

Projet 2015 : Qui Oz-ce?

FSAB1402

Marie-Marie VAN DER BEEK
(5389-1300)

Pablo GONZALEZ ALVAREZ
(5243-1300)

30 novembre 2015

Introduction

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

Nous allons d'abord vous décrire en deux temps la structure et les décisions prises pour les fonctions `BuildDecisionTree` et `GameDriver`. De plus, nous listerons les extensions que nous avons implémentées. Ensuite, nous discuterons des limitations de notre programme. Enfin, nous terminerons sur la complexité de notre code.

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.

1 Structure et conception

Lors de la conception, nous avons veillé à ce que nos fonctions soient récursives terminales pour éviter une utilisation inutilement intensive de la mémoire de l'ordinateur.

1.1 Fonction `BuildDecisionTree`

Le cœur 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 en faisant appel à d'autres sous-fonctions.

Les sous-fonctions de `BuildDecisionTree`, selon l'ordre d'apparition dans le code, sont les suivantes :

- `fun{Count Quest DB}` : fait partie de l'algorithme de décision des branches de l'arbre. Par induction sur la base de données, elle analyse pour chaque personnage la réponse à la question et la rajoute dans une des deux bases de données (`DBTrue` ou `DBFalse`) en fonction de celle-ci. Une fois arrivé à la fin de la base de données, elle renvoie cela sous la forme d'un tuple : `triple(Question DBTrue DBFalse)` ;
- `fun{CountBoolean Quest DB Bool}` : selon le booléen `Bool` passé en paramètre, renvoie la longueur de la liste de personnages de `DBTrue` et de `DBFalse` du tuple renvoyé par `Count` ;

- `fun{Choice Quest DB}` : est la base de l'algorithme de décision des branches de l'arbre. Elle est initialisée avec la première question et la différence entre le nombre de **true** ou de **false** de celle-ci. Ensuite, elle parcourt par occurrence de forme la liste des questions : si elle trouve une question dont la différence est plus petite que la question optimale actuelle, alors elle remplace celle-ci dans l'accumulateur **Quest**. Une fois que la liste est parcourue, retourne la question optimale ;
 - `fun{Remove List Quest}` : parcourt la liste des question par récurrence : si elle trouve la question **Quest**, l'enlève. Sinon elle continue jusqu'à ce que la liste soit parcourue ;
 - `fun{Names DB}` : parcourt la base de données par occurrence de forme et rajoute le nom de chaque personnage dans une liste. Une fois que la base de données est parcourue, cette liste est retournée ;
 - `fun{Add List DB}` : construit l'arbre de décision par récursion selon un certain nombre de condition. Elle fait appel à la plupart des fonctions ci-dessus. L'algorithme est le suivant :
 1. Si la liste des questions est vide, renvoie une feuille `leaf(<atom>)` qui contient la liste des personnages ;
 2. Si il ne reste plus qu'un personnage, renvoie une feuille `leaf(<atom>)` avec ce personnage ;
 3. Si plus aucune question n'est discriminatoire, renvoie une feuille `leaf(<atom>)` contenant la liste des personnages de la question précédente ;
 4. Si aucune des conditions précédentes ne sont vérifiées, il choisit une question via la fonction **Choice** et renvoie un arbre de décision de la forme suivante :
`question(<Quest> true:<DecisionTree> false:<DecisionTree>)`.
- Il répète les 4 pas précédents jusqu'à obtenir un arbre de décision optimal qu'il renvoie.

1.2 Fonction GameDriver

`fun{GameDriver Tree}` parcourt un arbre de décision **Tree** et produit une liste de personnages en fonction des réponses du joueur. Son corps applique une correspondance de forme à la fois sur l'arbre de décision et sur les réponses données par le joueur.

L'algorithme est le suivant :

1. Au moyen d'une interface graphique, la question se trouvant dans le noeud actuel de l'arbre est posée à l'utilisateur ;
2. si la réponse du joueur est **true**, on continue à parcourir l'arbre de décision dans sa branche droite **true T** ;
3. si la réponse du joueur est **false**, on continue à parcourir l'arbre de décision dans sa branche gauche **false F** ;
4. Tant qu'on arrive pas a une feuille `leaf(Ans)` de l'arbre on répète les trois pas précédents, ce qui nous mène à la réponse finale, renvoyée à l'utilisateur.

2 Extensions

2.1 Incertitude du joueur

Cette extension permet au joueur de passer une question en cas de doute. Nous avons modifié une sous-fonction de notre fonction `BuildDecisionTree` afin que l'arbre retourné ait trois branches (**true**, **false** et **unknown**) et non deux comme dans le cas de base. Elle crée un arbre dont la nouvelle structure est :

```
DecisionTree := leaf(List<Atom>) | question(<Atom>
true:<DecisionTree> false:<DecisionTree> unknown:<DecisionTree>)
```

Cette sous-fonction rajoute une branche supplémentaire **unknown** qui contient l'entièreté des personnages contrairement aux branches **true** et **false** qui prend les personnages correspondants à ces booléens.

