Rapport de projet

ALGAV - M1 STL

Rédigé par :

ELDAKAR Joumana

ZHANG Zimeng

Année universitaire 2021/2022

TABLE DES MATIÈRES

1 Introduction

Un projet d'algorithme de programmation basé sur la structure de données de l'arbre binaire de décision, et l'effectuation des opérations sur cet arbre de décision. Des opérations comme le mot de Lukasiewicz, la compression arborescente et la compression de ROBDD. Et aussi de tester la complexité de l'algorithme du code implémenté.

Le calcul du temps d'exécution du programme et l'efficacité de la complexité du algorithme effectuer. Tester le code avec tous les cas possible et générer un graphe pour comparer les résultats de chaque test.

2 Choix du langage de programmation

On a choisi Python pour implémenter le projet parce que c'est une langage de programmation par objet et cela nous aide à créer la structure de données de l'arbre binaire de décision.

Au début, on a choisi Java pour implémenter le projet mais on a rencontré quelques contraintes d'implémentation des fonctionnalités, donc on a basculé à Python parce que son syntaxe est moins facile par rapport à Java, on peut traduire rapidement nos pensées en langage de programmation.

3 Échauffement

3.1 La fonction decompositon

Cette fonction est de décomposer un entier en liste des **True** ou **False**. On commence par transformer les entiers en bits par la fonction de Python **format()** avec la premier argument est un nombre entier qui est l'argument de la fonction et le deuxième argument est le format, dans ce cas

là c'est "b" (vaut binaire). La transformation des 0 et 1 en False et True respectivement et on les stocke dans une liste. La fonction retourne cette liste.

Test :

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

3.2 La fonction completion

Cette fonction prend deux arguments, le première est une liste de True et False et le deuxième est un nombre entier positif n qui indique la longueur de cette liste. Si n est plus supérieur à la taille de la liste, on complète les indices par des False. Sinon on renvoie une liste de cette taille

Test :

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

3.3 La fonction table

On utilise les fonctions **decomposition** et **completion** pour transformer l'entier en liste de Booléan et retourne une completion de cette liste

Test :

```
table(38, 8)
[False, True, True, False, False, True, False, False]
```

4 Arbre de décision et compression

4.1 Structure de l'Arbre Binaire de Décision

La strucutures est un structure de l'arbre binaire Avec les attributs : La valeur du nœud, le mot Lukasiewicz associé à ce nœud, un id unique pour chaque noeud, le fils gauche et le fils droit. Avec les méthodes de getters et setters.

4.2 Construction de l'Arbre Binaire de Décision

Pour construire un arbre à partir d'une liste Booléen est une fonction récursive, le nombre des variables représenté dans l'arbre et égale à log(taille de la liste). On divise la liste en deux parties la première partie c'est la partie du fils gauche de l'arbre et l'autre le fils droit, avec une compléxité de O(n log(n)).

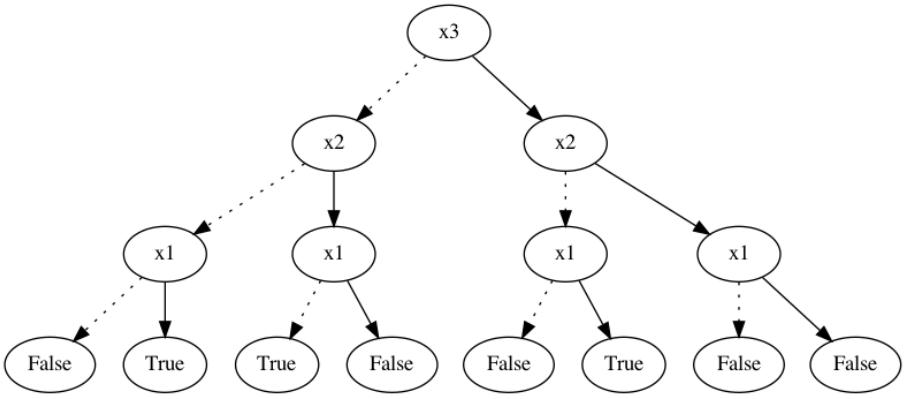

Figure 1: Arbre Binaire Décision.

4.3 Le mot de Lukasiewicz

En utilisant le parcours postfixe dans chaque noeud on stocke dans l'attribut **lukaval** le mot Lukasiewicz associé à ce noeud.

