UE d'Ouverture **Devoir de Programmation**

---

*Devoir à faire en **binôme**. Langage de programmation **OCaml**.*
Les enseignants décideront d'un ordre de passage pour une soutenance le 07/11/23 ou le 08/11/23.
Deux rendus sont nécessaires à effectuer sous moodle :

1. Pour le 05/11 à 23h59 : Rapport et Code Source dans une archive nommée *Nom1_ Nom2_ OUV.zip*
2. Pour le 14/11 à 23h59 : Nouvelles versions du Rapport et Code Source + **Présentation** dans une archive nommée *Nom1_ Nom2_ OUV.zip*

1 Présentation

Le but du problème consiste à générer des diagrammes de décision binaires particuliers abrégé ZDD par Knuth [1, Section 7.1.4] et originellement défini par Minato [2] sous le doux nom de *Zero-Suppressed Binary Decision Diagram*.

Le devoir permet en particulier de comparer les temps de génération de ces objets, obtenus par compression arborescente, suivant les structures de données sous-jacentes. Il est attendu un soin particulier concernant la réflexion et la mise en place concernant les expérimentations dans ce devoir.

1.1 Échauffement

Il est conseillé d'utiliser le module Int64 permettant de manipuler des entiers signés sur 64 bits exactement. Mais si on veut utiliser tous les 64 bits, il faut manipuler aussi le bit de signe. Il faudra bien expliquer vos choix.

Question 1.1 On va devoir utiliser des entiers avec grande précision. Pour simplifier notre approche, nous définissons des *listes d'entiers* 64 bits avec insertion en fin de liste. Ces listes d'entiers représentent les *grands entiers*. Préparer la structure de données et les primitives d'insertion et de récupération (et suppression) de la tête de la liste.

Par exemple, si nous souhaitons représenter l'entier 2^{100}, nous allons utiliser deux entiers 64 bits, le premier étant $2^{100} \mod 2^{64} = 0$ où $\mod$ est le reste de la division entière et le second étant $2^{100}/2^{64} = 2^{36}$ où / représente la division entière. Notre liste sera constituée en tête du premier entier, suivi en 2ème élément du second entier.

Par la suite, on représente les listes représentant un entier comme sur l'exemple suivant pour 2^{100} : $[0; 2^{36}]$.

Question 1.2 Étant donné un entier naturel x de taille arbitraire représenté dans une liste d'entiers, 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 sont présentés en tête de liste. On remarque que les bits de poids faible de x sont les bits de poids faible dans le premier éléments de la liste représentant x. Par exemple, puisque $38 = 2^1 + 2^2 + 2^5$, on obtient

```
>>> decomposition([38])
[false; true; true; false; false; true]
```

Et l'appel à `decomposition` sur $[0, 2^{36}]$, retourne une liste de 101 bits, donc les 100 les plus à gauche sont $false$ et le dernier est $true$.

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, de taille n, complétée par des valeurs `false`. 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 une liste de bits, écrire une fonction `composition` qui construit l'entier représenté en liste (en base 10) dont la liste de bits correspond à l'écriture binaire.

Question 1.5 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 certaines fonctions précédentes.

Par la suite nous souhaitons construire des tables de vérité aléatoire. Dans ce but nous générerons des grands entiers aléatoirement puis les convertirons en table de vérité.

Question 1.6 Pour constituer un entier aléatoire sur maximum n bits, on construit une liste d'entiers contenant ℓ entiers avec $\ell = n/64$, chaque entier étant sur 64 bits, suivi d'un entier aléatoire inférieur à $n - \ell \times 64$. Ainsi si on souhaite un entier aléatoire contenant au plus 100 bits, on génère aléatoirement un entier sur 64 bits, suivi d'un entier strictement plus petit que 2^{36}. Définir une fonction `GenAlea` prenant en entrée une valeur n et générant un grand entier aléatoire de n bits au maximum.

