Projet 2015 : Qui Oz-ce?

FSAB1402

Marie-Marie van der Beek (5389-1300)

Pablo Gonzalez Alvarez (5243-1300)

29 novembre 2015

Introduction

1 Structure et conception

Dans ce rapport, nous allons vous présenter le fruit de notre travail. Il nous a été demandé d'écrire deux fonctions.

Nous allons d'abord vous décrire en deux temps la structure et les décisions prises pour les fonctions `BuildDecisionTree` et `GameDriver`.

Blabla intro...

Les spécifications de chaque fonction et sous-fonction se trouvent dans le fichier source du programme ci-joint. Comme demandé dans l'énoncé, le code que nous avons écrit utilise de manière exclusive le paradigme fonctionnel. En revanche nous n'avons pas implémenté de gestionnaire d'exceptions : cela simplifie l'implémentation de nos différentes fonctions.

1.1 Fonction `BuildDecisionTree`

Le coeur de la fonction `BuildDecisionTree` est la sous-fonction `Add`. Celle-ci construit petit à petit un arbre de décision presque optimal selon l'algorithme proposé dans l'énoncé.

Les sous-fonctions de `BuildDecisionTree` sont les suivantes :

— `Add` :
— `Names` :
— `Choice` :
— `Count` :
— `CountBoolean` :
— `Remove` :

Les paramètres de `Add` sont les suivants :

— `List` :
— `DB` :

1.2 Fonction `GameDriver`

Le coeur de la fonction `GameDriver` est la sous-fonction `GameDriverOops`. Celle-ci parcoure un arbre de décision et produit une liste de personnages en fonction des réponses

du joueur. Son corps est une correspondance de forme à la fois sur l'arbre de décision et sur les réponses données par le joueur. Les paramètres de `GameDriverOops` sont les suivants :

— `Tree` : un arbre de décision à parcourir ;
— `PreviousTree` : une liste des arbres de décision précédents, par défaut la liste est vide.

2 Extensions

2.1 Incertitude du joueur

Cette extension permets au joueur de passer une question en cas de doute. Nous avons modifier notre fonction BuildDecisionTree afin que l'arbre retournée aie trois branches (true, false et unknown) et non deux. Elle cree un arbre du type :
DecisionTree := leaf(List<Atom>) | question(<Atom> true :<DecisionTree> false :<DecisionTree> unknown :<DecisionTree>). Nous avons modifier notre fonction Add afin qu'elle crée la troisième branche en enlevant la question choisie de la liste des question, mais n'élimine personne de la liste des personnages possibles.
Ensuite, lorsque Gamedriver pose la question au joueur et celui-ci affirme qu'il ne connais pas la réponse, Gamedriver suivra le chemin de la branche unknown.

2.2 Bouton "oups"

Cette extension permet au jouer de retourner à la question précédente lorsqu'il se rend compte d'avoir commis une erreur. A cet effet, nous avons modifier notre Gamedriver afin qu'il fasse appel a une fonction GamedriverOops qui prend en paramètre un accumulateur, contenant la liste des arbres précédents. Quand l'utilisateur employe le bouton oups de l'interface, nous appliquons gamedriverOops à l'arbre précédent, et dans ce cas, on enlève cet arbre de la liste dans l'accumulateur. Cependant, il étais nécessaire de faire un choix sur l'action qui aurais lieu si le joueur retourne au début de l'arbre et qu'il ny aie pas d'arbres précédents. Nous avons décidé de faire une boucle infinie, qui repète la première question jusqu'a ce qu'il réponde a celle-ci. Ceci oblige le joueur a descendre dans l'arbre.

3 Limitations

Notre arbre de décision renvoit toujours au moins une réponse, mais si le nombre de questions de la base de données n'est pas suffisant, on obtiendra pas toujours une réponse possible (ce qui serait idéal), mais bien une liste de réponses possibles (moins idéal). C'est un choix que nous avons fait.