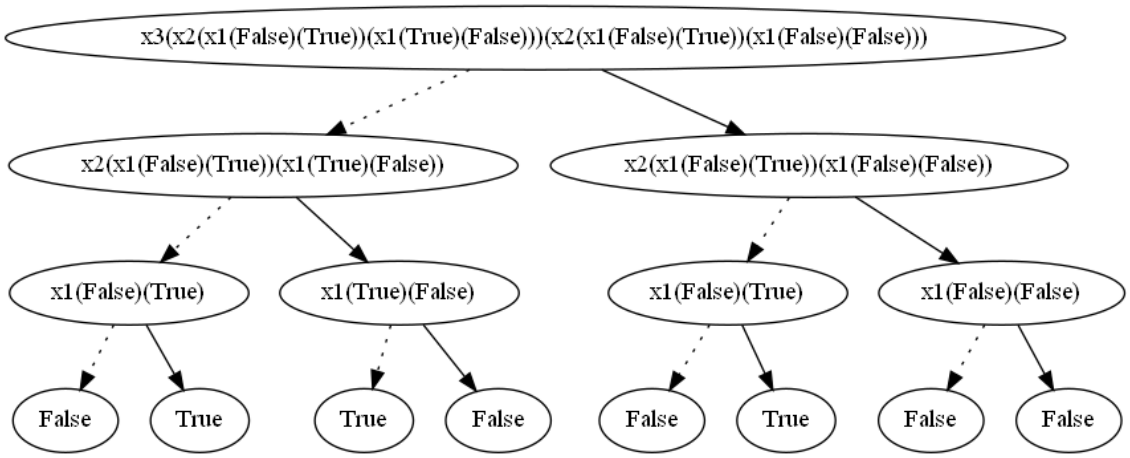

Figure 2: Arbre en affichant les mots Lukasiewicz pour chaque noeud.

4.4 Compression de l'arbre

Étant donnée un arbre enrichie par le mot de luka, en utilisant le parcours préfixe, si la visite est dans un nœud et pas une feuille, appel récursivement la fonction compression pour chaque fils et stocker le noeud dans un dictionnaire qui stocke tous les valeurs de mot luka dans l'arbre sans doublon, si la valeur est déjà existé dans le dictionnaire renvoie le noeud .

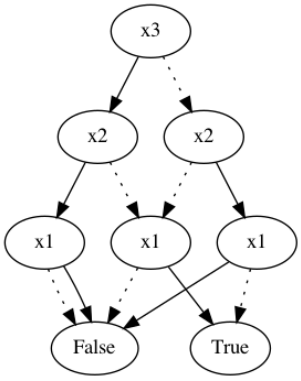

Figure 3: Arbre Binaire Décision Compressé.

4.5 Langage .dot

La visualisation de l'arbre est générée par le langage **dot**, et affichée par **graphviz**. Une fonction auxiliaire **listNoeud** est crée pour la fonction **dot** : cette fonction crée une liste de liste de chaque noeud tel que les sous-listes est de structure : [Noeud, son fils gauche, son fils droit]

Deux fonctions de dot:

- **dot(a)** : la création du fichier de type .dot dans le répertoire de l'arbre a.
- **dot_py(a)** : la création du fichier de langage .dot et la visualisation avec **graphviz** dans un fichier de type .png qui est associé de l'arbre a.

5 Arbre de décision et ROBDD

5.1 La Compression BDD

En appelant cette fonction avec un arbre déjà compressé dans l'argument, donc il n'existe pas des nœuds qui se répètent, il reste la deuxième condition de la compression bdd c'est si les deux

fils d'un nœud sont égaux (ont le même id). La suppression de la structure du nœud d'un de deux fils, en gardant l'autre et pointer ces deux fils vers ce nœud.

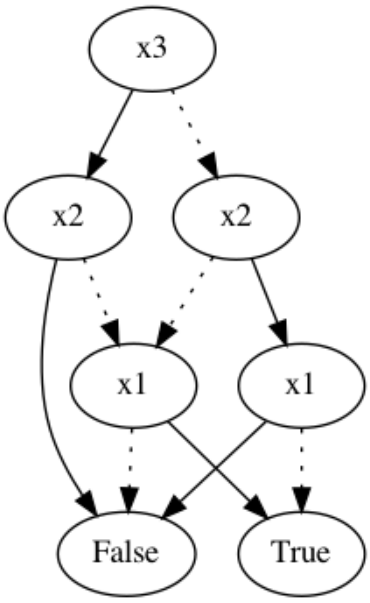

Figure 4: ROBDD.

