Ecole Polytechnique de Louvain

Informatique II

Rapport de projet

Antoine Gennart

Réalisé le 26 novembre 2015.

Dans le cadre de ce projet, il nous a été demandé de réaliser un jeu "Qui oz-ce" du même style que le célèbre jeu "Qui est-ce" . Notre jeu part d'une base de données sur les différents membre de l'équipe des Diables Rouges de la coupe du monde 2014, contenant des questions et des réponses permettant de les identifier.

Il nous a été demandé de réaliser avant tout une solution de base qui se diviserait en trois étapes.
La première, **lire la base de donnée**, nous a été fournie. Les deux étapes qu'il nous restait à implémenter étaient **construire un arbre de décision pour guider la tache suivante** et **poser les questions au joueur humain pour trouver la personne choisie**.

1 Construire un arbre de décision (BuildDecisionTree)

Tout abord, nous avons implémenté la construction d'un arbre basique, qui prendrait en compte les questions dans leur ordre d'apparition. Cela nous a permis de comprendre la structure fondamentale que devait prendre cette fonction, et donc cet arbre.

La fonction prend la **database** en argument et une sous fonction crée immédiatement la liste des questions à poser selon leur apparition dans la base de données. Dans le cas simple, la liste des questions est crée dans l'ordre de d'apparition de celles-ci. Pour chacune d'entre elles, la fonction auxiliaire parcourt toute la database et construit une **ListFalse**, contenant les noms des joueurs dont la réponse à la question actuelle est **false**, ainsi qu'une **ListTrue**, contenant les noms des joueurs dont la réponse à la question est **true**.
La première question posée constitue le noeud principal à partir duquel l'arbre de décision va être construit.
Cet arbre sera ensuite construit par **appels récursifs** de la fonction
{`BuidDecisionTreeAcc ..`} avec, à chaque appel, la question suivante. A **droite** de toute question (=noeud), la fonction sera appelée avec la **liste true** à la place de la **database**, alors qu'à **gauche** elle sera appelée avec la **liste false**.

Une fois que toutes les questions ont été posées, c'est à dire que la **"question" est nil**, la fonction **renvoie une liste** contenant tous les joueurs encore présents dans la **database actuellement prise en compte** par celle-ci, à savoir une liste false ou true.

Ensuite, nous avons implémenté différentes améliorations,...

Il nous a ensuite été demandé de faire en sorte que notre arbre soit le plus optimal possible. Nous sommes partis sur le principe que l'optimisation de l'arbre se ferait par la maintenance de son équilibre.
Nous avons donc implémenté une fonction {`BestQuestion`} qui prend en argument la base de données, la liste de question actuelle ainsi qu'une variable unbound, la nouvelle liste de

question, qui sera modifée par la fonction. {`BestQuestion`} renvoie la question qui engendrera la plus petite différence entre le nombre de personnes qui répondront true et celles qui répondront false, afin de maintenir l'arbre le plus équilibré possible.
Nous avons également du prendre en compte le nombre de personnes possédant une réponse à la question, non pas pour l'équilibre de l'arbre, mais pour minimiser le nombre de questions à poser pour trouver la bonne personne. En effet, l'extension "unknown" fait en sorte que les personne dont la réponse n'est pas connue se retrouvent dans la liste true et dans la liste false. Nous sommes arrivés au critère de précision suivant :

Abs((# true)-(# false)) - (# réponses)

Cela se fait en plusieurs étapes :

1. Calcul du critère de précision pour chaque question de la liste. La fonction {`DiffTrueFalseList DB ListOfQuestions`} parcourt la base de données pour chaque question présente dans la liste de question actuelle et renvoie une liste contenant la valeur critère de précision de chaque question, dans le même ordre que l'apparition des questions dans la liste.

2. Trouver l'emplacement du plus petit élément de la liste des différences, au moyen de {`MinList L`}

3. Extraire ce N^{ieme} élément de la liste de questions. C'est la question qui déséquilibrera le moins l'arbre, elle est renvoyée par la fonction {`Nth ListOfQuestion N`} déjà présente dans Oz.

4. Modifier la "nouvelle liste de question", la variable unbound entrée en argument. La fonction {`RemoveList L Nth `} retire de la liste de question la "question optimale" qui sera renvoyée afin d'être posée.

