SORBONNE UNIVERSITÉ

PROJET MU4IN511

OUVERTURE

Zero-Suppressed Binary Decision Diagram

Auteurs :

Truong Do Truong Thinh

Diallo Elhadj Alseiny

Encadrants :

Antoine Genitrini

Emmannuel Chailloux

Table des matières

Table des figures

1. Introduction

L'objectif de ce devoir est de concevoir un algorithme pour compresser un Arbre de Décision Binaire (Binary Decision Tree - BDD) en un Diagramme de Décision Binaire Sans Zéro (Zero-Suppressed Binary Decision Diagram - ZBDD). Les BDD sont couramment utilisés pour représenter des fonctions booléennes et sont particulièrement utiles en informatique pour des applications telles que l'optimisation de circuits, la vérification de modèles et la résolution de problèmes de décision.

Énoncé du problème

Le problème à résoudre est de développer un algorithme qui prend un BDD en entrée, généralement construit à partir de tableaux de vérité, et le comprime en un ZBDD. Les ZBDD sont une forme spécifique de diagramme binaire qui supprime les nœuds redondants, ce qui permet une représentation plus compacte et efficace des fonctions booléennes. Ce rapport aborde sur les étapes clés sont les suivantes :

Conversion du BDD en ZBDD : L'algorithme devra parcourir le BDD et construire un ZBDD équivalent en supprimant les nœuds redondants. Cela nécessitera une analyse approfondie de la structure du BDD.

Utilisation d'une structure de données : L'algorithme devra utiliser une structure de données appropriée pour représenter le ZBDD compressé. Cela peut être réalisé à l'aide d'une liste, d'un arbre binaire ou d'une autre structure de données optimisée pour ce problème.

Analyse de Complexité : Il est essentiel d'effectuer une analyse de la complexité de l'algorithme pour évaluer son efficacité. Cela inclut l'analyse du temps et de l'espace nécessaires pour effectuer la compression.

Étude Expérimentale : Une étude expérimentale est nécessaire pour évaluer les performances de l'algorithme. Pour ce faire, des données générées de manière aléatoire et uniforme peuvent être utilisées pour tester l'algorithme dans divers scénarios.

En fin de compte, la réussite de ce projet repose sur la capacité à créer un algorithme efficace pour compresser des BDD en ZBDD, en minimisant la taille de la structure de données résultante tout en préservant la sémantique des fonctions booléennes. Cette tâche est essentielle dans de nombreuses applications informatiques, où l'efficacité de la gestion des fonctions booléennes peut avoir un impact significatif sur les performances des systèmes.

2. Présentation du code

Structures utilisées

`bigInt` : une liste de `int64` pour représenter un nombre ayant une valeur plus grande que 2^{64}

`decisionBST` : une structure arborescente qui s'agit d'un BDD

`visitedNodesList` : une liste pour aider avec la compression par liste

`visitedNodesTree` : une structure arborescent pour aider avec la compression par arbre

Fonctions principales

```
composition : bool list -> bigInt
```

Renvoyer un bigInt à partir d'un tableau de vérité

```
decomposition : bigInt -> bool list
```

Renvoyer un tableau de vérité à partire d'un bigInt

```
genAlea : int -> bigInt
```

Renvoyer un bigInt aléatoire à partir de nombre de bits

```
createDecisionBST : bool list -> decisionBST
```

Renvoyer un BDD à partir d'un tableau de vérité

```
compressionList : decisionBST -> visitedNodesList -> decisionBST
```

Renvoyer un ZBDD après la compression du BDD par liste

```
compressionTree : decisionBST -> visitedNodesTree -> decisionBST
```

Renvoyer un ZBDD après la compression du BDD par arbre

Fonctions utilitaires/primitives

```
front : 'a list -> 'a
```

Renvoyer la tête de la liste

```
remove_head : 'a list -> 'a list
```

Renvoyer la liste avec la tête enlevée

```
intToBool : int64 -> bool list
```

Renvoyer un tableau de vérité à partir d'un int64

```
boolToInt : bool list -> int64
```

Renvoyer un tableau de vérité à partir du tableau de vérité

```
completion : bool list -> int -> bool list
```

Renvoyer un tableau de vérité fixé à une taille à partir d'un tableau de vérité

```
ceilBase2 : int -> float
```

Renvoyer un float arrondi au plus proche exponent de base 2

```
getFirstHalf : bool list -> bool list
```

Renvoyer la première moitié d'un tableau de vérité

```
getSecondHalf: bool list -> bool list
```

Renvoyer la dernière moitié d'un tableau de vérité

```
searchVisitedNodesList : bigInt -> visitedNodesList -> (bool * decisionBST)
```

Renvoyer un couple de bool * decisionBST trouvé dans la liste de nœuds déjà visités à partir d'un bigInt

```
listLeaf : decisionBST -> bool list
```

Renvoyer la liste de feuilles d'un BDD

```
addList: (bigInt * decisionBST) -> visitedNodesList -> vistedNodesList
```

