Sorbonne Université - Master STL

MU4IN511 - Ouverture

Année 2023-2024

Projet : Génération des ZDD

Alex XU & Jean-Marc ZHUANG

SOMMAIRE

1. Echauffement

Question 1.1

Ici, nous avons créer une structure de donnée nommée *ListInt64* qui est une référence vers *'a list*, dans cette structure nous avons plusieurs primitives:

create() qui créer une réference vers une liste vide
insertHead(x, list) qui ajoute en tête de liste
insertTail(x, list) qui ajoute en queue
removeHead(list) qui supprime la tête et la renvoie
length(list) qui renvoie la taille de la liste en utilisant *List.length*

Question 1.2

La fonction *decomposition* prend en entrée un grand entier *l* et le décompose en une liste de bits, dont le bit de poids faible se trouve à gauche. Pour ce faire, elle utilise deux fonctions auxiliaires : *loop* et *loop2*.

Dans le sujet, *0L* représente 2^{64}, on a donc implémenté la fonction *bit64* qui renvoie une liste contenant uniquement des *false*, puisque 2^{64} s'écrit avec 64 bits à 0 et le bit le plus fort à 1.

La fonction *loop* prend en argument un grand entier *x* et une liste *list*. Elle décompose *x* en bits en utilisant une division par 2 et un modulo qui sont tous deux des opérations unsigned, afin de traiter les entrées comme des entiers non signés.

Si x est égal à 0, elle renvoie la liste.

Sinon, elle vérifie si le reste de la division de *x* par 2 est 1 (ce qui signifie que son bit de poids faible est 1). Si c'est le cas, elle ajoute *true* à la liste, sinon elle ajoute *false*.

Ensuite, elle appelle récursivement *loop* avec *x* divisé par 2 et la nouvelle liste.

La fonction *loop2* prend en argument le grand entier *l* et l'accumulateur *list*. Elle parcourt la liste *l* et pour chaque élément, si l'élément est égal à 0, elle ajoute à la liste une liste de 64 bits. Sinon, elle appelle loop avec l'élément et la liste.

Enfin, *decomposition* appelle *loop2* avec le grand entier *l* et l'accumulateur vide.

Question 1.3

La fonction *completion* permet de tronquer la liste décomposée si *n* est inférieur à la taille de la liste, sinon elle "complète" la liste en ajoutant des false aux bits les plus forts pour changer la taille de la représentation sans changer sa valeur. Pour cela nous avons 2 primitives

- *ajoutFalse(list, n, accu)* qui ajoute *n* false dans *accu*, et dès que n est nul on concatène *list et accu* et on renvoie le couple (list, [])
- *suppElem(list, n, accu)* qui tronque list pour que la taille de la liste soit égale à n. Donc à la sortie de la récursivité, on renvoie un couple de *(accu, list)* où accu est la liste tronquée, et list le reste sans la liste tronquée car nous en avons encore besoin dans la fonction package (Question 1.4).

Ainsi pour la fonction *completion* si la taille de la liste est supérieur à n, on fait appel à *suppElem* sinon on fait appel à *ajoutFalse*

Question 1.4
La fonction *composition* passe de la représentation binaire à sa représentation en base 10, c'est-à-dire un grand entier. Chaque élément du grand entier est sur 64 bits, donc nous avons créé une fonction *package(list)* qui va créer plusieurs paquets de 64 bits du grand entier si la taille de la liste est supérieur à 64. Ainsi, package renvoie une matrice de boolean, la matrice représente le grand entier, et les sous tableaux représentent les éléments.
Visuellement [0L, 1L] est la matrice, 0L et 1L sont les éléments.
LoopExt parcours les sous tableaux et *LoopInt* créer les éléments du grand entier. Pour créer l'élément, on fait un décalage à gauche de l'accumulateur puis on fait un OR logique entre l'accumulateur et le boolean se trouvant dans la matrice.

Question 1.5
La fonction *table(x, n)* crée la table de vérité de l'entier *x*, on applique donc *decomposition* sur *x* pour finir on applique *completion* sur la représentation binaire de x, on peut remarquer que *table* renvoie un couple car il fait appel à suppElem qui renvoie *(tronqué, reste)* où *tronqué* est la liste tronqué et *reste* le reste de la liste.