6 Preuves

Question 3.11

On sait que dans l'étage 0, on a $2^0 = 1$ noeud, pour chaque étage il y a 2^i noeuds, donc le nombre de noeuds internes est égal à la somme des noeuds dans les étage de 0 à $h - 1$, ce sera $\sum_{i=0}^{h-1} 2^i = 2^h - 1$. Pour le nombre de feuilles est le nombre de noeuds dans l'étage h, ce sera 2^h.

Nous supposons que chaque feuille est 'False', alors la longueur de chaque feuille est 5, et la longueur de chaque nœud est 6 de plus que la longueur des deux feuilles (par exemple, la longueur du nœud x2 a 'x', '2', '(', ')', '(', ')' en plus de la longueur des deux enfants). En supposant que le nombre de nœuds internes est de 2^{h-1}et que l'arbre de feuilles est de 2^h , la longueur totale des

feuilles est de $5 \cdot 2^h$ et la longueur totale ajoutée par les nœuds est de $6 \cdot 2^{h-1}$, donc la longueur du mot de Lukasiewicz de la racine d'un arbre de hauteur $1 \leq h \leq 9$ est majorée par

$$l_h = 11 \cdot 2^h - 6$$

Question 3.12

Ici, nous supposons qu'il y a 2^i noeuds au niveau i, chaque noeud interne a une longueur de $11 \cdot 2^{h-i} - 6$ et le nombre de feuilles est de 2^h. Nous prenons 5 comme longueur d'une feuille, où nous calculons le nombre de noeuds comparés par noeud qui est la somme des longueurs de chaque noeud interne et de chaque feuille. Voici la preuve:

$$\sum comparaison = \sum comparaisonNoeud + \sum comparaisonFeuille$$

$$= \sum_{i=0}^{h-1} (11 \cdot 2^{h-i} - 6) \cdot 2^i + 5 \cdot 2^h$$

$$= \sum_{i=0}^{h-1} (11 \cdot 2^{h-i}) \cdot 2^i - 6 \cdot \sum_{i=0}^{h-1} 2^i + 5 \cdot 2^h$$

$$= 11h \cdot 2^h - 6(2^h - 1) + 5 \cdot 2^h$$

$$= 11h \cdot 2^h - 2^h + 6$$

$$= (11h - 1) \cdot 2^h + 6$$

En conclusion, au pire cas de l'algorithme de compression passant de l'arbre de déecision de hauteur $h \in N$ au ROBDD est majorée par

$$11h \cdot 2^h$$

Question 3.13

Nous supposons que le nombre de nœuds est somme de nombre de noeuds internes $2^h - 1$ et nombre de feuilles 2^h, ce sera $n = 2^{h+1} - 1$ donc h est égal à $\log_2(n+1) - 1$. En remplaçant h par n dans le formule précédente, nous obtenons:

$$11 \cdot \log_2(n+1) - 1 \cdot 2^{\log_2(n+1)-1}$$

donc au pire cas, en fonction de n si n est le nombre de nœuds (internes et feuilles), la complexité de l'algorithme est :

$$n \log_2 n$$

Question 3.14

Pour $1 \leq h \leq 9$, on a la longueur du mot de la racine:

$$l_h = 11 \cdot 2^h - 6$$

Pour $10 \leq h \leq 99$, on a la longueur du mot de la racine:

$$l_h = 12 \cdot 2^h - 7$$

Pour $100 \leq h \leq 999$, on a la longueur du mot de la racine:

$$l_h = 13 \cdot 2^h - 8$$

Pour toute valeur de h, on aura:

$$l_h = (11 + \lfloor \log_{10} h \rfloor) \cdot 2^h - (6 + \lfloor \log_{10} h \rfloor)$$

donc la complexité se sera:

$$= \sum_{i=0}^{h-1} ((11 + \lfloor \log_{10} h \rfloor) \cdot 2^{h-i} - (6 + \lfloor \log_{10} h \rfloor)) \cdot 2^i + 5 \cdot 2^h$$

est peut être majorée par:

$$(11h + \lfloor \log_{10} h \rfloor \cdot h - \lfloor \log_{10} h \rfloor) \cdot 2^h + \lfloor \log_{10} h \rfloor$$