2 Arbre de décision

Question 2.7 Définir une structure de données permettant d'encoder des arbres binaires de décision. Il s'agit d'une structure de données arborescente dont les nœuds internes contiennent un entier égal à sa profondeur [1] ; chaque nœud interne possède deux enfants qui sont des arbres de décision et les feuilles contiennent un booléen, *true* ou $false$.

Question 2.8 Étant donné une table de vérité, écrire une fonction `cons_arbre` qui construit l'arbre de décision associé à la table de vérité T. Il s'agit d'un arbre binaire totalement équilibré, dont les nœuds internes ont pour étiquette la valeur de leur profondeur et les feuilles sont étiquetées (via le parcours préfixe) avec les éléments de T.

Remarquons que l'arbre de décision issu de la table de vérité obtenue avec le grand entier [25899] et sur 4 variables est présenté dans la Figure 1.

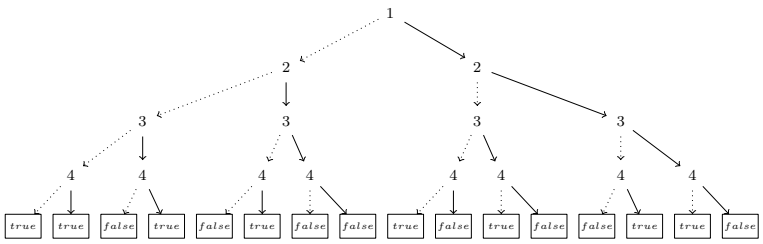

FIGURE 1 – Arbre de décision issu de la table de vérité de taille 16 construite sur le grand entier [25899]

Question 2.9 Étant donné un nœud N de l'arbre (interne ou feuille), écrire une fonction, nommée `liste_feuilles` et qui construit la liste des étiquettes des feuilles du sous-arbre enraciné en N, liste ordonnée de la feuille la plus à gauche jusqu'à celle la plus à droite.

1. La profondeur d'un nœud est égale à sa distance à la racine de l'arbre.

3 Compression de l'arbre de décision et ZDD

L'objectif de cette section consiste à proposer deux approches (l'une plus élémentaire que l'autre) et permettant par compression de l'arbre de décision de construire l'unique ZDD associé.
Voilà les règles de compression de l'arbre :

- **règle-M :** Si deux nœuds M et N sont les racines de sous-arbres ayant le même résultat pour `liste_feuilles`, alors les arêtes pointant vers N sont remplacées par des arêtes pointant vers M dans toute la structure ; puis le nœud N est supprimé de la structure.
- **règle-Z :** si l'enfant droit de N pointe vers $false$, alors toutes les arêtes pointant vers N sont remplacées par des arêtes pointant vers l'enfant gauche de N ; puis le nœud N est supprimé de la structure.

Après avoir utilisé ces règles aussi longtemps que possible l'arbre de décision de départ est compressé en le graphe ZDD lui correspondant.

Ces règles sont confluentes. On peut les appliquer dans l'ordre que l'on souhaite et sur les nœuds que l'on souhaite, on obtiendra à terme toujours la même structure compressée : l'unique ZDD.

Pour la suite, les nœuds d'un arbre pourront être utilisés pour être des nœuds d'un graphe, puisque les ZDD sont des graphes.

3.1 Compression avec historique stocké dans une liste

Question 3.10 Définir une structure de données permettant d'encoder une liste, nommée `ListeDejaVus` par la suite, dont les éléments sont des couples avec la première composante étant un grand entier (i. e. une liste d'entiers), et la seconde composante un pointeur vers un nœud d'un graphe.

Voilà l'algorithme élémentaire de compression d'un arbre de décision.