Ensuite, lorsque `GameDriver` pose la question au joueur et celui-ci affirme qu'il ne connaît pas la réponse, la fonction poursuivra son chemin dans la branche **unknown**.

2.2 Bouton « oups »

Cette extension permet au joueur de retourner à la question précédente lorsqu'il se rend compte qu'il a commis une erreur. À cet effet, nous avons modifié notre `GameDriver` afin qu'il fasse appel à une fonction auxiliaire `GameDriverOps` qui prend en paramètre un accumulateur, contenant la liste des arbres précédents. Quand l'utilisateur emploie le bouton « oups » de l'interface, nous appliquons `GameDriverOps` à l'arbre précédent, et dans ce cas, on enlève cet arbre de la liste dans l'accumulateur.

Cependant, il était nécessaire de faire un choix sur l'action qui aurait lieu si le joueur retourne au début de l'arbre et qu'il n'y a donc plus d'arbres précédents. Nous avons décidé de faire une boucle infinie, qui répète la première question jusqu'à ce que le joueur réponde à celle-ci. Ceci l'oblige à poursuivre son chemin dans l'arbre.

3 Limitations

Malgré nos efforts, notre code contient un certains nombre de limitations et de problèmes.

Tout d'abord, si le nombre de questions de la base de données n'est pas suffisant, on obtiendra pas toujours une seule réponse possible (cas idéal), mais bien une liste de réponses possibles (cas moins idéal). C'est une contrainte de l'énoncé : le programme renvoie toujours au moins un panel de personnes en fonction des réponses du joueur. On aurait pu rajouter une fonction aléatoire qui pioche un personnage parmi ce panel, mais cela aurait parfois rendu les réponses fausses.

Enfin, notre programme ne gère pas les erreurs (e.g. la base de données n'est pas sous le bon format, l'encodage n'est pas le bon, ou encore la base de données n'est plus accessible). Sans exceptions, l'implémentation de nos différentes fonctions est simplifiée et plus claire mais moins robuste.

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 BuildDecisionTree

Afin d'obtenir la complexité de `BuildDecisionTree`, nous devons analyser les six sous-fonctions auxquelles elle fait appel :

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

Soient m et n le nombre de personnages et de questions respectivement.

4.1.1 Count

Cette fonction parcourt la base de données et renvoie un tuple contenant respectivement une question, une liste des personnes ayant répondu **true** et une liste des personnes ayant répondu **false** à cette question. La complexité liée à cet algorithme est relativement simple car elle parcourt une fois la base de données. Sa complexité sera donc toujours $O(m)$.

4.1.2 CountBoolean

Cette fonction applique la fonction prédéfinie `Length` à une des listes de la fonction `Count` selon que le booléen passé en paramètre soit **true** ou **false**.

Avec l'hypothèse que la complexité de `Length` est $O(1)$, nous pouvons en déduire que la complexité de `CountBoolean` est identique à celle de `Count`, c'est-à-dire $O(m)$.

4.1.3 Choice

`Choice` est une fonction qui à chaque exécution appelle deux fois la fonction `CountBoolean` 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 $2 \cdot O(m) \cdot O(n) + O(m)$ or nous ne prenons en compte que les termes dominants, ce qui nous donne une complexité finale de $O(2 \cdot m \cdot n)$.

4.1.4 Remove

`Remove` enlève un élément de la liste des questions passée en paramètre. Pour ce faire, elle la parcourt jusqu'à ce qu'elle trouve l'élément et l'enlève ensuite.

Dans le meilleur des cas, la question se trouve au début de liste, c'est-à-dire $O(1)$. Dans le pire des cas, la question se trouve en fin de liste, c'est-à-dire $O(n)$. On suppose

qu'il y a une distribution uniforme de l'emplacement des questions, la complexité sera donc de $O(n)$.

4.1.5 Names

Cette fonction parcourt la base de données et en extrait le nom des personnages. La complexité est relativement simple car elle n'appelle pas d'autres fonctions. Elle fera m récurrences et est donc de complexité $O(m)$.

4.1.6 Add

Add fait appel à 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 récurrences. La dernière exécution se fera lorsqu'une des conditions de fin est remplie, ce qui appelle la fonction **Names**. Nous pouvons donc borner la complexité à $O(2m \cdot n \cdot 2^m)$.

Conclusion

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

4.2 GameDriver

GameDriver parcourt 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 joueur avec **ProjectLib.found**.

Nous savons que les deux fonctions liées à l'interface auxquelles 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

Ce projet nous a permis de mieux comprendre la programmation déclarative et plus particulièrement de mieux appréhender le paradigme fonctionnel avec la correspondance de forme, la récursion terminale ainsi que le principe des vases communicants.

Pour bien terminer, comme le dit si bien L. Frank Baum dans le Cycle d'Oz 3 :

« - Ces routes sont curieuses... et il y en a tant ! Où croyez-vous qu'elles mènent ?

- Les routes ne vont nulle part, affirma l'Homme Hirsute. Elles restent où elles sont afin que les gens puissent les emprunter. »