Question 1.6
La fonction *gen_alea*, prend en entrée une valeur *n* et génère un grand entier aléatoire de n bits au maximum.
Dans cette fonction nous implémentons une fonction auxiliaire loop qui prend en argument le grand entier aléatoire de n bits à renvoyer *acc* ainsi que le nombre de bits restant *n*.
On construit notre acc en insérant en tête tous les entiers aléatoire de 64 bits, tant que le nombre de bit restant est supérieur ou égale à 64, puis le dernier entier aléatoire de $n-\ell\times64$.
Enfin, lorsqu'il ne reste aucun bit, on renvoie List.rev du grand entier de n bits, car on souhaite que l'entier de $n-\ell\times64$ soit en queue.
Admettons que nous avons le cas où n = 63, alors si on fait un décalage à gauche de 1L 63 fois, nous obtenons un entier négatif donc si on applique la fonction Random.int64 sur un nombre négatif, cela lève une exception, pour palier à ce problème nous avons tout simplement utilisé Int64.max_int lorsque n = 63 car cette valeur correspond à l'entier le plus grand avec 64 bits sinon on décale 1L n fois.

2. Arbre de décision

Question 2.7

```
type btree =
 | Leaf of bool //true ou false
 | Node of int * btree * btree //profondeur * enfant_gauche * enfant_droit
```

Question 2.8
La fonction *cons_arbre*, prend en entrée une table de vérité *table* et construit l'arbre de décision associé à la table de vérité.
On utilisera une fonction auxiliaire *construction* afin de construire notre arbre à partir d'une profondeur 0 qu'on incrémente à chaque appel, il prend en deuxième paramètre une table de vérité.
Si la table est vide on renverra une exception *"Table de vérité vide"*.
S'il n'y a qu'un élément, on renvoie la feuille contenant le booléen.
Sinon on construit notre arbre en appelant récursivement *construction,* en incrémentant la profondeur pour l'indice et divisant la table de vérité en deux à l'aide de la fonction *diviser_liste* qui prend en argument la table de vérité à couper en deux.
Notre fonction *diviser_liste,* possède aussi une fonction auxiliaire, qui permet de décrémenter la taille de la première liste au fur et à mesure que l'on construit celle-ci. On retourne un couple de liste, dont la première de taille n/2 et la deuxième est la table restante.

Question 2.9
La fonction *liste_feuilles*, prend en entrée un nœud *arbre* et construit la liste des étiquettes des feuilles du sous-arbre enraciné en N. On construit notre liste en parcourant notre arbre de manière préfixe, ça permettra d'avoir une liste des feuilles gauches à droites.

3. Compression de l'arbre de décision et ZDD

Question 3.10

```
type dejaVus = int64 list * btree ref;;
type listeDejaVus = dejaVus list;;
```

Question 3.11
Pour créer la fonction compressionParListe, nous avons au préalable créé 3 primitives:

- *findSeen* qui va rechercher si l'élément n (composition d'une liste de boolean) à été vu, si en parcourant toute la liste, la fonction ne trouve pas n alors on renvoie None, sinon on renvoie le pointeur vers le noeud
- *onlyFalse renvoie vrai si le tableau contient uniquement des false sinon renvoie faux*
- *regleM* prend un arbre, le tableau des int64 vu, et la représentation binaire de l'entier comme argument. Tout d'abord on applique composition à la représentation binaire pour renvoyer un int64 qui sera maintenant appelé n, on va ensuite appliquer *findSeen sur n. Si la fonction renvoie node* alors tree sera remplacé par node sinon on touche pas à l'arbre et on ajoute le couple *(n, tree),* tree étant le nœud où on se trouve actuellement.

La fonction compressionParListe possède un arbre *(tree)* et une liste de déjà vu *(seen)* comme argument, si *tree* est une feuille alors on applique la règle M, sinon si c'est un nœud, on va appliquer *onlyFalse* au fils droit de *tree*.

Si le fils droit contient que des false alors on fait une récursion que sur le fils gauche et ensuite, *tree* est remplacé par le fils gauche de *tree (règle Z).*
Sinon on fait un parcours suffix et on applique la règle M.
Nous faisons la compression par copie.

Question 3.12

Pour créer le fichier dot, nous avons écrit 3 fonctions *nodeGraph, edgeGraph, et graph.*
La hashmap que j'utilise dans *nodeGraph et edgeGraph* possède des couples de *(tree, id),* tel que *tree* est le nœud déjà parcouru et *id* est l'identifiant qu'on a attribué à *tree.*

Tout d'abord *nodeGraph* qui prend un arbre de décision et un buffer comme argument, cette fonction va déterminer l'identifiant de chaque nœud en fonction du compteur. Mais aussi son label en fonction de sa profondeur si le nœud courant est un nœud, et si c'est une feuille, le label sera True ou False. Pour implémenter cela, nous avons une hashmap qui sert à associer un nœud avec son identifiant, de plus avec cette représentation, on peut déterminer que l'élément a déjà été parcouru s'il est présent dans la hashmap. Plus précisément, si le nœud courant est une feuille, on va tester sa présence dans la hashmap, s'il n'est pas présent, cela veut dire que la méthode *find* de *Hashtbl* renvoie l'exception Not_Found, on l'ajoute et on écrit la chaîne correspondant à une feuille dans le buffer. Par contre si le nœud courant est un nœud on va faire le même procédé qu'une feuille mais on écrit la chaîne correspondant à un nœud dans le buffer. De plus on va devoir faire la récursion, si le fils gauche n'est pas présent dans la hashmap, cela veut dire qu'il n'a pas été parcouru donc on incrémente le compteur et on fait la récursion sur le fils gauche. Idem pour le fils droit.

