontant les niveaux comme à l'étape 3

5. Lorsque les valeurs atteignent le sommet de l’arbre de jeu, MAX (le joueur blanc) récupère la valeur donnée par la fonction d'évaluation.

Sortie : la valeur de configuration de cet état du jeu.

2.2. Algorithme version mémoization dans le fichier `HexapawnV2.hs` :

Entrée : hauteur et largeur de la carte du jeu, et la configuration du jeu (configuration étant définie par la carte qui contient les pions ainsi que le joueur dont c'est le tour)

1. Selon le joueur (Noir ou Blanc), prendre tous ses pions, établir tous les moves possibles avec ces pions :
    - Manger le pion adversaire à son gauche diagonale, s'il y a 1 pion adversaire à cette position.
    - Manger le pion adversaire à son droite diagonale, s'il y a 1 pion adversaire à cette position.
    - Avancer un pas en avance, s'il n'y a pas aucun pion à cette position.
2. Calculer la valeur de configuration de chaque move et ajouter sa config dans la liste des configurations de cet état.
    - Si son adversaire a déjà gagné (atteindre à la dernière ligne) ou il ne peut pas bouger, il a perdu, donc sa config=0.
    - S'il atteint à l'avant dernière ligne, et il peut avancer ou manger son adversaire en diagonale, il a de chance de gagner dans 1 coup, donc sa config=1.
    - Sinon, continuer à calculer la configuration de nouvel état (selon étape 1), avec l'autre joueur, parce que maintenant c'est le tour de l'adversaire. Continuer jusqu'à quand un des joueurs perd, pour sa config=0.
3. Après avoir calculé les valeurs de configurations de tous les moves possibles, calculer la valeur de config de cet état du jeu, à partir de la liste des configurations, qui a été faite à la 2ème étape. Retourne le résultat.

Sortie : la valeur de configuration de cet état du jeu.

2.3. Algorithme version mémoization dans le fichier `HexapawnV3.hs` :

Entrée : hauteur et largeur de la carte du jeu, la configuration du jeu (configuration étant définie par la carte du jeu ainsi que le joueur dont c'est le tour), et un Map (pour la mémoization) qui contient l'état du jeu (GameState) comme clé et sa valeur étant la valeur de la configuration.

1. Selon le joueur (Noir ou Blanc) :

    - Si cet état est trouvé dans le map donnée, retourne la valeur de sa configuration.
    - Sinon, prendre tous ses pions, établir tous les moves possibles avec ces pions :
        - Manger le pion adversaire à son gauche diagonale, s'il y a 1 pion adversaire à cette position.
        - Manger le pion adversaire à son droite diagonale, s'il y a 1 pion adversaire à cette position.
        - Avancer un pas en avance, s'il n'y a pas aucun pion à cette position.

2. Calculer la valeur de la configuration de chaque move et ajouter sa config dans le map qui contient cet état et sa configuration :

    - Si son adversaire a déjà gagné (atteindre à la dernière ligne) ou il ne peut pas bouger, il a perdu, donc sa config=0.
    - S'il atteint à l'avant dernière ligne, et il peut avancer ou manger son adversaire en diagonale, il a de chance de gagner dans 1 coup, donc sa config=1.
    - Sinon :
        - Si cet état est trouvé dans le map, retourne sa configuration.
        - Sinon continuer à calculer la configuration de nouvel état (selon étape 1), avec l'autre joueur, parce que maintenant c'est le tour de l'adversaire. Continuer jusqu'à quand un des joueurs perd, pour sa config=0.

3. Après avoir calculé les valeurs de configurations de tous les moves possibles, chercher la valeur de configuration de cet état du jeur dans le map, qui a été faite à la 2ème étape. Retourne le résultat.

Sortie : la valeur de configuration de cet état du jeu.

Question 3 :

V. Résultat des tests :

1. Pour les tests des différentes versions, on utilise les tests sur :

    - La plateforme http://contest.fil.univ-lille1.fr/contest/classic/show/tp2-master1-hexapawn, copier et coller totalement le code depuis un fichier (HexapawnV1.hs par exemple) dans la zone d'écriture du code de la plateforme en prenant soin de bien choisir Haskell comme environnement.

    - Un test local sur le compilateur de Haskell `ghci` avec une board t9 = `[" ","pp pp",`

`"PPp "," P ", " P"]`. Le test local affiche le temps d'éxécution et l'espace de mémoire qui a été utilisé pour calculer la valeur de configuration de cette board.