Pas de gestion des erreurs.

4 Complexité

Nous allons développer dans cette section, la complexité de nos deux programmes principaux `BuildDecisionTree` et `GameDriver`, dans le cas de base sans extensions.

4.1 `BuilDecisionTree`

Nous devons afin d'obtenir la complexité de `BuilDecisionTree`, analyser les six sous-fonctions auquel elle fait appel :

- `Count`
- `CountBoolean`
- `Choice`
- `Remove`
- `Names`
- `Add`

4.1.1 `Count`

Cette fonction parcoure la database et renvoie un tuple contenant respectivement une question, une liste des personnes ayant repondu true et une liste des personnes ayant repondu false à cette question. La complexité liée a cet algoritme est relativement simple car elle parcoure une fois la database. Sa complexité sera donc toujours O(m) où m est le nombre de personnes dans la database.

4.1.2 `CountBoolean`

Cette fonction applique la fonction prédéfinie Length à une des listes de la fonction Count selon que le boolean passée en paramètre soit true ou false. Avec l'hypothèse que la complexité de Length soit O(1), nous pouvons déduire que la complexité de Countboolean est identique a celle de Count, O(m).

4.1.3 `Choice`

Choice est une fonction qui a chaque exécution appelle deux fois la fonction Count-Boolean pour définir la différence en valeur absolue entre le nombre de réponse true et false à une question. Elle fera ceci pour chaque question dans la liste. Un fois la liste vide, elle appelle la fonction Count avec la question optimale. Nous avons donc une complexité 2O(m)*O(n) + O(m) or nous tenons en compte que des termes dominants, ce qui nous donne une complexité O(2m*n).

4.1.4 `Remove`

Remove enlève un élément de la liste des questions passée en paramètre. Pour ce faire, elle la parcoure jusqu'à ce qu'elle trouve l'élément et l'enlève ensuite. Dans le meilleur cas, la question se trouve au début de liste, O(1) Dans le pire cas, la question se trouve en fin de liste, $\theta(n)$. On suppose qu'il y ait une distribution uniforme de l'emplacement des questions, la complexité sera O(n).

4.1.5 `Names`

Cette fonction parcoure la database et en extrait les noms des personnages. La complexité est relativement simple car elle n'appele pas d'autres fonctions. Elle fera m-1 récurrences et est donc de complexité O(m)

4.1.6 `Add`

Add fait appel a la fonction Choice afin de choisir une question optimale. Ensuite, elle vérifie la condition de fin en appliquant deux fois Countboolean et finit par faire deux récurrence. La dernière exécution se fera lorsqu'une des conditions de fin est remplie, ce qui appelle la fonction Names. Nous avons pour chaque exécution, une complexité de O(2m*n)+O(2m) excepté la dernière qui aura une complexité de O(m). En combinant ceci avec le nombre d'occurrences, nous pouvons écrire O(2m*n+2m)*O(2^m)+O(m) que nous réduisons à O(2m*n)*O(2^m).

Conclusion Notre `BuildDecisionTree` fait appel à Add une seule fois, nous pouvons donc conclure qu'elles auront la même comlexité O(2m*n)*O(2^m) avec m le nombre de personnages dans la database et n le nombre de questions posées à ceux-ci.

4.2 `GameDriver`

GameDriver parcoure l'arbre de décision, pose une question au joueur au moyen de la fonction Projectlib.askQuestion et selon la réponse donnée rappelle GameDriver avec le sous-arbre true ou false. Ensuite, une fois une liste de personnages trouvée, il renvoie celle-ci au jouer avec ProjectLib.found. Nous savons que les deux fonctions liées à l'interface auquel nous faisons appel, Projectlib.askQuestion et ProjectLib.found sont de complexité O(1), nous pouvons donc les négliger. La complexité dépendra également de la hauteur de l'arbre mais nous pouvons la borner par O(n) où n est le nombre de questions posées au personnages.

Conclusion