Renvoyer la liste des nœuds déjà visités en ajoutant un nœud

```
generateDotFile : decisionBST -> string -> unit
```

Créer un fichier dot pour l'affichage

```
searchvisitedNodesTree : bool list -> visitedNodesTree -> (bool * decisionBST)
```

Renvoyer un couple de bool * decisionBST trouvé dans l'arbre de nœuds déjà visités à partir d'un bigInt

```
addTree: bool list -> decisionBST -> visitedNodesTree -> visitedNodesTree bool list
```

Renvoyer l'arbre des nœuds déjà visités en ajoutant un nœud

3. Structure : bigInt

Le but de cette structure s'agit de représenter les entiers ayant la valeur plus grande que 2^{64}. Notre implémentation se sert d'un tableau de `int64` dont chaque élément est un bit en base 64. Par exemple, pour représenter $2^{100} + 10$, le premier élément du tableau sera $(2^{100} + 10)$ mod $2^{64} = 10$, et donc $(2^{100} + 10) / 2^{64} = 2^{36}$, le résultat final sera [10L ; 2^{36}] qui peut être interprété en tant que $10 \times (2^{64})^0 + 2^{36} \times (2^{64})^1$. Ensuite, il est possible de décomposer la structure en un tableau de vérité de 101 bits qui représente l'entier $(2^{100} + 10)$ en base 2. En outre, on aura un outil pour générer des grands entiers comme ainsi de manière aléatoire uniforme.

Implémentation

Pour la décomposition d'un bigInt, on se sert d'une fonction auxiliaire `intToBool`

```
let rec intToBool : int64 -> bool list = fun n ->
    match n with
    | 0L -> []
    | x -> if ((logand x 1L) = 1L)
        then true :: (intToBool (shift_right x 1))
        else false :: (intToBool (shift_right x 1))
```

qui utilise des opérations sur les bits du `int64` à savoir, un `logic_and` et des `shift_right` et la fonction auxiliaire `completion` pour décomposer chaque élément dans le tableau `int64` en tableau de vérité de 64 bits et puis on concaténer tous les tableaux pour avoir le résultat final.

Quant à la composition, on divise le tableau de vérité en tranche de 64 bits et puis en fait un élément du tableau `int64` en utilisant `boolToInt`

```
let rec boolToInt : bool list -> int64 = fun list ->
    match list with
    | [] -> 0L
    | h :: t -> if h
        then Int64op.(1L + 2L * (boolToInt t))
        else Int64op.(0L + 2L * (boolToInt t))
```

qui s'agit d'un algorithme simple de calcul sur `int64`.

Pour la génération aléatoire d'un grand entier sur n nombre de bits, si n est plus grand que 64, on génère un nombre aléatoire sur les premiers 64 bits en appelant `Random.int64 (Int64.max_int)` pour en faire des premiers éléments du tableau `int64` et finalement pour le dernier élément on génère un nombre entre 0 et 2 à la puissance du nombre de bits qui restent.

```
let rec genAlea : int -> bigInt = fun n ->
    if (n < 64) then [Random.int64 (of_int(int_of_float(float_of_int 2**float_of_int n)))]
    else Random.int64 (Int64.max_int) :: (genAlea (n - 64))
```

Limitations

Théoriquement, d'après notre implémentation, un entier tel grand que (2^{63}+5) est censé être représenté en tant que [(2^{63}+5) L] vu que (2^{63}+5) mod 2^{64} = (2^{63}+5) et (2^{63}+5) / 2^{64} = 0. Pourtant, en tant qu'un entier sur 64 bits, `int64` a un bit de signe et donc est limité d'être entre [-2^{63}, 2^{63} -1], donc on ne peut pas représenter les valeurs entre [2^{63}, 2^{64}-1] même si elles sont les entiers sur 64 bits, et donc notre tableau `int64` ne peut pas contenir des valeurs dans cette intervalle, autrement dit, les bits en base 64 dans cette intervalle ne peuvent jamais être jamais représentés, et alors notre implémentation ne peut pas représenter les grands entiers qui ont un de ses valeurs dans sa décomposition en bits de base 64.

4. Arbre de décision

Notre Binary Decision Diagram (BDD) s'agit d'une structure arborescente qui permet de stocker la profondeur aux nœuds internes et les valeurs booléens aux feuilles.

Construction de l'arbre

On se sert d'une fonction utilitaire `ceilBase2` pour obtenir la hauteur de l'arbre à partir de la taille du tableau de vérité et donc le remplir avec `completion` pour qu'il ait une taille d'un exponentiel de base 2.

```
let rec createDecisionBST : bool list -> decisionBST = fun list ->
    let height = ceilBase2 (List.length list) in
    let list = completion list (int_of_float(2. ** height)) in
```

Et puis en descendant dans l'arbre, on se sert des deux fonctions auxiliaires `getFirstHalf` et `getSecondHalf` pour repartitionner le tableau en deux jusqu'aux feuilles où on insère les valeurs booléennes du tableau.

```
let rec aux list a =
    match (list,a) with
    | (h::t, x) when x = (int_of_float height) + 1 -> Leaf (h)
    | (list, x) -> Node (aux (getFirstHalf list) (x + 1) , x, aux (getSecondHalflist) (x + 1))
    in aux list 1
```