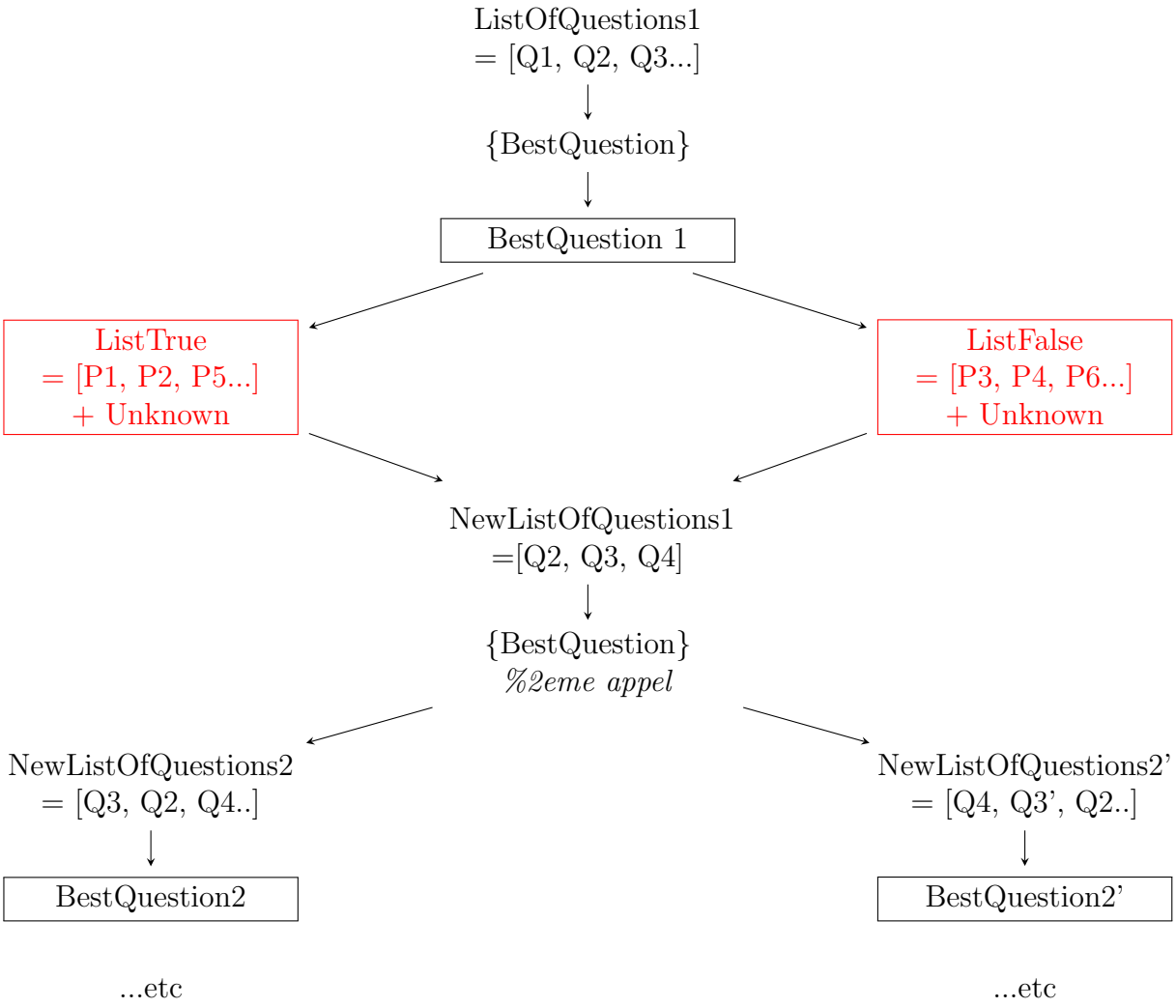

FIGURE 1 – **Schéma de fonctionnement de la fonction {BuildDecisionTree ...}**

2 GameDriver

Cette fonction utilise la fonction {`BuildDecisionTree..`} afin de coordonner et diriger le bon déroulement du jeu.
Elle prend en argument un arbre et renvoie toujours "unit", qui est un atome. Pour son implémentation, nous avons du utiliser deux fonctions de la librairie fournie par les coordinateurs du projet.

- **`projectLib.found`** : prend une liste de noms en arguments et affiche la liste dans l'interface graphique. Ce sont les solutions finales. Cela se produit lorsqu'il n'y a plus qu'un seul nom dans la liste ou qu'il n'y a plus de questions.
- **`projectLib.askquestion`** : prend une question en argument, l'affiche et renvoie la réponse introduite par l'utilisateur.

L'extension **`"bouton oups"`** rappelle l'arbre du gamedriver précédent, toujours présent dans l'accumulateur gamedriveracc.

3 Extensions

3.1 Incertitude dans la base de donnée

3.2 Questions non binaires

3.3 Incertitude du joueur (true, false, I don't know

3.4 Gérer les erreurs du joueur

3.5 Bouton "OUPS" (revenir à la décision précédente)