7 Étude expérimental

La table de vérité est générée en appelant la fonction table dans la section échauffement. n dans table(x, n) est la taille de la table de vérité correspondant aux différentes variables, par exemple, pour 3 variables, n est égal à 2^3, indiquant que les 3 variables auront une table de vérité de longueur 8. x est la somme du nombre d'expressions booléen que les différentes variables peuvent avoir, et dans le cas de 3 variables, il peut y avoir $2^{2^3} = 256$ expressions booléennes possibles.

Nous construisons ensuite un arbre de décision binaire basé sur la table de vérité, nous le compressons pour obtenir le nombre de nœuds, nous utilisons un dictionnaire pour stocker le nombre d'occurrences de arbre avec different nœuds, et enfin nous l'affichons sous un graphe.

Question 4.15

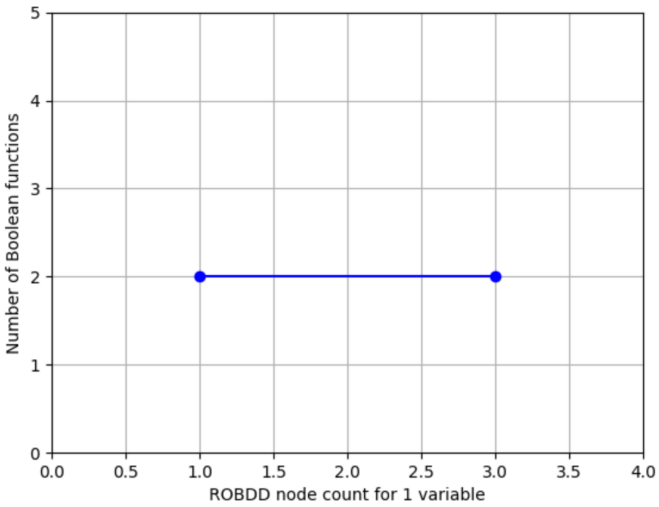

Figure 5: Résuiltat de 1 variable.

Pour 1 variable, nous avons un total de quatre expressions booléennes, qui, après compression, apparaissent deux fois sous la forme d'un nœud et deux fois sous la forme de trois nœuds

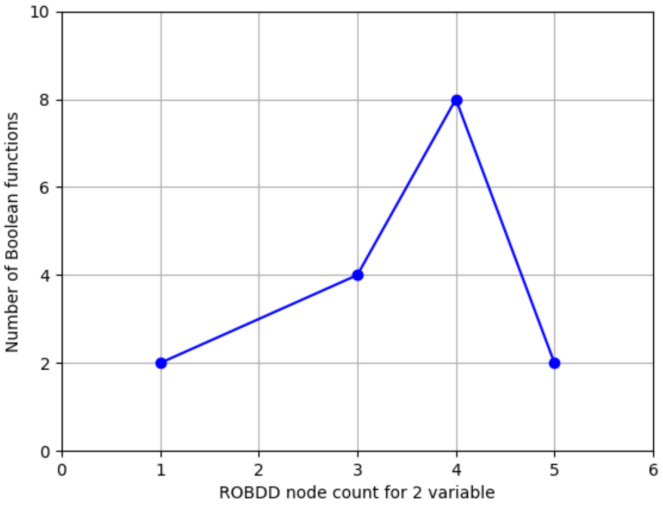

Figure 6: Résuiltat de 2 variables.