Le résultat de l'appellation sur un tableau obtenu de l'entier 10L est :

```
let tree = createDecisionBST (table [10L] 8);;
```

```
val tree : decisionBST =

  Node

   (Node (Node (Leaf false, 3, Leaf true), 2, Node (Leaf false, 3, Leaf
true)),

   1,

   Node (Node (Leaf false, 3, Leaf false), 2, Node (Leaf false, 3, Leaf
false)))
```

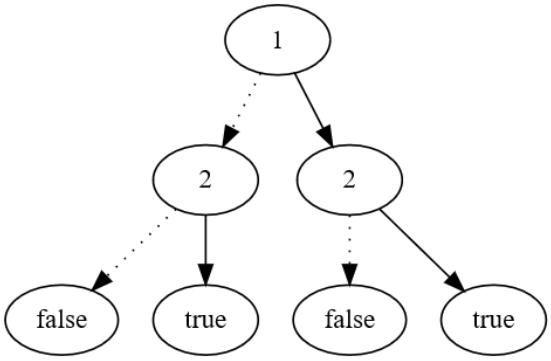

Figure 1 Arbre de décision de l'entier 10

5. Compression en ZBDD par liste

Algorithme Élementaire

L'algorithme élémentaire est décrit comme ci-dessous :

— Soit G l'arbre de décision qui sera compressé petit à petit. Soit une liste VisitedNodesList 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 VistedNodesList, 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 VisitedNodesList un couple constitué du grand entier n et d'un pointeur
vers N .

Implémentation

```
let rec compressionList : decisionBST -> visitedNodesList -> decisionBST =
fun tree list ->
```

On se sert d'une fonction auxiliaire afin d'éviter la liste de feuilles d'un nœud à chaque fois on le visite

```
let rec compression tree list =
match tree with
| Empty -> (Empty,[])
```

Dès qu'on visite une feuille on la recherche dans la liste des feuilles qu'on a déjà visitées pour la retourner sinon on l'ajoute dans la liste et on la retourne

```
| Leaf (b) ->
  let n = composition [b] in
  let (found, decision) = searchVisitedNodesList n list in
  if found then (decision, [b])
  else
  let _ = addList (n, tree) list in (tree, [b])
```

En parcourant l'arbre, on appelle récursivement la fonction auxiliaire sur le sous arbre gauche et le sous arbre droit

```
| Node (left, value, right) ->
    let (new_left,leftLeafs) = compression left list in
    let (new_right,rightLeafs) = compression right list in
```

Une fois tous les à droite d'un nœud sont `false` on remplace le pointeur vers le nœud depuis son parent vers l'enfant gauche du nœud

```
    if (List.for_all (fun x -> x = false) rightLeafs) then
       (new_left, leftLeafs @ rightLeafs)
```

Dans le cas contraire, on concatène tous les feuilles droites et gauches et puis on recherche dans la liste si on a déjà visité un nœud avec telles feuilles

```
    else
    let listLeafs = leftLeafs @ rightLeafs in
    let n = composition listLeafs in
    let (found, decision) = searchVisitedNodesList n list in
```

Si le nœud est bien dans la liste, on le retourne

```
    if found then (decision, listLeafs)
```

Sinon on le crée, on l'ajoute dans la liste et puis on le retourne

```
    else
```

```
      let newNode = Node (new_left, value, new_right) in
      let _ = addList (n, newNode) list in (newNode, listLeafs)
```

On retourne seulement le ZBDD finalement

```
    let (compresed,feuilles) = compression tree list in
    compresed
```

L'appellation sur l'arbre de décision construit de l'entier 25899 retourne

```
decisionBST =
Node
 (Node
   (Node (Node (Leaf true, 4, Leaf true), 3, Node (Leaf false, 4, Leaf
true)),
   2, Node (Leaf false, 4, Leaf true)),
 1,
 Node (Node (Leaf true, 3, Leaf true), 2,
  Node (Node (Leaf false, 4, Leaf true), 3, Leaf true)))
```

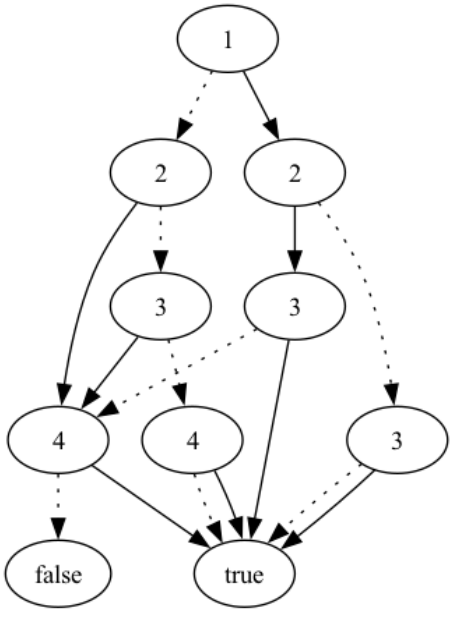

Figure 2 Compression de l'arbre par liste

6. Compression en ZBDD par arbre

Algorithme Élementaire

De la même manière que l'algorithme compressionList, on remplace la liste par une structure arborescente pour stocker les nœuds qu'on a déjà visités.

