Sorbonne Université

Devoir de Programmation

UE d'Ouverture

Danaël CARBONNEAU (28709878), Javaid Mohammad-Habib (21307723)

Diagrammes de décision binaires (*ZDD*).

Enseignants : Antoine Genitrini, Emmanuel Chailloux

Master STL, Semestre 1, septembre - janvier 2023 - 2024

Table des matières

1 Échauffement

Le code associé à cette partie se trouve dans le fichier *echauffement.ml*.

Nous représenterons dans la suite du projet les entiers précis par ce type : `type entier_precis = int64 list;;`. Nous avons également écrit pour les manipuler une primitive permettant l'ajout en fin, une permettant de récupérer la tête, et une permettant de récupérer sa suite.

Par la suite, nous avons pu écrire les fonction nous permettant de manipuler les grands entier sous plusieurs représentations (listes d'entiers 64 bits, listes de booléens). Grâce aux fonctions `Int64.unsigned_rem` `Int64.shitf_right_logical`, et `Int64.shift_left`, on traite bien notre entier comme s'il n'était pas signé, donc en utilisant ses 64 bits.

Concernant la génération aléatoire, depuis la version https://ocaml.org/releases/4.14.0, le module Random implémente une fonction `Random.bits64()` qui nous retourne 64 bits aléatoires représentés sur un entier 64 bits. Lorsqu'on teste `GenAlea`, on voit alors bien des nombres négatifs apparaître dans la décomposition (c'est à dire des nombres ayant leur bit de poids fort à 1, ce qui correspond à ce qu'on veut pour traiter les entiers 64 bits de la liste comme des bitmaps). Il est donc nécessaire d'avoir cette version d'installée.

2 Arbres de Décision

Le code associé à cette partie se trouve dans le fichier *Arbre_decision.ml*.

La structure est faite à l'aide d'un type Somme (et d'un alias de type pour la profondeur) :

```
type profondeur = int
type arbre_decision =
    | Feuille of boolean
    | Noeud of profondeur* arbre_decision * arbre_decision.
```

3 Compression de l'arbre de décision et ZDD

Le code associé à cette partie se trouve dans les fichiers *deja_vus.ml*, *compression.ml*, et *dot.ml*.

3.1 Choix d'implémentation : l'utilisation de foncteurs

L'algorithme de compression utilise une structure permettant de conserver une trace des grands entiers déjà vus. Étant donné que nous serons amenés à utiliser cette algorithme avec deux structures (une liste dans cette partie, une arborescence dans la suivante), nous avons décidé d'utiliser le mécanisme de **foncteurs** offertes par le langage OCaml.

En effet, du point de vue de l'algorithme, il nous faut une structure qui nous rend trois services :

- créer une structure vide
- insérer un couple (grand entier, arbre de décision) dans la structure
- déterminer si un grand entier est dans la structure (auquel cas, retourner l'arbre de décision associé, rien sinon).

On va donc définir un type de modules correspondant à cette interface : il s'agit du module `SetDejaVus`, dont la signature est donnée dans *deja_vus.ml*.

De là, on veut pouvoir utiliser n'importe quelle structure pour laquelle seraient définis ces trois services dans notre algorithme. Pour cela, on va écrire, dans *compression.ml* un foncteur `AlgoCompression`, c'est à dire un module paramétré par le type de modules `SetDejaVus`. À l'intérieur de ce foncteur, on peut désormais écrire notre fonction de compression, qui utilise le module passé en paramètres pour avoir :

- Le type de la structure utilisée
- Les opérations, qui sont nécessaires à l'algorithme, dont l'implémentation va dépendre de la structure

On arrive ainsi à factoriser notre code et séparer le déroulement de l'algorithme de la gestion sous-jacente, de notre ensemble d'arbres déjà vus.

3.2 Manipulation des arbres de compression

Dans l'algorithme de compression, nous voulons remplacer, lorsqu'une règle de compression est appliquée, le noeud courant N par un autre noeud vis à vis de son père. Étant donné que notre fonction reconstruit l'arbre compressé en fonction des appels récursifs, il nous suffit pour cela de retourner le bon nœud remplaçant le nœud N par celui qui lui correspond dans notre arbre compressé.

