Algorithmique Avancée **Devoir de Programmation**

---

*Devoir à faire en **binôme**. Soutenance en séance de TD 10 (entre le 12/12 et 15/12).*
Rapport (de format pdf, d'une dizaine de pages) et Code Source à déposer sur Moodle au plus tard le 16/12 à 23h59, dans une archive nommée Nom1_Nom2_ALGAV.zip (où Nom1 et Nom2 sont vos noms de famille).
Langage de programmation libre.

Done
À tester
Buggé

1 Présentation

Le but du problème consiste à générer des diagrammes de décision binaires, réduits et ordonnés avec une approche analogue à celle présentée dans l'article de Newton et Verna :
A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams.
Le devoir permet en particulier de comparer nos résultats à leurs expérimentations, en particulier celles présentées dans les Figures 9, 10 et 11.
Il est attendu un soin particulier concernant la réflexion et la mise en place concernant les expérimentations dans ce devoir.

1.1 Échauffement

Question 1.1 Si le langage de programmation choisi ne permet pas de manipuler directement des entiers de taille arbitraire, choisir une bibliothèque permettant de le faire.

Question 1.2 Étant donné un entier naturel x de taille arbitraire, écrire une fonction `decomposition` renvoyant une liste de bits représentant la décomposition en base 2 de l'entier x, telle que les bits de poids les plus faibles soient présentés en tête de liste. Par exemple, puisque $38 = 2^1 + 2^2 + 2^5$, on obtient

```
>>> decomposition(38)
[False, True, True, False, False, True]
```

Question 1.3 Étant donné une liste de bits et un entier naturel n, écrire une fonction `completion` renvoyant soit la liste tronquée ne contenant que ses n premiers éléments, soit la liste complétée à droite par des valeurs `False`, de taille n. Par exemple,

```
>>> completion([False, True, True, False, False, True], 4)
[False, True, True, False]
>>> completion([False, True, True, False, False, True], 8)
[False, True, True, False, False, True, False, False]
```

Question 1.4 Étant donné deux entiers naturels x et n, définir une fonction `table` qui décompose x en base 2 et qui complète la liste obtenue afin qu'elle soit de taille n. Le résultat de cette fonction est appelé *table de vérité*.
Il suffit de composer les deux fonctions précédentes.

2 Arbre de décision et compression

Question 2.5 Définir une structure de données permettant d'encoder des arbres binaires de décision (cf. la partie Compression Arborescente issue du chapitre 1 de cours).

Question 2.6 Étant donné une table de vérité, écrire une fonction `cons_arbre` qui construit l'arbre de décision associé à la table de vérité T. Remarquons que l'arbre de décision issu de la table de vérité obtenue avec l'entier 38 et sur 3 variables est présenté dans la Figure 1.

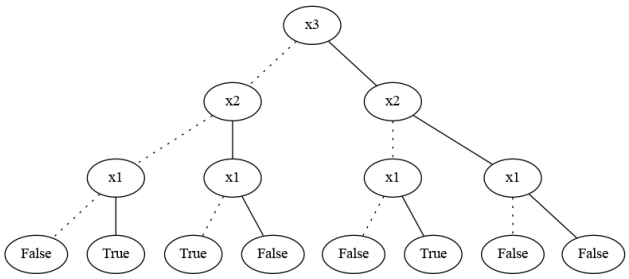

FIGURE 1 – Arbre de décision issu de la table de vérité de taille 8 construite sur l'entier 38

Question 2.7 Étant donné un arbre de décision T, définir une fonction `luka` qui à chaque nœud de T associe le mot de Lukasiewicz enrichi associé au sous-arbre enraciné en ce nœud. L'idée consiste à implémenter l'équivalent de ce qui est présenté en page 73 du cours.

Question 2.8 Étant donné un arbre de décision enrichi par les mots de Lukasiewicz pour chacun de ses nœuds, définir une fonction `compression` qui construit le graphe dirigé acyclique obtenu par compression de l'arbre en fusionnant les sous-arbres isomorphes. L'idée consiste à coder l'algorithme permettant d'obtenir le DAG présenté en page 75 du cours. La compression de l'arbre donné en Figure 1 est présenté sur la Figure 2.