Implémentation

```
let rec compressionTree : decisionBST -> visitedNodesTree -> decisionBST = fun tree arbreData ->

    let rec compression tree arbreData =

    match !tree with

    | Empty -> (tree, [])
```

Dès qu'on visite une feuille on la recherche dans l'arbre des feuilles qu'on a déjà visitées pour la retourner sinon on l'ajoute dans l'arbre et on la retourne

```
| Leaf (b) -> let leafs = [b] in     let (found, decision) = searchVisitedNodesTree leafs arbreData  in

        if found

            then

                (decision, leafs)

            else

                let _= addTree [b] tree arbreData in

                (tree, leafs)
```

En parcourant l'arbre, on appelle récursivement la fonction auxiliaire sur le sous arbre gauche et le sous arbre droit

```
    | Node (left, value, right) ->

        let (new_left, leafLeft) = compression left arbreData in

        let (new_right, leafRight) = compression right arbreData in
```

Une fois tous les à droite d'un nœud sont `false` on remplace le pointeur vers le nœud depuis son parent vers l'enfant gauche du nœud

```
      if (List.for_all (fun x -> x = false) leafRight) then

            (new_left, leafLeft @ leafRight)
```

Dans le cas contraire, on concatène tous les feuilles droites et gauches et puis on recherche dans l'abre si on a déjà visité un nœud avec telles feuilles

```
      else

            let leafsList = leafLeft @ leafRight in
```

```
            let (found, decision) = searchVisitedNodesTree leafsList
arbreData in
```

Si le nœud est bien dans l'arbre, on le retourne

```
            if found then
                (decision, leafList)
```

Sinon on le crée, on l'insère dans l'arbre et puis on le retourne

```
else
                let newNode = Node (!new_left, value, !new_right) in
                let newRefNode = ref newNode in
                let _ = addTree leafsList newRefNode arbreData in
                (newRefNode, leafsList)
```

On retourne seulement le ZBDD finalement

```
    let (compresed,leafs) = compression tree arbreData in
    compresed
```

L'appellation sur l'arbre de décision construit de l'entier 25899 retourne

```
decisionBST = Node
 (Node
   (Node (Node (Leaf true, 4, Leaf true), 3, Node (Leaf false, 4, Leaf
true)),
   2, Node (Leaf false, 4, Leaf true)),
 1,
 Node (Node (Leaf true, 3, Leaf true), 2,
  Node (Node (Leaf false, 4, Leaf true), 3, Leaf true)))
```

On peut constater que l'arbre compressé est le même arbre obtenu en compression par liste.

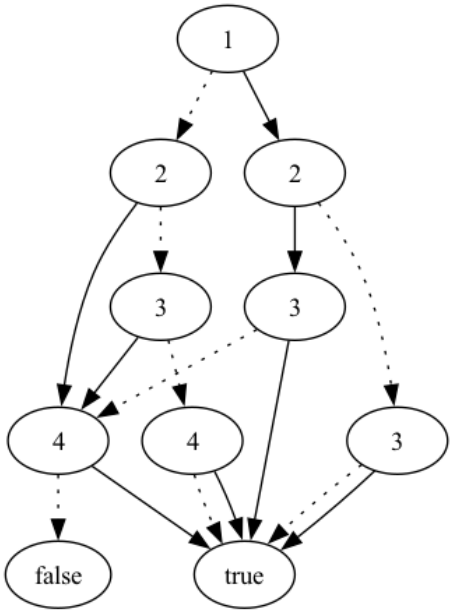

Figure 3 Compression de l'arbre par arbre

7. Analyse de complexité

Pour analyser la complexité des deux algorithmes de compression (CompressionList et CompressionTree), on va définir une notion de taille du problème, puis on mesure la complexité en termes d'opérations effectuées pendant la compression. Enfin, on examine la complexité au pire des deux algorithmes.

Notion de taille du problème

La taille du problème dépendra de la taille de l'arbre de décision initial. La taille de l'arbre sera mesurée en fonction du nombre de nœuds (internes et feuilles) et d'arêtes.

Mesure de complexité

La mesure de complexité consistera en comptant le nombre total d'opérations élémentaires effectuées par chaque algorithme. Les opérations élémentaires incluent les opérations de recherche, d'ajout et de parcours des nœuds dans l'arbre.

Complexité au pire des deux algorithmes

CompressionList : L'algorithme parcourt l'arbre de décision en effectuant une recherche et, le cas échéant, une opération d'ajout dans la structure vistedNodesList. La complexité de recherche dans une liste non triée est en moyenne O(n), où n est le nombre d'éléments dans la liste. La complexité d'ajout dans la liste est également O(1) en moyenne. Le parcours de l'arbre a une complexité de O(m), où m est le nombre total de nœuds dans l'arbre. La complexité au pire est donc O(m^2), car dans le pire cas, l'algorithme peut effectuer une recherche et un ajout pour chaque nœud de l'arbre.

