Projet 2015 : Qui Oz-ce ?

LFSAB1402 – Informatique 2

Jérémie Melchior - Guillaume Maudoux - Hélène Verhaeghe - Florentin Rochet

Deadline : le 30 novembre 2015

Introduction

Etes-vous un garçon ou une fille ? Avez-vous les cheveux long ou court ? Voici tant de questions que l'on peut vous demander pour vous connaître. Il y a un jeu de société bien connu dans lequel vous devez poser des questions pour trouver quelqu'un : le Qui est-ce ? Vous connaissez surement ce jeu, mais saurez-vous le faire vous-même ? Vous allez en effet réaliser ce jeu durant le projet de cette année.

Consignes

Le projet se déroulera sur deux semaines, du lundi 16 novembre au lundi 30 novembre. Vous devez former des groupes de deux étudiants et soumettre vos travaux sur INGInious[1] pour le lundi 30 novembre à 23h00 (heure belge) au plus tard. Ne vous y prenez pas au dernier moment car le serveur sera vite surchargé dans les dernières heures avec la limite.

Vous trouverez sur iCampus une base de données qui servira d'entrée à votre programme. Celle-ci contient des personnes et leurs caractéristiques. Votre programme devra déterminer quelles questions poser, et dans quel ordre, afin de trouver qui est la personne que le joueur a choisie préalablement.

Vous devez tous commencer par faire une solution de base que vous soumettrez dès que celles-ci répondra aux consignes. Ensuite, vous pourrez agrémenter votre solution en implémentant des extensions supplémentaires en bonus.

Votre projet doit être un fichier zip de la forme NOMA1-NOMA2.zip contenant deux fichiers. Un fichier **Code.oz** avec votre code et un fichier **Rapport.pdf** avec votre rapport.

Votre solution doit être entièrement déclarative. Le partage entre groupe est évidemment interdit et considéré comme du plagiat. Vous pouvez cependant discuter de vos stratégies pour réaliser votre solution mais sans échanger de code.

Dans le rapport, de **maximum 5 pages**, vous devez expliquer la structure de votre programme, les décisions de conception que vous avez prises et bien sûr la liste des extensions supplémentaires que vous avez implémentées. Mentionnez également les limitations et les problèmes connus de votre programme.

Votre rapport doit également comprendre une analyse de la complexité de vos deux fonctions majeures **BuildDecisionTree** et **GameDriver**, dans le cas où on reste dans la version de base sans extension supplémentaire (pour plus de simplicité). Pour pouvoir faire cela, vous devez connaître la complexité des fonctions que vous appelez. Nous vous précisons donc que **ProjectLib.askQuestion**, **ProjectLib.found**, **ProjectLib.surrender** peuvent être considérées comme étant $\Theta(1)$.

Une interview pourrait vous être demandée dans le cas où votre projet ne satisferait pas aux consignes données dans cet énoncé.

1. https://inginious.info.ucl.ac.be/course/FSAB1402/project

Objectif

Dans la solution de base, vous devez effectuer les étapes suivantes :

1. Lire la base de données ;
2. Construire un arbre de décision pour guider la tâche suivante ;
3. Poser les questions au joueur humain pour trouver la personne choisie.

Lire la base de données

Vous devez commencer par lire la base de données des personnes depuis un fichier. Puisque vous n'avez pas appris à manipuler les fichiers en Oz, nous vous fournissons le module qui se charge de lire un fichier et de vous le rendre sous une forme de liste de records. Pour cela, vous devez utiliser la fonction loadDatabase qui est définie dans le module **ProjectLib** (voir le code exemple suivant) :

ListOfPersons = {ProjectList.loadDatabase file "my-database.txt"}

Ou depuis une URL :

ListOfPersons = {ProjectLib.loadDatabase url "http://site.com/db/database.txt"}

Nous vous proposons une base de données contenant les Diables Rouges qui faisaient partie du jeu de société sorti à leur effigie pour la Coupe du Monde 2014. Comme vous pouvez le voir, celle-ci contient une liste de records, il s'agit donc de données standards de Oz. Le label du record est **person**. Le nom du joueur est accessible avec la feature 1. Les autres champs du record sont les questions avec les réponses. Les noms et questions sont des atomes