Ensuite *edgeGraph* qui prend un arbre et un buffer comme argument, cette fonction va créer les arcs entre les nœuds, et leurs 2 fils, et les ajouter dans le buffer.
Pour implémenter cela, nous avons de nouveau la hashmap et le compteur qui utilisent le même principe que dans *nodeGraph.*
Si le nœud courant est une feuille on ne fait rien puisqu'une feuille n'a pas d'arc sortant.
Si c'est un noeud on va tester l'appartenance du fils gauche au hashmap, si c'est le cas, pas besoin de récursion car on l'a déjà parcouru et on ajoute l'arc entre le noeud courant et son fils gauche, sachant que l'identifiant du fils gauche a été obtenu grâce au hashmap. Par contre si le noeud n'est pas retrouvé, on incrémente le compteur, on ajoute le fils gauche à la hashmap, on ajoute l'arc dans le buffer et on fait la récursion sur le fils gauche, même procédé pour le fils droit.

Pour finir la fonction *graph* prend un grand entier ***l*** et un entier ***n*** comme argument.
Cette fonction va créer la table de vérité du grand entier sur ***n*** bits, ensuite on va construire l'arbre correspondant à la table avec *cons_arbre*, appliquer la *compressionParListe* sur l'arbre. Ensuite sur cette arbre compressé on fait appel à nodeGraph et edgeGraph pour écrire le fichier dot dans le buffer. Pour finir on ouvre un fichier dot, on écrit le buffer dans le fichier ouvert, et on le ferme.

Question 3.13

À l'intérieur de la fonction graph, on a créé un second buffer qui va construire l'arbre de décision qui n'est pas compressé, donc on a uniquement appliquer *nodeGraph et edgeGraph* sur l'arbre construit à partir de la table de vérité. Ensuite on a ajouté ce buffer dans le fichier "Figure 1.dot".
Comme on peut le constater sur le fichier png, ce n'est pas l'arbre de la Figure 1 du sujet, mais en y regardant de plus près on peut s'apercevoir que c'est le même, mais avec un début de compression. Pour expliquer cela, nos fonctions nodeGraph et edgeGraph ajoutent dans la hashmap les nœuds déjà visité, donc on ne fait pas de test d'égalité référentielle. C'est pourquoi si l'algo parcourt l'arbre et qu'il trouve plusieurs feuilles "True", il va considérer que ce sont les mêmes, et donc faire une "compression". Essentiellement c'est le même graph que la Figure 1 du sujet.

Question 3.14
Figure 2.png, vous pouvez tester avec d'autres grand entier, à la ligne 288 du fichier "Projet.ml"

4. Compression avec historique stocké dans une structure arborescente

Question 4.15

```
(*4.15*)
type arbreDejaVus =
| Leaf_
| Node_ of btree option * (arbreDejaVus ref) * (arbreDejaVus ref)
```

Question 4.16
En passant de la liste ListeDejaVus à l'arbre de recherche ArbreDejaVus, nous avons créé 1 nouvelle fonction nommée *insertionArbre*, et adapté 2 fonctions déjà existantes, findSeen et regleM. Ainsi après l'adaptation ces fonctions sont désormais appelées findSeenArbre et regleM_Arbre.

Nous allons d'abord décrire la fonction d'insertion dans un arbre de recherche, cette fonction possède 3 arguments:

arbre est l'arbre de recherche des déjà vus
bits est la liste de feuille du noeud à insérer
node est le noeud qui a été parcouru par l'algo, et donc celui qu'on veut insérer

Pour le cas terminal, si dans l'arbre c'est une feuille et que la liste des feuilles est vide, on peut insérer *node* dans l'arbre.
Si l'arbre est une feuille, et qu'à la tête de la liste c'est le boolean true, on crée un nouveau nœud qui ne sera pas étiqueté, une feuille gauche, et la récursion sur une feuille Leaf_ et la suite de la liste. Idem si à la tête de la liste se trouve le boolean false mais la récursion se fait sur le fils gauche contrairement au cas true.
Ensuite, si nous sommes dans un noeud et que la liste est vide, on peut ajouter directement *node* à l'arbre, car le noeud sera forcément pas étiqueté, si on tombe sur un noeud déjà étiqueté, c'est qu'ils ont la même représentation binaire.