CompressionTree : Cet algorithme utilise une structure d'arbre binaire de recherche visitedNodesTree pour gérer les nœuds déjà visités. La complexité moyenne de recherche dans un arbre binaire de recherche est O(log n), où n est le nombre de nœuds dans l'arbre. La complexité d'ajout dans un arbre binaire de recherche est également O(log n) en moyenne. Le parcours de l'arbre de décision a une complexité de O(m), où m est le nombre total de nœuds dans l'arbre. La complexité au pire est donc O(m * log(m)* log(m)), car l'algorithme effectue une opération de recherche ou d'ajout pour chaque nœud de l'arbre. Mais vu que Multiplier à nouveau par log(m) ne changerait pas la classe de complexité, car les facteurs constants (comme 2 * log(m)) sont généralement omis dans la notation "O". Ainsi, O(m * log(m)) est la manière conventionnelle de représenter la complexité de cet algorithme, car elle capture efficacement la relation entre la taille de l'arbre (m) et le nombre d'opérations logarithmiques nécessaires pour chaque nœud.

En comparant les deux algorithmes, on peut voir que CompressionTree a généralement une complexité plus faible que CompressionList en raison de l'utilisation d'une structure d'arbre binaire de recherche qui offre une recherche plus efficace que la recherche linéaire dans une liste non triée. Cependant, la complexité dépendra également de la structure spécifique de l'arbre de décision initial.

La complexité O(m * log(m)) est une borne supérieure, mais en pratique, la complexité réelle peut être bien meilleure. En effet, la complexité effective dépend fortement de la structure de l'arbre et de l'ordre dans lequel les nœuds sont traités. De plus, l'ajout de nœuds n'atteint qu'une seule fois une complexité de log(m) car il n'y a qu'un seul nœud de taille maximale dans l'arbre à compresser. Cette caractéristique contribue à réduire considérablement la complexité effective de l'algorithme CompressionTree dans de nombreux cas. Même en ce qui concerne la complexité de CompressionList, il est important de noter que la recherche ne peut jamais atteindre O(m). En effet, la seule manière pour que tous les nœuds de l'arbre se trouvent dans la liste à rechercher serait que nous les ayons déjà tous traités, auquel cas l'algorithme serait déjà terminé.

8. Étude expérimentale

Dans cette partie, on va avoir une étude expérimentale sur la compression par arbre vu que CompressionArbre est l'algo le plus rapide entre les deux.

Méthode

On va générer des données aléatoires et donc mesurer les ressources utilisées

```
let generate_random_compressed_tree = fun n ->
```

On génère les grands entiers aléatoires et puis les arbres de décision aléatoires

```
let randomBigInt = genAlea n in

let table = decomposition randomBigInt in
```

```
let randomTree = createDecisionBST table in
```

On va mesurer le taux de compression, le temps exécuté et la mémoire utilisée par le programme

```
let nNodes = countNodes randomTree in
let timeStart = Unix.gettimeofday() in
let memoryStart = (Gc.stat()).minor_words in
let compressedTree = compressionTree randomTree Empty3 in
let memoryEnd = (Gc.stat()).minor_words in
let timeEnd = Unix.gettimeofday() in
let time = (timeEnd -. timeStart ) in
let memory =  (memoryEnd -. memoryStart) *. 8. in
let memory = Float.to_int memory in
let nNodesCompressed = countNodes compressedTree in
let compressionRate = (float_of_int nNodesCompressed) /. (float_of_int nNodes) *. 100. in
n, time, memory, compressionRate
```

Résultat

On peut constater que le temps d'exécution de l'algorithme avec l'aide d'une liste est beaucoup pire que celui avec un arbre.