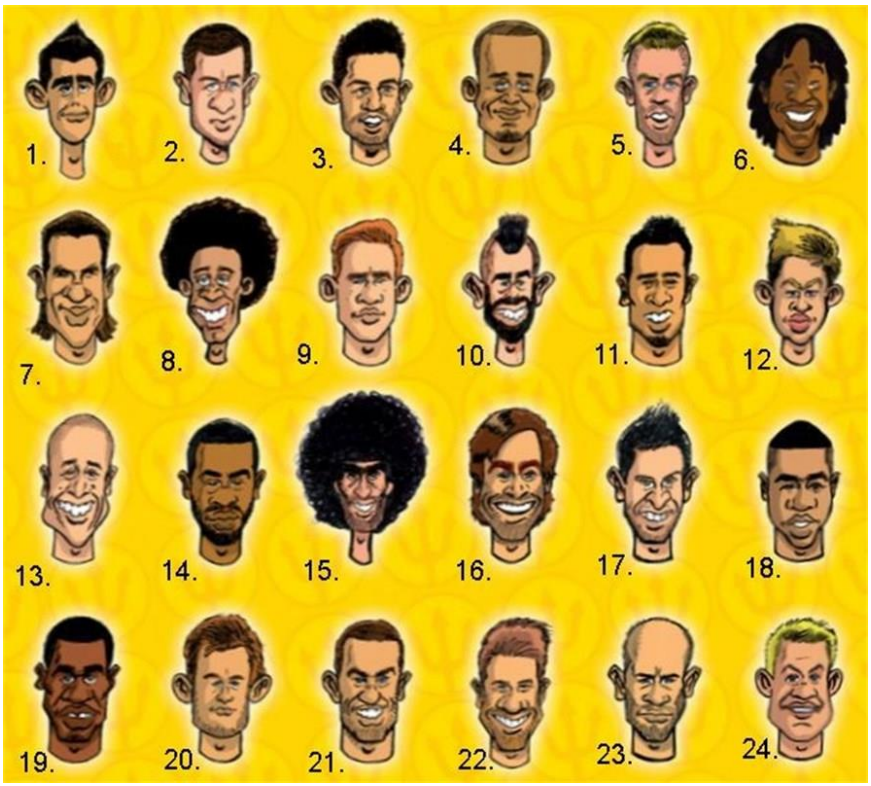

Figure 1: Image tirée du jeu "Qui Est-ce ? Edition Diables Rouges[2]

Pour la version de base, toutes les personnes ont le même ensemble de questions, et les réponses sont toujours **true** ou **false**.

2. http://www.gva.be/ahimgpath/assets_img_gvl/2014/02/07/2831162/probeer-zelf-het-wie-is-het-spel-van-de-rode-duivels-id5326122-1000x800-n.jpg

Construire un arbre de décision

Votre première tâche est d'écrire la fonction **BuildDecisionTree** qui prend en paramètre la base de données et renvoie un arbre de décision qui sera utilisé durant la deuxième tâche.

Cet arbre que nous appellerons **DecisionTree** peut, par exemple, suivre la structure suivante :

<DecisionTree> ::= leaf(<List[Atom]>) % une liste de nom de personnes
| question(<Atom> true :<DecisionTree> false :<DecisionTree>)

Dans les noeuds internes (la seconde forme), le premier champ est une question à poser au joueur. Sous la feature **true** (resp. **false**), on trouve un plus petit arbre de décision qui contient toutes les personnes qui répondraient **true** (resp. **false**) à la question.

Une fois cette arbre de décision créé, nous pouvons le parcourir de la racine jusqu'à atteindre une feuille en posant la question qui se trouve à chaque nœud. Si le joueur répond **true** (resp. **false**), on recommence avec le sous-arbre sous la feature **true** (resp. **false**).

Lorsqu'on arrive à une feuille (la première forme), c'est qu'une personne a été trouvée. Le nom de la personne trouvée est l'unique élément de la liste dans le tuple **leaf**. Puisque rien n'interdit à deux personnes différentes d'avoir toutes les mêmes réponses à toutes les questions, il se peut que la liste contienne plusieurs noms de personnes. Il faut alors toutes les proposer au joueur, car plus aucune question ne peut aider à les départager sans améliorer la base de données.

Voici une partie d'un exemple d'arbre de décision pour la base de données exemple :

```
Question('A-t-il des cheveux longs ?'
  true:question('A-t-il des cheveux noirs ?'
    true:question('Est-il blanc de peau ?'
      true :leaf(['Axel Witsel' 'Marouane Fellaini'])
      false :leaf(['Romelu Lukaku'])
    )
    false:question('A-t-il une barbe ?'
      true:question('Voit-on ses dents ?'
        true:leaf(['Nicolas Lombaerts'])
        false:leaf(['Guillaume Gillet'])
      false:leaf(['Daniel Van Buyten'])
   )
  )
  false:…)
```

D'autres arbres sont également valides pour la même base de données. Étant donné que le but (implicite) du jeu est de trouver la personne choisie *en posant un minimum de questions*, il est préférable de produire *un arbre qui ait une hauteur la plus petit possible*. Pour rappel, la hauteur d'un arbre est la distance entre sa racine et sa feuille la plus basse.