La représentation des valeurs, et notamment des types sommes, nous permet de ne pas avoir besoin d'utiliser de références : les valeurs en OCaml sont soit des entiers , soit des pointeurs vers un bloc sur le tas. Lorsqu'on ajoute un noeud en tête de notre ensemble de (grand entier, noeud) déjà visités, on créé sur le tas un bloc contenant un pointeur vers notre noeud, et un pointeur vers la suite de la liste. Lorsqu'on le récupère ailleurs dans le code, on ne récupère alors pas de copie mais bien un pointeur vers le nœud souhaité.

3.3 La fonction compressionParListe

3.3.1 Le module `SetList`

La première approche pour représenter notre ensemble de couples `(grand_entier, arbre_decision)` est de le faire par une liste. Il nous suffit pour ça d'écrire un module `SetList`, dont la signature est décrite par `SetDejaVu`. Sa structure comporte :

- Un type ens défini comme une liste de couples `(grand_entier, arbre_decision)`
- La valeur empty, définie comme étant une liste vide
- Une fonction d'insertion, qui se fait par un simple ajout en tête dans la liste.
- Une fonction de recherche, qui se fait en parcourant la liste en comparant les grands entiers présents avec celui passé en paramètres, et en rendant un `arbre_decision option` (`Some(arbre)` s'il est présent dans la liste, `None` sinon).

3.3.2 Définir la fonction grâce aux modules et au foncteur

Maintenant que nous avons un module décrivant une manière de gérer ces ensembles, nous pouvons instancier le foncteur AlgoCompression en lui passant en paramètres notre module `SetList` avec `module FL = AlgoCompression(SetList)`

On peut alors définir la fonction compressionParListe avec ce foncteur : `let compressionParListe = FL.compression`.

3.4 Faire des graphiques avec le langage dot

Dans le fichier *dot.ml*, nous avons écrit deux fonctions : une qui parcourt l'arbre de décision pour écrire dans un fichier ce qu'il faut pour chaque nœud, et une qui s'occupe de formater comme souhaité chaque noeud.

3.4.1 Formatage en dot

Dot nous permet de faire des graphes en décrivant les arrêtes, et potentiellement les nœuds, en les écrivant lignes après lignes. Il nous faut donc, pour transcrire facilement notre graphe, un identifiant unique par nœud (qui ne sera pas affiché, grâce aux labels). On peut ensuite, pour chaque nœud interne du graphe, écrire dans notre fichier la mise en forme correspondante :

```
Noeud(profondeur, fils_gauche, fils_droit) ->
     idNoeud [label = profondeur];
     idNoeud -> idFils_gauche [style=dotted];
     idNoeud -> idFils_droit [style=dotted];

| Feuille(boolean) -> idN [label = boolean];
```

3.4.2 Des identifiants uniques... avec un peut de magie

Afin d'avoir un **idN unique**, nous avons choisi d'utiliser la fonction `Obj.magic`. Il s'agit d'une fonction (*peu recommandée*) de la librairie standard OCaml : elle force un cast de type sur l'objet qu'elle manipule (soit un entier, soit un bloc, donc un pointeurs vers le tas), qui est dans tous les cas codé sur un entier (soit de taille 32bits, doit de taille 64), et retourne sa valeur "réelle" (au runtime).

Étant donné que nous appliquons cette fonctions à des arbres de décision, c'est à dire un type somme, c'est une manière de récupérer l'adresse de chaque block. Il s'agit donc d'un nombre **unique** à chaque nœud dans notre arbre, ce qui en fait un candidat parfait pour notre identifiant [1].

3.4.3 Résultats

En lançant le script *Tests_graphiques.sh*, nous avons alors pu obtenir un affichage graphique de l'arbre de l'énoncé avant (1) et après compression (2).