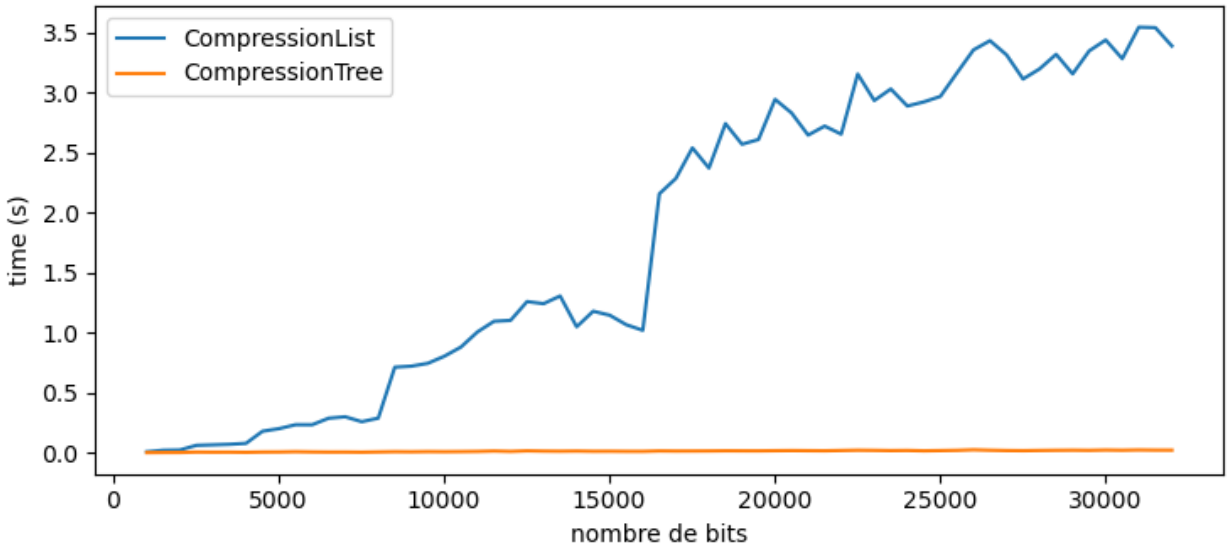

Figure 4 CompressionTree vs. CompressionList

En effet, il est évident que la performance de CompressionTree est dix fois plus rapide que CompressionList.