Pour 2 variables, nous avons un total de 16 expressions booléennes, dans lesquelles les cas de nœuds comprimés sont, respectivement, deux occurrences d'une forme de nœud, quatre occurrences de trois nœuds, huit occurrences de quatre nœuds et deux occurrences de cinq nœuds

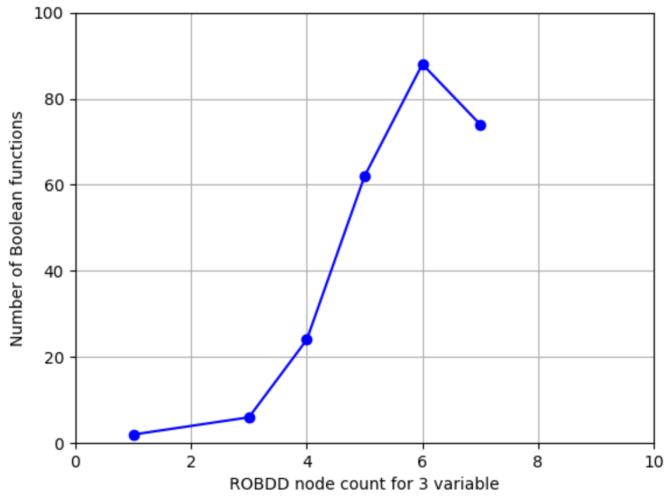

Figure 7: Résuiltat de 3 variables.

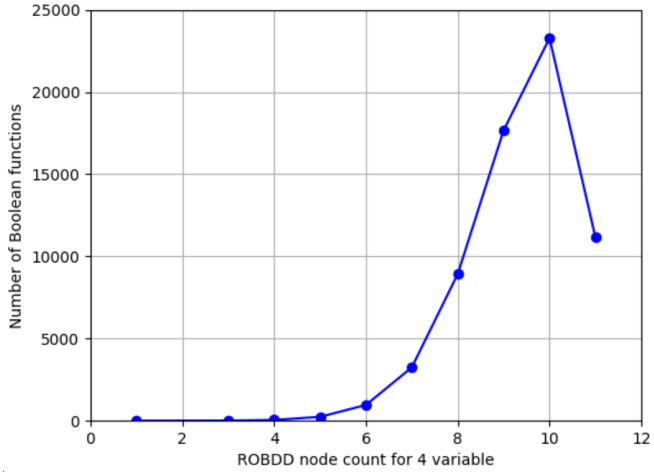

Figure 8: Résuiltat de 4 variables.

Question 4.16

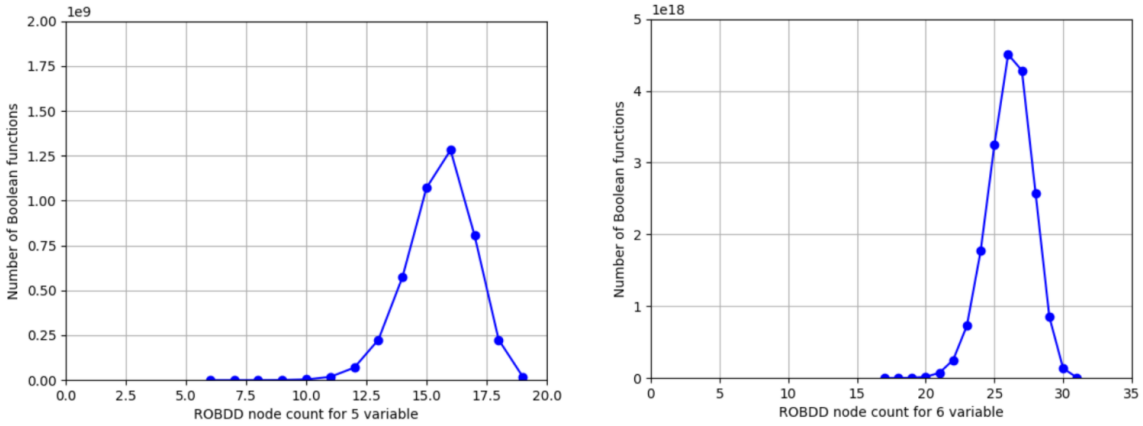

Figure 9: Résuiltat de 5 et 6 variables.