Pour éxécuter le test local sur les ordinateurs en salle TP du M5 :

Lancer `ghci` sur le terminal de commandes avec la commande `ghci` suivante:

```
$ ls
Tests       HexapawnV1.hs   HexapawnV2.hs   HexapawnV3.hs   HexapawnV4.hs
Makefile    rapport.pdf     README.md       Setup.hs

$ ghci
```

Cette commande ouvre le compilateur de Haskell qui nous permet d'exécuter les différents fichiers de code avec la commande `:l <Nom du fichier>`. On configure d'abord l'environnement pour nous donner le temps de compilation ou d'exécution avec la commande `:set +s`. Après avoir lancé un fichier, on peut tester avec la commande `test9`.

```
Prelude> :set +s

Prelude> :l HexapawnV3.hs
[1 of 1] Compiling Main             ( HexapawnV3.hs, interpreted )
Ok, modules loaded: Main.
(0.20 secs, 46,827,824 bytes)

*Main> test9
3
(0.59 secs, 15,514,316 bytes)
```

2. Le fichier `HexapawnV1.hs` contienne la version naive qui génère toutes les configurations possibles du jeu et applique la fonction d'évaluation sur ces configurations pour trouver la valeur de la configuration initiale. Avec cette version, les tests 6 et 7 sur la plateforme ne passent pas. Dans le test local, elle prend ~18s pour finir le calcul, en plus de prendre beaucoup d'espace mémoire avec plus de 820 millions d'octets.

```
-------- Test sur plateforme ----------
Test 1  Succès (0.004s)
Test 2  Succès (0.004s)
Test 3  Succès (0.068s)
Test 4  Succès (1.092s)
Test 5  Succès (0.216s)
Test 6  Echec TIMED_OUT (Time limit exceeded (wall clock))
Test 7  Echec TIMED_OUT (Time limit exceeded (wall clock))
Test 8  Succès (0.012s)

----------- Test local ---------------
*Main> test9
3
```

```
(17.91 secs, 820,261,068 bytes)
```

3. Le fichier `HexapawnV2.hs` contienne une version avec mémoization qui consomme beaucoup d'espace mémoire. Nous avons utilisé une liste pour enregistrer la valeur des configurations de chaque état de jeu. Avec cette version, les tests 2, 4, 5 et 8 exécutent plus vite que la version 1, mais les test 6 et 7 n'ont toujours pas suffisamment de temps pour finir le calcul. On voit sur le test local, l'espace mémoire que la fonction occupe est 1/10 de la version 1. Cette version permet de calculer les configurations à la fin du calcul, selon une liste des configurations de chaque move des pions.

```
-------- Test sur plateforme ----------
Test 1  Succès (0.004s)
Test 2  Succès (0s)
Test 3  Succès (0.012s)
Test 4  Succès (0.056s)
Test 5  Succès (0.016s)
Test 6  Echec TIMED_OUT (Time limit exceeded (wall clock))
Test 7  Echec TIMED_OUT (Time limit exceeded (wall clock))
Test 8  Succès (0.008s)

----------- Test local ---------------
*Main> test9
3
(1.52 secs, 79,268,956 bytes)
```

4. Le fichier `HexapawnV3.hs` contienne une version avec mémoization qui consomme moins d'espace mémoire. Parce que l'on a utilisé un Map pour sauvegarder la configuration et sa valeur dès qu'on l'a calculé. La compléxité pour récuperer la valeur d'un configuration dans le jeu est en O(log n) selon l'implémentation de Map en Haskell dans notre cas. Cela permet d'éviter la recalculation des valeurs pour une configuration qu'on a déjà rencontrée. Tous les tests sur le plateforme sont passés avec un temps total de 7.484s.

Avec le test local, cette version prend moins de temps et consomme d'espace par rapport à la version 2. Elle prend moins d'espace de mémoire parce que les états du board sont calculés et enregistrés dans le map, qui permet de ne pas doubler de calcul(avec fonction Map.insert, O(logn)) et plus facile à trouver (avec fonction Map.lookup O(logn)).

```
```
-------- Test sur plateforme ----------
Test 1 Succès (0.004s)
Test 2 Succès (0.004s)
Test 3 Succès (0.016s)
Test 4 Succès (0.056s)
Test 5 Succès (0.016s)
Test 6 Succès (4.8s)
Test 7 Succès (2.596s)
```
```

```
Test 8  Succès (0.012s)
J'ai passé 8 tests sur 8. J'ai résolu l'exercice en Haskell en 7.484s

----------- Test local ---------------
**Main> test9
3
(0.62 secs, 15,510,808 bytes)
```