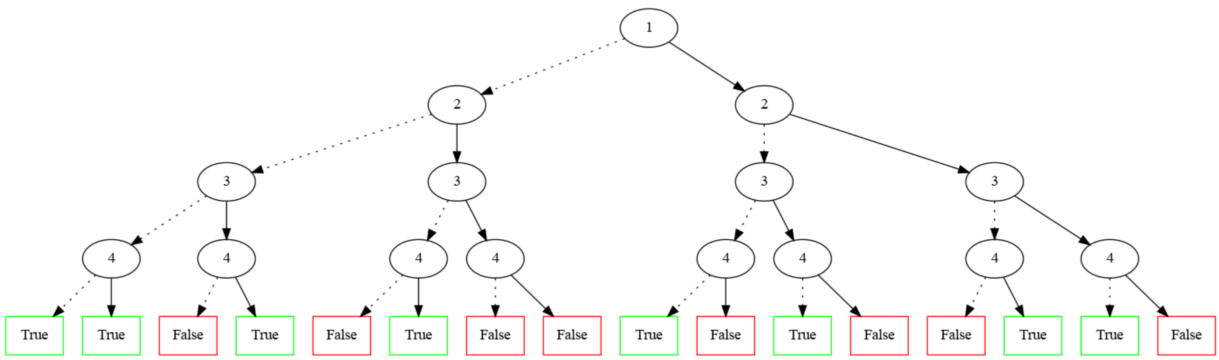


Figure 1: Arbre de décision issu de la table de vérité de taille 16 construite sur [25899]

1. Le système d'identifiant par pointeur nous permet également de vérifier que la compression s'est aussi faite en mémoire.

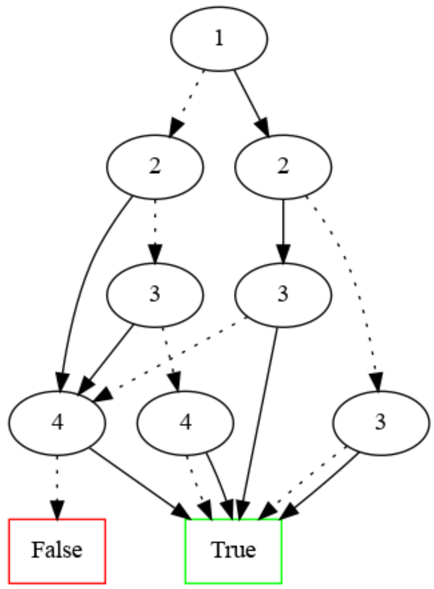


FIGURE 2: ZDD construit sur le grand entier [25899],
compressé avec CompressionParListe

L'algorithme de compression que nous avons écrit nous semble alors fonctionner conformément aux attentes.

4 Compression avec historique stocké dans une structure arborescente

Le code correspondant à la structure arborescente se trouve dans le module `SetTree`, présent dans le fichier *deja_vus.ml*. La définition de la fonction de compression dans *compression.ml*.

4.1 La fonction compressionParArbre

4.1.1 Le module `SetTree`

Le module `SetTree` répond à la signature décrite par `SetDejaVu`. Sa structure comporte :

- Un type ens défini comme un arbre dont les feuilles sont vides et les nœuds sont des `arbre_decision option`, c'est-à-dire qu'il peut soit y avoir un arbre de décision sur le nœud, soit rien.
- La valeur empty, définie comme étant une feuille.
- Une fonction d'insertion, qui se fait en parcourant les bits du grand entier du couple à insérer en allant dans le sous arbre gauche si le bit est à 0, à droite si le bit est à 1. Lorsqu'on a fini de parcourir les bits, on insère le pointeur dans le nœud courant [2].
- Une fonction de recherche, qui se fait en parcourant les bits du grand entier du couple en suivant le même aiguillage. Lorsqu'on a fini de regarder les bits du grand entier, on retourne le `arbre_decision option` correspondant (`None` s'il n'y a pas le pointeur).

2. Dans le cas, qui n'est pas interdit par notre structure, mais qui ne devrait pas arriver vue la manière dont on se sert de ces arborescences, où on insère un couple dont le grand entier est déjà dans l'arborescence, on remplace l'ancienne valeur du pointeur par la nouvelle.