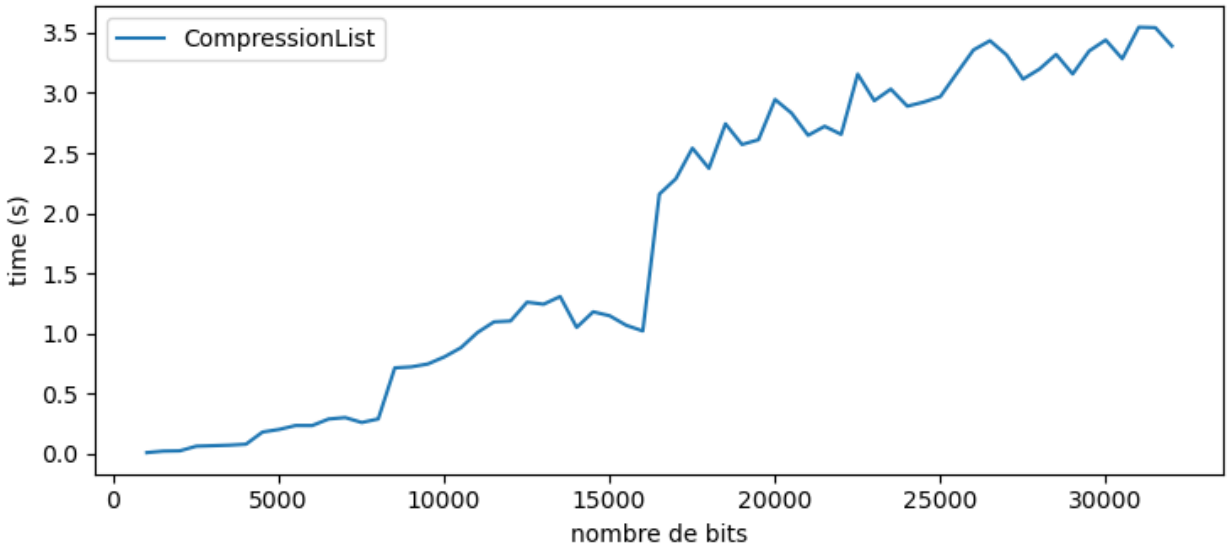

Figure 5 Temps d'exécution CompressionList

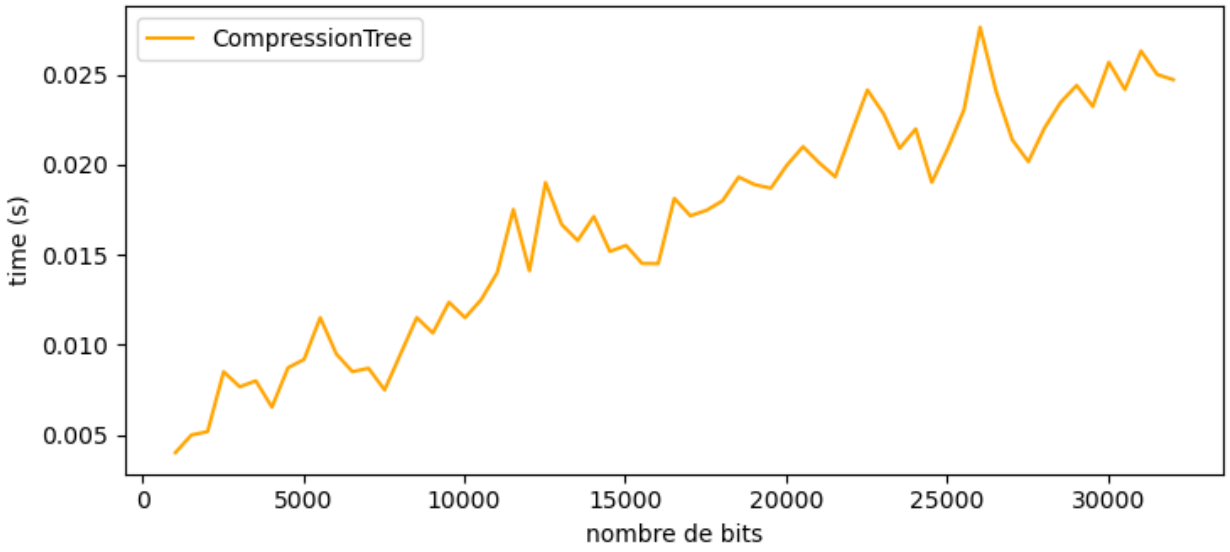

Figure 6 Temps d'exécution CompressionTree

Le taux de compression est borné par l'intervalle de 30% et 70%, celui-ci dépend du nombre de bits du grand entier et donc le nombre de bits dans le tableau de vérité et donc le nombre de feuilles de l'arbre de décision. On peut constater que le taux de compression le plus haut possible est atteint quand le nombre de bits du grand entier s'approche d'un nombre n de puissance de base 2, ce qui est le nombre exact de feuilles de l'arbre et le plus bas possible quand ce nombre n est le nombre qui suit un nombre de puissance de base 2. En résumé, il

s'agit d'un cycle de courbe croissant entre deux nombres de puissance de base 2.

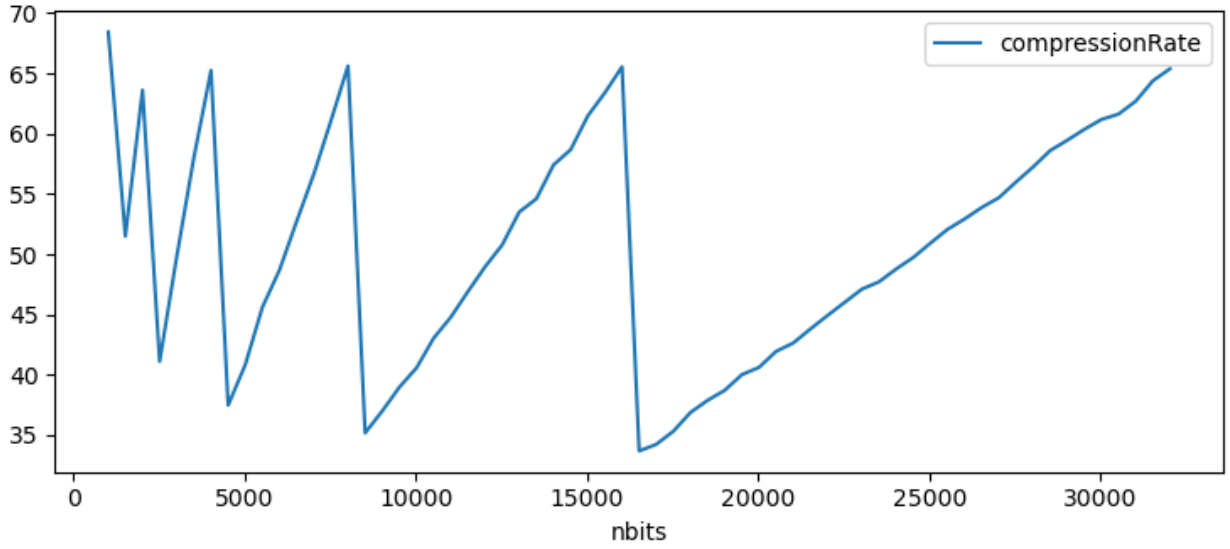

Figure 7 Taux de compression

Par rapport à la mémoire, on peut constater que l'évolution de mémoire utilisée est linéaire et que l'utilisation d'un arbre auxiliaire pour la compression est plus couteuse qu'une liste. En effet, stocker les nœuds dans une liste en tuples de `(bigInt * decisionBST)` est bien moins couteux qu'en faire des nœuds dans une structure arborescente, ce qui évidemment prend plus de place en mémoire.