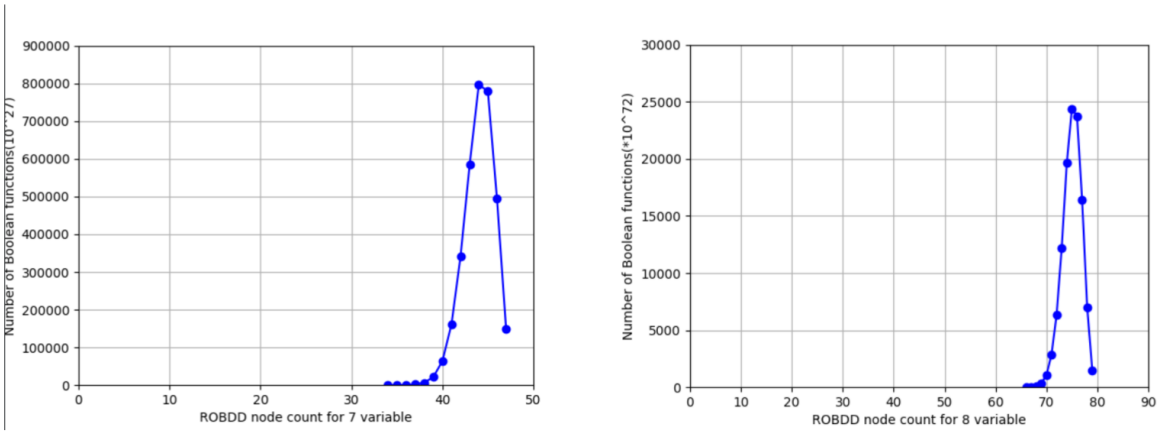

Figure 10: Résuiltat de 7 et 8 variables.

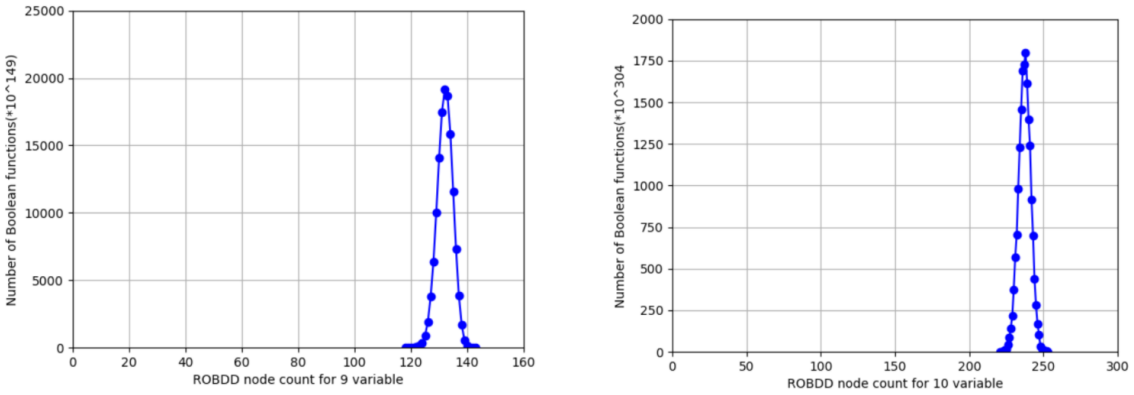

Figure 11: Résuiltat de 9 et 10 variables.

Question 4.17

No. variable	No. samples	No. unique size	Compute Time(s)	Second per ROBDD
5	500003	14	182.0483038	0.00036409
6	400003	15	317.4122945	0.00079352
7	486892	14	701.6183084	0.00144101
8	56343	15	168.4496319	0.00298971
9	94999	23	648.4444074	0.0068258
10	17975	33	272.9928327	0.01518736

Figure 12: Number of samples and compute times for generating the plots in Figure 9,10,11. The table also shows the number of unique ROBDD sizes which were detected for each value of n.

En comparant notre tableau avec celui de l'article, nous pouvons voir que nous obtenons des résultats différents pour le unique size, par exemple, toutes les valeurs booléennes sont False ou True, et la compression donne un nœud qui n'est pas présent dans nos résultats, nous pensons que la raison en est que l'échantillon testé manque certains cas particuliers, donc un résultat plus raisonnable serait d'inclure le cas de 1 noeud.

8 Test

pour tester les fichiers :

- **echauffement.py** : décommenter la ligne 61.
- **arbre_decision_comp.py**:
  - pour tester la fonction **cons_arb** : décommenter de la ligne 170 à 174
  - pour tester la fonction **luka** : décommenter de la ligne 178 à 183
  - pour tester la fonction **compression** : décommenter de la ligne 186 à 192

- **arbre_ROBDD.py** : pour tester la fonction **compression_bdd** décommenter de la ligne 25 à 32
- **exprimentale.py** : décommenter à partir de la ligne 440 pour tester.