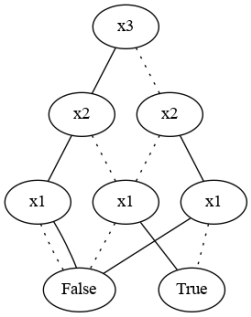

FIGURE 2 – DAG obtenu par compression de l'arbre de la Figure 1

Question 2.9 Étant donné un graphe (arbre de décision ou DAG), écrire une fonction `dot` qui construit un fichier représentant le graphe en langage *dot*. Une fois un fichier *dot* construit, l'application *graphviz* permet d'en donner la visualisation.
Des exemples de représentation en *dot* sont présentés ici : `https://graphs.grevian.org/example`.

3 Arbre de décision et ROBDD

On rappelle que le ROBDD (unique) associé à un arbre de décision est obtenu par compression de ce dernier en prenant en compte des règles un peu plus poussées que l'unique règle présentée en Question 2.8.
Les trois règles de compression sont présentées en bas de la page 4 de l'article :

- Terminal rule ;
- Deletion rule ;
- Merging rule.

Question 3.10 Étant donné un arbre de décision enrichi par les mots de Lukasiewicz pour chacun de ses nœuds, définir une fonction `compression_bdd` qui construit le ROBDD obtenu par compression de l'arbre en appliquant les règles précédentes. Le ROBDD associé à la Figure 1 est présenté sur la Figure 3.

Le but de la fin de cette section consiste à proposer une étude de la complexité au pire cas de l'algorithme `compression_bdd`. *Nous prenons comme mesure de complexité le nombre de comparaisons de caractères.*

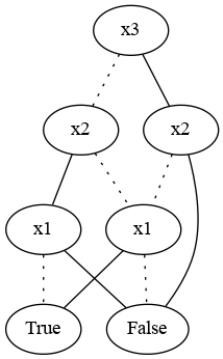

FIGURE 3 – ROBDD associé à l'arbre de décision de la Figure 1

Voilà le mot de Lukasiewicz associé à l'arbre représenté en Figure 1 *:*
`x3(x2(x1(False)(True))(x1(True)(False)))(x2(x1(False)(True))(x1(False)(False)))`

Question 3.11 Prouver que la longueur du mot de Lukasiewicz de la racine d'un arbre de hauteur h (qui s'écrit avec au plus c_h chiffres) est majorée par

$$\ell_h = (10 + c_h) \cdot 2^h - (5 + c_h).$$

Afin d'obtenir le résultat, on commencera par justifier qu'il y $2^h - 1$ nœuds internes, 2^h feuilles dans chaque arbre. On obtient pour borne supérieure de la longueur du mot précédent 82 (la longueur exacte est 79).

Question 3.12 On suppose que l'algorithme de compression passant de l'arbre de décision au ROBDD compare, au pire des cas toutes les chaînes associées aux nœuds d'un même niveau dans l'arbre. On utilise une comparaison naïve entre chaînes de caractères : il faut comparer les 2 chaînes caractère par caractère. Montrer que la complexité, au pire cas de l'algorithme de compression est majorée par $\Theta(2^{2h})$.

Question 3.13 Quelle est la complexité de l'algorithme, au pire cas, en fonction de n si n est le nombre de nœuds (internes et feuilles) qu'il possède ?

4 Étude expérimentale

La dernière partie du devoir consiste à reproduire les expérimentations des Figures 9, 10 et 11 de l'article mentionné ci-dessus. Normalement votre code devrait être plus efficace, et vous devriez avoir de meilleurs résultats par rapport aux durées de calcul présentées dans la Figure 10.

Question 4.14 Refaire les courbes de la Figure 9. Vous est-il possible de calculer la distribution exacte jusqu'à 5 variables ? Le cas échéant, en combien de temps ?

Question 4.15 Faire des courbes similaires à celles présentées en Figure 10.

Question 4.16 Indiquer dans un tableau, tel celui de la Figure 11, les informations relatives à vos expérimentations de la Figure 10.

5 Pour aller plus loin...

Question 5.17 Implémenter les algorithmes mentionnés à partir de la page 90 du cours et permettant de combiner de façon efficace deux ROBDD.

Question 5.18 Proposer une série de tests permettant d'avoir une idée si votre implémentation est correcte.