4.1.2 Définir la fonction `compressionParArbre`

Maintenant que nous avons un module décrivant une manière de gérer ces ensembles avec des arbres, nous pouvons instancier le foncteur `AlgoCompression` en lui passant en paramètres notre module `SeTree`, on peut alors définir la fonction `compressionParArbre` avec ce foncteur :

```
module FT = AlgoCompression(SetTree)
let compressionParArbre = FT.compression
```

4.2 Résultats

Dans *dot.ml*, on ajoute alors une autre ligne de code appliquant la fonction dot à l'arbre compressé avec la fonction `compressionArbre`.

On obtient alors, toujours avec le script *Tests_graphiques.sh* une image de l'arbre compressé identique à celui de notre réalisé à l'aide de la compression par liste 3.

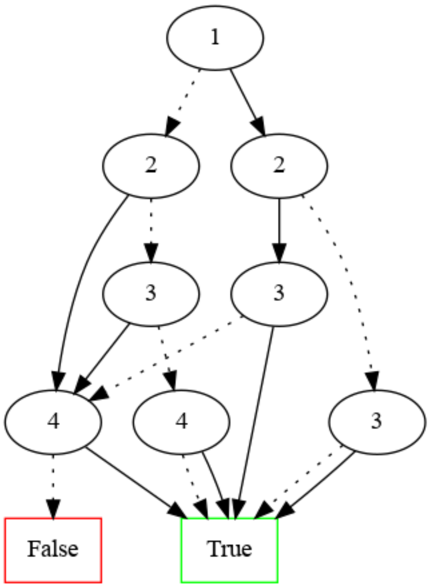


FIGURE 3: ZDD construit sur le grand entier [25899], compressé avec `compressionParArbre`

5 Analyse de complexité de notre fonction de compression

Pour analyser la complexité de l'algorithme, nous compterons cette dernière en nombre de comparaisons[3]. Nous la calculerons d'abord sur la taille de l'arbre. Il sera ensuite possible de passer de la taille de l'arbre à la taille de notre table de vérité par un simple calcul.

Notre fonction de compression prend la forme d'un algorithme de type **diviser pour mieux régner** : lorsqu'on a un arbre de taille n, on commence par résoudre le problème sur les deux enfants de sa racine, qui sont tout deux de taille majorée par $n/2$[4], puis on combine nos deux résultats. Analyser la complexité de notre algorithme nécessite alors d'estimer le coût de cette recombinaison.

3. Que ce soit d'égalité entre deux grands entiers, ou bien dans des structures d'alternatives sur les bits (`if booléen then [...]` équivaut à comparer le booléen à true et faire le saut correspondant).

4. C'est le cas avant compression car nous avons un arbre parfait par construction, mais puisqu'on recombine des arbres potentiellement compressés, ils peuvent avoir moins de noeuds

Nous analysons la complexité de la fonction suivante :

```
1  let compression (arbre : arbre_decision) : arbre_decision =
2      let rec aux (arbre : arbre_decision) (deja_vus : E.ens) : (arbre_decision * E.ens) =
3        let feuilles = liste_feuille arbre in
4        match arbre with
5        | Noeud(profond, sag, sad) ->
6            let nouveau_sag, n_deja_vus1 = aux sag deja_vus in
7            let nouveau_sad, n_deja_vus2 = aux sad n_deja_vus1 in
8              (*Regle de compression Z*)
9              if (deuxieme_moitie_false feuilles) then (nouveau_sag, n_deja_vus2)
10             else
11             (let grand_entier = (composition64 feuilles) in
12               match (E.mem grand_entier n_deja_vus2) with
13               | Some(pointeur) ->
14                     (*Regle de compression M*)
15                     (pointeur, n_deja_vus2)
16               | None ->
17                     let nouveau_noeud =   (Noeud (profond,nouveau_sag,nouveau_sad))
18                     in ( nouveau_noeud , (E.insert (grand_entier, nouveau_noeud) n_deja_vus2))
19             )
20       | Feuille(booleen) ->
21         (let grand_entier = (composition64 feuilles) in
22           match (E.mem grand_entier deja_vus) with
23           | Some(pointeur) ->
24             (*Regle de compression M *)
25             (pointeur, deja_vus)
26           | None -> ( arbre , (E.insert (grand_entier, arbre) deja_vus))
27         )
28     in let (g,_) = (aux arbre E.empty) in g
```

Dans le cas où notre arbre est sous la forme d'un Nœud :