Un arbre de décision correct est dit *optimal* s'il n'existe pas d'autre arbre correct dont la hauteur soit strictement plus petite que sa hauteur. Notez qu'il peut exister plusieurs arbres optimaux, si tous ont la même hauteur.

Construire un arbre ayant la hauteur la plus petite.

Construire un arbre de décision optimal n'est pas une tâche facile. Nous n'allons pas exiger que vous le fassiez. Cependant, vous ne pouvez pas construire un arbre de décision stupide (prendre en premier une question qui ne discrimine qu'un seul personnage !).

Voici un petit algorithme qui permet de souvent construire des arbres de décisions proches de l'optimalité. Vous devez implémenter un algorithme au moins aussi bon.

Algorithme arbre presque optimal :

Pour chaque question, vous devez compter le nombre de personnes qui répondraient **true** et ceux qui répondraient **false**. Vous allez ensuite sélectionner une des questions pour laquelle les nombres de **true** et de **false** sont les plus proches. La racine de l'arbre utilisera alors cette question.

Pour construire le sous-arbre **true**, vous allez prendre le sous-ensemble des personnes qui répondent **true**, et retirer de l'ensemble des questions la question utilisée à la racine. Vous allez ensuite recommencer l'opération avec ces deux sous-ensembles. Vous ferez également de même pour le sous-arbre false. On dit de l'algorithme qu'il est *récursif*, cela signifie qu'il s'appelle lui-même jusqu'à trouver la bonne réponse.

Quand plus aucune question ne peut discriminer une personne dans le sous-ensemble, vous pouvez alors construire une feuille **leaf** avec la liste des personnes restantes. C'est le *cas de base* de cet algorithme récursif.

Démarrage du jeu et choix de la personne

Nous avons conçu une interface utilisateur graphique (*GUI* pour Graphical User Interface) qui va vous permettre de jouer de manière agréable. Vous pouvez la lancer via la procédure suivante :

Options = opts(builder:BuildDecisionTree persons:ListOfPersons driver:GameDriver)
{ProjectLib.play Options}

BuildDecisionTree est une fonction que vous devez définir. Elle prend la DB en argument et construit l'arbre de décision à partir de celle-ci.

GameDriver est une fonction que vous devez définir, elle reçoit en argument, par **ProjectLib**, l'arbre de décision créé par la fonction **BuildDecisionTree** et renvoie toujours unit. Elle va appeler des procédures de **ProjectLib** pour diriger le jeu.

Pour ce faire, vous avez besoin de deux routines :

- **{ProjectLib.askQuestion Question}** pose la question Question au joueur, et renvoie sa réponse. Par exemple, si je pense à *Romelu Lukaku*, **{ProjectLib.askQuestion 'A-t-il des cheveux longs ?'}** renverra **true**.
- **{ProjectLib.found ListOfNames}** doit être appelée lorsque votre programme a trouvé une feuille, c'est-à-dire lorsqu'il a deviné la personne. Cette fonction renvoie le nom de la personne s'il est dans la liste, ou **false** s'il n'y était pas (oups, cela veut dire que vous vous êtes trompé !).

Voici un exemple d'implémentation de **GameDriver** qui teste avec des IF imbriqués et représente l'arbre ci-dessus. Votre code ne doit pas supposer un arbre existant, mais bien s'adapter à n'importe quel arbre construit par votre première étape.

```
fun {GameDriver Tree}
  Result
in
  if {ProjectLib.askQuestion 'A-t-il des cheveux longs ?'} then
    if {ProjectLib.askQuestion 'A-t-il des cheveux noirs ?'} then
      if {ProjectLib.askQuestion 'Est-il blanc de peau ?'} then
        Result = {ProjectLib.found ['Axel Witsel' 'Marouane Fellaini']}
      else
        Result = {ProjectLib.found ['Romelu Lukaku']}
      end
    else
      if {ProjectLib.askQuestion 'A-t-il une barbe ?'} then
        if {ProjectLib.askQuestion 'Voit-on ses dents ?'} then
          Result = {ProjectLib.found ['Nicolas Lombaerts']}
        else
          Result = {ProjectLib.found ['Guillaume Gillet']}
        end
      else
        Result = {ProjectLib.found ['Daniel Van Buyten']}
      end
    end
  else
    …
  end
  if Result == false then
    {Browse 'Aucune personne ne correspond à cette description'}
  end

  % Toujours renvoyer unit
  unit
end
```

La procédure **ProjectLib.play** va d'abord appeler **BuildDecisionTree** pour créer l'arbre de décision. Ensuite elle appelle **GameDriver** autant de fois que le joueur désire jouer l'arbre de décision en paramètre. L'arbre de décision calculé par **BuildDecisionTree** va donc pouvoir être utilisé plusieurs fois !