La totalité du projet ce trouve dans le fichier **projet_Algav.py**

NB : avant de passer à un autre ficher il faut les parties en commentaires soient en commentaires.

9 Conclusion

En réalisant ce projet, nous avons acquis une meilleure compréhension de la structure des arbre binaire décision. Comme nous avons changé de langage de programmation au début du projet, cela nous a également permis de mieux comprendre les caractéristiques des différents langages de programmation. Le choix du langage de programmation est également un test de notre compréhension des structures de données. Pour différents projets, nous devrions être flexibles et choisir un langage plus adaptable pour implémenter nos structures, ce qui permet non seulement de gagner du temps mais aussi de rendre l'algorithme plus efficace et les résultats expérimentaux meilleurs.

En outre, en comparant les résultats des expériences, nous pouvons identifier certaines failles dans les expériences d'échantillonnage, ce qui peut également nous aider à mieux comprendre les propriétés de cette structure de données.

10 Annexe

Question 1.2

```
def decomposition(num):
    listBi = []
    bi = format(num, "b")
    bi = str(bi)
    for i in bi:
        if i == "1":
            listBi.append(True)
        else:
            listBi.append(False)
    listBi.reverse()
    return listBi
```

Question 1.3

```
def completion(liste, size):
    if size < len(liste):
        return liste[0:size]
    else:
        for i in range(len(liste), size):
            liste.append(False)
        return liste
```

Question 1.4

```
def table(x, n):
```

```
    liste1 = decomposition(x)
    liste2 = completion(liste1, n)
    return liste2
```

Question 2.5

```
class ArbreBinaire:
    def __init__(self, valeur, gauche=None,
                                droit=None, lukaval=None):
        self.valeur = valeur
        self.lukaval = lukaval
        self.gauche = gauche
        self.droit = droit
        self.id = str(round(random.uniform(0, 1), 20))

    def insert_gauche(self, valeur):
        if self.gauche == None:
            self.droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.gauche = self.gauche
            self.gauche = new_node

    def insert_droit(self, valeur):
        if self.droit == None:
            self.droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
```

```
23                 new_node.droit = self.droit
24                 self.droit = new_node
25
26         def get_valeur(self):
27             return self.valeur
28
29         def get_gauche(self):
30             return self.gauche
31
32         def get_droit(self):
33             return self.droit
34
35         def get_id(self):
36             return self.id
37
38         def get_luka(self):
39             return self.lukaval
```

Question 2.6

```
1  def cons_abr(liste):
2      taille = len(liste)
3      if taille == 1:
4          return ArbreBinaire(liste[0])
5      mid = taille // 2
6      return ArbreBinaire("x" + str(int(math.log2(taille))),
7      cons_abr(liste[:mid]), cons_abr(liste[mid:]))
```

Question 2.7

```
def luka(Noeud):
    if Noeud == None:
        return

    if Noeud.gauche != None:
        luka(Noeud.gauche)

    if Noeud.droit != None:
        luka(Noeud.droit)

    if not isinstance(Noeud.valeur, bool):
        Noeud.lukaval = str(Noeud.valeur) + "("
        str(Noeud.gauche.get_luka()) + ")" + "(" + str(
            Noeud.droit.get_luka()) + ")"
    else :
        Noeud.lukaval = Noeud.valeur
    return Noeud
```

Question 2.8

```
def compression(dicMap,Noeud):
    tree = dicMap.get(Noeud.get_luka())

    if(tree == None):
        dicMap[Noeud.get_luka()] = Noeud
        if(not isinstance(Noeud.valeur,bool)):
```

```
7                 Noeud.gauche = compression(dicMap,Noeud.gauche)
8                 Noeud.droit = compression(dicMap,Noeud.droit)
9             return Noeud
10        else:
11            return tree
```

Question 3.10

```
1  def compression_bdd(Noeud):
2
3      if isinstance(Noeud.lukaval,bool):
4          return Noeud
5      else:
6          Noeud.gauche = compression_bdd(Noeud.gauche)
7          Noeud.droit = compression_bdd(Noeud.droit)
8
9          if(Noeud.gauche.get_id()==Noeud.droit.get_id()):
10             return Noeud.get_droit()
11
12         return Noeud
```