- Deux appels récursifs (*l.6-7*) sur des sous arbres de taille au maximum $\frac{n}{2}$
- Test sur la liste de feuilles pour faire une sortie anticipée selon la règle Z (*l.9*)
  Complexité : $O(logn)$ [5]
- Calcul du grand entier correspondant au nœud courant (*l.11*)
  Complexité : $O(logn)$ [6]
- Recherche de ce grand entier dans l'ensemble (*l.12*)
  Complexité : $MemE(n)$, nous la nommons ainsi, il faudra la calculer selon la structure choisie
  - Retourner le nœud trouvé dans l'ensemble s'il y en a un selon la règle de compression M (*l.15*)
    Complexité : $O(1)$
  - Créer un nouveau nœud avec ceux récupérés par les appels récursifs et insérer ce nœud dans l'ensemble sinon (*l.17-18*)
    Complexité : $InsE(n)$, également à calculer

Dans le cas où notre arbre est sous la forme d'une Feuille :

- Calcul du grand entier correspondant au nœud courant (*l.21*)
  Complexité : $O(logn)$ **(idem *l.11***
- Recherche de ce grand entier dans l'ensemble (*l.22*) :
  Complexité : $MemE(n)$, (idem *l.12*)

5. La liste de booléens est de taille log(n), on la parcourt
6. La liste de booléens est de taille log(n), on la parcourt

- Retourner le nœud trouvé dans l'ensemble s'il y en a un selon la règle de compression M (*l.25*) **Complexité :** $O(1)$
- retourner l'arbre courant et l'insertion de ce dernier dans l'ensemble sinon (*l.26*) **Complexité :** $InsE(n)$, (idem *l.17-18*)

Notre pire cas est lorsque pour tous nos arbres visités, ils ne sont pas déjà dans l'ensemble, et qu'il faut donc les rajouter [7].

Étant donné que le pire des cas, pour tout n, est lorsque n n'est pas dans l'arbre, on peut majorer la complexité de notre algorithme par

$$T(n) = 2T(\frac{n}{2}) + 2log(n) + MemE(n) + InsE(n)$$

Pour approximer cette borne supérieure, nous allons devoirs étudier plus précisément les ordres de grandeur de $MemE(n)$ et $InsE(n)$, qui diffèrent selon l'implémentation.

On rappelle également le théorème maître, qui nous permet de calculer la complexité d'un algorithme type diviser pour mieux régner :

Soient a $\geq$ 1 et b $>$ 1 deux constantes, soient f une fonction à valeurs dans $\mathbb{R}$ et une fonction de $\mathbb{N}*$ dans $\mathbb{R}^+$ vérifiant pour tout n suffisamment grand l'encadrement suivant :

$$aT(\lfloor\frac{n}{b}\rfloor) + f(n) \leq T(n) \leq aT(\lceil\frac{n}{b}\rceil) + f(n) \tag{1}$$

Alors T peut être bornée asymptotiquement comme suit :

1. Si $f(n) = O(n^{(log_b a)-\epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \Theta(n^{log_b a})$.
2. Si $f(n) = \Theta(n^{log_b a})$, alors $T(n) = \Theta(n^{log_b a} log(n))$.
3. Si $f(n) = \Omega(n^{(log_b a)+\epsilon})$ pour une certaine constante $\epsilon > 0$, et si on a pour $c > 1$ $af(\frac{n}{b}) < cf(n)$, alors $T(n) = \Theta(f(n))$.

5.1 Complexité de la fonction `compresionParListe`

Dans notre structure de liste :

- La complexité d'une insertion est en $O(1)$, étant donné que nous faisons un ajout en tête
- La complexité d'une recherche est en $O(L)$, où L est la taille de la liste.

Or, ici, on fait la supposition d'un pire cas où tous les sous arbres sont différents (notre pire cas théorique), pour un arbre de taille n, L est de l'ordre de grandeur de n. L'insertion est donc en $O(n)$. On peut même la considérer en $\Theta(n)$ dans la mesure où on a fait la supposition que l'arbre n'est pas dans la liste pour avoir le pire cas possible.