- Soit G l'arbre de décision qui sera compressé petit à petit. Soit une liste `ListeDejaVus` vide.
- En parcourant G via un parcours suffixe, étant donné N le nœud en cours de visite :
  - Calculer la `liste_feuilles` associées à N (le nombre d'éléments qu'elle contient est une puissance de 2).
  - Si la deuxième moitié de la liste ne contient que des valeurs `false` alors remplacer le pointeur vers N (depuis son parent) vers un pointeur vers l'enfant gauche de N
  - Sinon, calculer le grand entier n correspondant à `liste_feuilles` du sous-arbre enraciné en N ;
  - Si n est la première composante d'un couple stocké dans `ListeDejaVus`, alors remplacer le pointeur vers N (depuis son parent) par un pointeur vers la seconde composante du couple en question ;
  - Sinon ajouter en tête de `ListeDejaVus` un couple constitué du grand entier n et d'un pointeur vers N.

Question 3.11 Encoder l'algorithme précédant dans une fonction nommée `CompressionParListe` prenant G et une liste vide `ListeDejaVus` comme arguments.

Question 3.12 Étant donné un graphe (ou arbre de décision), écrire une fonction `dot` qui construit un fichier représentant le graphe en langage *dot*. Utiliser la même sémantique : enfant gauche relié au parent via une arête en pointillés et enfant droit relié via une arête pleine. 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`.

Question 3.13 Construire l'arbre associé au grand entier [25899] et vérifier via votre affichage qu'il s'agit bien de celui représenté sur la Figure 1.

Question 3.14 Compresser l'arbre précédent, et vérifier que sa représentation est la même que celle de la Figure 2

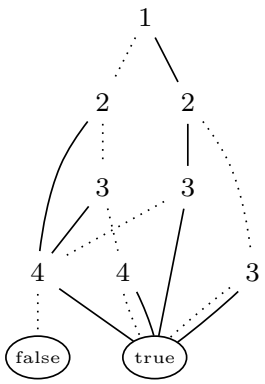

FIGURE 2 – ZDD construit sur le grand entier [25899]

4 Compression avec historique stocké dans une structure arborescente

Question 4.15 Définir une structure arborescente de données `ArbreDejaVus` permettant d'encoder un arbre binaire tel que l'arête gauche est associé au booléen $false$ et l'arête droite au booléen $true$. Les nœuds sont étiquetés avec un pointeur vers un nœud d'un graphe (ou ne sont pas étiquetés suivant le cas).

L'association d'un booléen à chaque arête permet de se déplacer dans un arbre via une table de vérité. Ainsi pour atteindre le nœud associé à $[false; true; true; false]$, on descend à gauche depuis la racine, puis 2 fois à droite et finalement à gauche.

On va utiliser `ArbreDejaVus` en tant qu'arbre de recherche pour stocker les pointeurs vers des sous-arbres déjà vus.

Question 4.16 Adapter l'algorithme élémentaire pour utiliser l'arbre de recherche `ArbreDejaVus` au lieu de la `ListeDejaVus`.

Question 4.17 Encoder cet algorithme dans une fonction `CompressionParArbre`.

Question 4.18 Vérifier que vous obtenez des résultats identiques avec les deux algorithmes de compression.

5 Analyse de complexité

Question 5.19 Après avoir défini une notion de taille des problèmes considérés, puis une mesure de complexité naturelle pour ces problème de compression en ZDD, calculer et prouver la complexité au pire de chacun des deux algorithmes de compression.

6 Étude expérimentale

La dernière partie, assez ouverte, vous permet de réfléchir quelles seraient les notions intéressantes à tester expérimentalement. On peut penser par exemple au taux de compression, à la vitesse d'exécution, à l'espace mémoire utilisé (rapport entre la taille de l'arbre et celle du ZDD associé).

Question 6.20 Proposer différentes courbes expérimentales avec représentation adéquate et des données tirées aléatoirement uniformément au hasard.

Question 6.21 Étant donné une taille n fixée de tables de vérité (suffisamment grande), calculer une approximation de la distribution de probabilité des tailles des ZDD.

Références

[1] D. E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms*. Addison-Wesley Professional, 2011.

[2] S.-I. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.