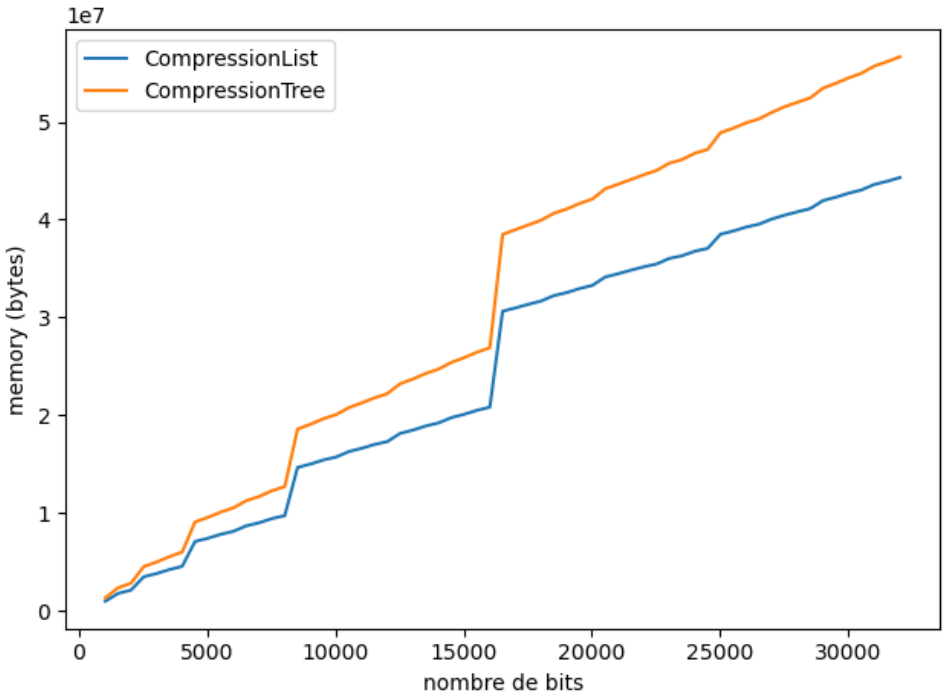

Figure 8 Memoire Utilisée

Ensuite, pour avoir une idée sur la distribution de la taille de ZDD, on génère 10000 ZDDs à partir de 1000 bits aléatoirement générés.

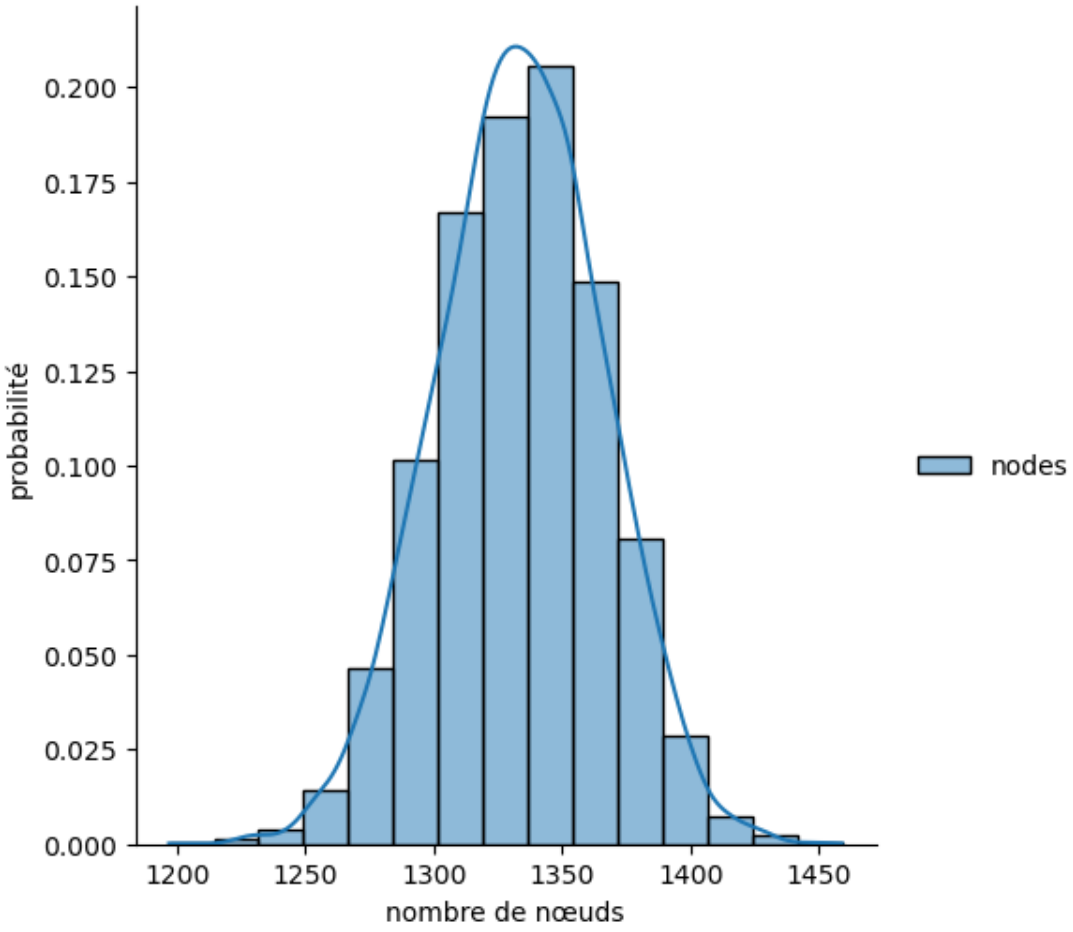

Figure 9 Histogramme de taille de ZDD

On peut constater que la taille des ZDDs générés a une distribution normale d'après le histogramme, de plus un test **Shapiro-Wilk** sur la donnée retourne une p-valeur = 0.1 qui confirme la distribution normale.

```
shapiro(data[:5000])
```

```
ShapiroResult(statistic=0.9994075894355774, pvalue=0.10719496756792068)
```

Figure 10 Test Shapiro-Wilk