On en déduit

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 2log(n) + MemE(n) + InsE(n) \\ &= 2T(\frac{n}{2}) + O(logn) + O(1) + \Theta(n) \\ &= 2T(\frac{n}{2}) + \Theta(n) \end{aligned}$$

7. On pourrait trouver un pire cas plus fin, en réfléchissant sur la combinaison des feuilles étant donné qu'elles ne peuvent prendre que deux valeurs...

Par la règle 2 du théorème maître 2, avec $a = 2$, $b = 2$ et $log_b a = 1$, on obtient

$$T(n) = \Theta(nlog(n))$$

Donc `compressionParListe (g)` $= O(nlog(n))$, où g est un graphe de taille n.

5.2 Complexité de la fonction `compresionParArbre`

Dans notre structure de liste :

- La complexité d'une insertion est en $\Theta(L)$, L étant la longueur de la liste de bits, car on va parcourir l'arbre jusqu'à épuiser ses éléments puis insérer l'arbre
- La complexité d'une recherche est en $O(L)$, L étant la longueur de la liste de bits, car on va parcourir l'arbre jusqu'à épuiser ses éléments puis retourner le nœud courant, ou bien s'arrêter avant si on trouve une feuille.

Or, ici, la longueur de la liste de bits correspond au nombre de feuilles de notre arbre, qui est majoré par $log_2(n)$.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 2log(n) + MemE(n) + InsE(n) \\ &= 2T(\frac{n}{2}) + O(logn) + O(log(n)) + \Theta(log(n)) \\ &= 2T(\frac{n}{2}) + O(logn) \end{aligned}$$

Par la règle 1 du théorème maître 1, avec $a = 2$, $b = 2$ et $log_b a = 1$, et le fait que $log(n) = O(n)$ [8] on obtient

$$T(n) = \Theta(n^{log_2 2}) = \Theta(n)$$

Donc `compressionParArbre (g)` $= \theta(n)$, où g est un graphe de taille n.

8. les logarithmes sont majorés par les fonctions linéaires pour de grandes valeurs

6 Étude expérimentale

Dans cette section, nous allons présenter différentes statistiques obtenues expérimentalement concernant notre algorithme.

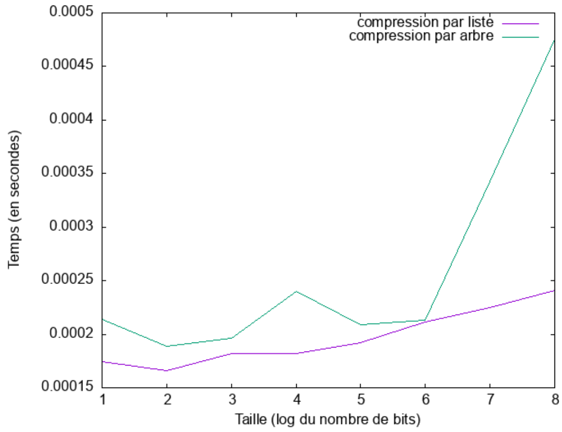
Le code correspondant à ces expérimentations se trouve dans *tests.ml*. Pour les calculer, on utilise le script *Statistiques.sh*. Afin d'ensuite produire les graphiques, il suffit de lancer gnuplot depuis le terminal, et entrer la commande `load "graphiques.gnu"`.

6.1 Comparaison de nos deux algorithmes

6.1.1 Comparaison du temps que prend la compression

Un premier test à effectuer expérimentalement, et d'observer le temps d'exécution de nos deux codes, afin de comparer les deux structures, mais aussi leurs complexités théoriques avec le temps effectivement pris. (4)

FIGURE 4: Comparaison temporelle entre `compressionParListe` et `compressionParArbre`



On voit alors que la compression par arbre est en réalité plus coûteuse la plupart du temps, et l'est significativement plus pour de petites valeurs : cela peut s'expliquer de plusieurs manières :

- L'insertion n'est pas en temps constant pour l'arbre, on fait beaucoup plus d'insertions (relativement à la taille de la table) pour de petites tables de vérité, dans la mesure où au fur et à mesure qu'elle augmente, la probabilité que deux sous arbres aient la même table de vérité augmente. Par rapport à une liste, on peut perdre à ce niveau
- Le temps de recherche dans une liste est bornée par sa longueur, il est donc logique qu'elle prenne moins de temps quand on a peu d'éléments. Là où celui d'un arbre dépend de la longueur du plus grand préfixe commun entre l'élément recherché et ceux dans la table, le pire étant la longueur de la table.