Extensions

Nous avons parlé jusqu'ici de la base à implémenter. Maintenant, parlons un peu des extensions possibles. Chaque extension à une difficulté qui lui est associée. Vous devez choisir un sous-ensemble de ces extensions pour un minimum de 4 étoiles. Vous pouvez obtenir plus que 4 étoiles, un bonus pourrait être accordé à ceux qui ont obtenus plus d'étoiles. Mais attention qu'au-delà des 4 étoiles requises, cela ne vous aidera pas à sauver un mauvais projet. Il vaut donc mieux se concentrer sur la base qui est à soumettre avant de travailler sur les extensions.

Extension	Difficulté
Incertitude dans la base de données	*
Questions non binaires (pas seulement **true**/**false**)	* *
Incertitude du joueur (**true**, **false**, **I don't know**)	* * *
Gérer les erreurs du joueur	* * * *
Bouton « **oups** » (revenir à la question précédente)	* * *

L'implémentation de ces extensions peut mener à des modifications de la structure de l'arbre de décision.

Incertitude dans la base de données

Le record d'une personne peut ne pas contenir les mêmes champs que celui d'une autre personne. Dans le cas d'une question où le joueur répond **true**, il faudra donc également garder ceux pour qui vous ne connaissez pas la réponse à cette question.

Astuce : la solution idéale ne modifie que l'étape de construction de l'arbre.

Questions non binaires

Cette extension gère les cas où la réponse n'est pas **true** ou **false**. Par ex : « La couleur des cheveux est-elle : ». Les réponses possibles pouvant être : noir, brun, blond ou roux.

Attention : cette extension complique l'implémentation des autres extensions. Réfléchissez donc bien à la manière d'implémenter vos extensions avant de vous lancer sur celle-ci.

Incertitude du joueur

En plus de pouvoir choisir entre **true** et **false**, le joueur pourra maintenant répondre « je ne sais pas » (en anglais : I don't know). L'atome **unknown** sera utilisé pour représenter la non-réponse à une question de la part du joueur.

Lorsque cela se produit, vous ne pouvez éliminer aucune personne mais vous devez continuer en posant une nouvelle question pour continuer à éliminer des personnes. Pour autoriser cette réponse dans la GUI, vous devez ajouter le champ **allowUnknown:true** dans le record **Options** avant de le passer en paramètre **ProjectLib.play**.

Gérer les erreurs du joueur

Jusqu'ici nous avons supposé que le joueur était sûr de ses réponses et ne pouvait pas se tromper. Dans cette extension nous allons tenir compte que quand le joueur répond, il est possible qu'il ait fait une erreur en répondant **true** alors que la réponse aurait dû être **false**.

Mais comment savoir si c'est le cas ? Si un **{ProjectLib.found}** renvoie **false**, cela voudra dire que c'est le joueur qui s'est trompé et non votre algorithme qui est incorrect.

Vous n'allez pas vous arrêter à chaque question en pensant qu'il y a une erreur. Cependant vous devez pouvoir prendre en compte une possible erreur du joueur, et recommencer à partir de cette erreur.

Astuce : Vous pouvez considérer que le joueur n'a fait qu'une seule erreur au plus. Il suffit alors de parcourir à nouveau l'arbre de décision pour trouver les personnes qui ont au plus une différence par rapport aux réponses données. Reconstruisez alors un arbre de questions à poser pour seulement ces personnes. Si aucune personne n'a qu'une seule différence, on suppose alors que le joueur a fait deux erreurs et on parcourt à nouveau l'arbre pour trouver les personnes qui ont deux différences, et ainsi de suite.

Bouton « oups »

Dans l'extension précédente, nous supposons que le joueur a pu faire une erreur. Cependant, le joueur peut se rendre compte qu'il vient de faire une erreur.

Dans cette extension nous alors offrir la possibilité au joueur de corriger la dernière question dans le cas où il voudrait changer sa dernière réponse.

Pour afficher ce bouton dans la GUI, vous devez ajouter le champ **oopsButton:true** dans le record **Options** avant de le passer en paramètre à **ProjectLib.play**.