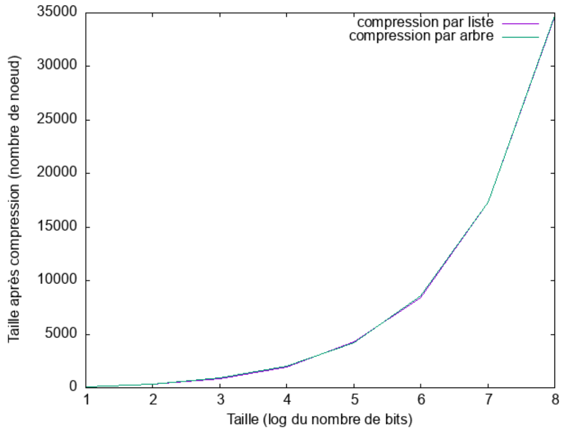
Ces grandes variations font que les pires cas estimés pour nos deux structures ne sont pas souvent atteints, voire pas. La borne supérieure ne permet pas de rendre compte des cas survenant dans la réalité, pouvant parfois significativement baisser le temps de calcul.

On peut également noter qu'une structure où l'insertion se fait en un temps qui n'est pas constant ne semble pas être un bon choix par rapport à notre algorithme, et qu'il semble préférable de privilégier une structure où l'insertion est rapide, quitte à avoir une recherche d'une complexité au pire cas supérieure, surtout lorsque ce pire cas est rarement atteint.

6.1.2 Estimation de la taille moyenne de l'arbre compressé

Nous avons choisi de représenter les courbes issues des algorithmes, mais elles sont sensiblement identiques (les algorithmes aboutissant nécessairement au même arbre), cela nous permettant de corroborer l'équivalence de nos deux algorithmes. Ce second test nous permet d'observer l'évolution de la taille de l'arbre compressé en fonction de la taille de l'arbre non compressé. Cette croissance semble exponentielle, ce qui est conforme à nos attentes. (5)

FIGURE 5: Évolution de la taille de l'arbre compressé en fonction de la taille de l'arbre non compressé

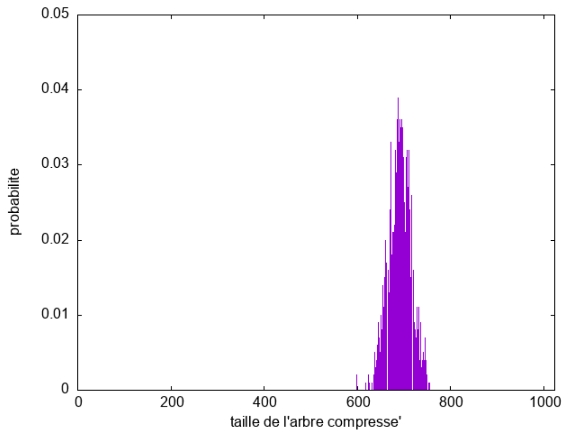


6.2 Approximation de la distribution de probabilité des tailles des ZDD

Pour approximer la distribution de la probabilité des tailles des ZDD, nous avons fixé la taille n de la table de vérité à 512 bits, ce qui va nous donner une plage de valeurs possibles entre 0 et 1023[9].

La distribution de probabilité des tailles des ZDD est alors la probabilité, pour chaque taille possible, qu'il existe un ZDD de cette taille correspondant à une table de vérité de la taille que nous avons fixé. La distribution prend alors une forme gaussienne (6), ce qui est conforme aux résultats attendus.

FIGURE 6: Distribution des probabilités des tailles des ZDD pour une table de vérité de 512 bits



9. cas d'un arbre parfait, où pour un étage ayant 2^n feuilles, on trouve 2^{n-1} à l'étage supérieur, la somme des étages nous donne alors bien $2^n - 1$ nœuds dans l'arbre.