Pour finir les 2 derniers cas servent à parcourir l'arbre de recherche, si à la tête de la liste se trouve le boolean false, on fait la récursion sur le fils gauche, et inversement pour true.

La fonction findSeenArbre fait une recherche dans l'arbre déjà vu, s'il tombe sur une feuille, cela veut dire que l'élément n'est pas présent donc on renvoie None, si on tombe sur un pointeur, c'est parfait, on le renvoie, et sinon on fait le parcours habituel en fonction de ce qui se trouve à la tête de la liste des feuilles

La fonction regleM_Arbre possède 3 arguments:

- *tree* est le noeud actuel dans le parcours de l'algo
- *seenArbre* est l'arbre déjà vu
- *liste* est la liste de feuille de l'argument *tree*

cette fonction va faire appel à findSeenArbre qui va renvoyer soit None soit le pointeur vers le noeud du graphe, si c'est None, on fait appel à insertionArbre, et on renvoie tree, puisque notre compression se fait par copie, sinon c'est Some pointeur, si le pointeur renvoie None il y a une erreur, sinon on renvoie le noeud du graphe qui a déjà été parcouru.

Question 4.17
Essentiellement notre compressionParArbre ne change pas de notre compressionParListe car l'algorithme ne change pas mais juste la représentation des nœuds déjà parcouru change.
Donc dans compressionParArbre on fait appel à regleM_Arbre et c'est tout.
De plus on à changé le nom listseen par seenArbre pour qu'on comprenne bien qu'on utilise maintenant un arbre de recherche et non une liste.

Question 4.18
"Figure 2 Arbre.png, vous pouvez tester avec d'autres grand entier, à la ligne 363 du fichier "Projet.ml"

5. Analyse de complexité

Question 5.19
CompressionParListe :
Complexité temporelle :
Le calcul de la liste de feuilles se fait en O(log(n)).
L'application de la règle M se fait en O(m), car l'insertion en tête de liste se fait en temps constant O(1), avec m la taille de liste déjà vu de la dernière itération, dont on peut majorer toutes les opérations de recherche par m.
Dans notre compression par liste, on applique la règle M sur chaque nœud, ce qui nous donne une complexité en O(nm).
On calcule la liste des feuilles de chaque noeud en O(nlog(n)) en moyenne
Ainsi, on a O(nlog(n) + nm) ce qui nous donne une complexité en O(nm) car m est supérieur à log(n).

Complexité mémoire :
La complexité mémoire variera dépendamment de la construction de la liste déjà vus et de l'arbre déjà vus. On va donc s'intéresser à la complexité de la construction des deux structures.
Étant donnée l'arbre binaire ayant n nœud, la taille de la liste déjà vus sera majoré par le nombre de nœuds de l'arbre, il y a au plus n insertions, donc la complexité mémoire de la liste est de O(n).

CompressionParArbre :
Complexité temporelle :
Le calcul de la liste de feuilles se fait en O(log(n)).
L'application de la règle M se fait en O(h) (h la hauteur de l'arbre déjà vu, h = log(m), m le nombre de noeud de l'arbre), car la recherche et l'insertion d'un élément de l'arbre s'effectue en O(h), puisqu'on utilise la liste des feuilles pour rechercher et insérer.
Dans notre compression par arbre, on applique la règle M sur chaque nœud, ce qui nous donne une complexité en O(nlog(m)), avec n le nombre de nœuds de l'arbre compressé.
On calcule la liste des feuilles de chaque nœud, donc on a O(nlog(n)) en moyenne.
Ainsi notre complexité est en O(2nlog(n)) et donc en O(nlog(n)).

Complexité mémoire :
Étant donnée l'arbre binaire ayant n nœuds, l'arbre déjà vus sera formé de la liste de feuilles de la racine, qui est de taille n/2, ainsi que de ses enfants chacun de taille n/4, puis n/8 et récursivement on a sur chaque noeud, une liste feuille de taille $n/2^k$ (avec k la profondeur du noeud + 1).
On en déduit donc que la complexité de l'arbre déjà vus serait en $O(2^{\log(n)})$, donc en O(n).

6. Etude expérimentale

Question 6.20
Ici nous avons fait le choix de tester uniquement le taux de compression et le temps d'exécution.
Tout d'abord pour le taux de compression nous avons créé une fonction nommée nodeCount qui va renvoyer le nombre de nœuds dans l'arbre compressé. Pour implémenter cela, on incrémente le compteur à chaque fois qu'un nœud est étiqueté.
Nous avons aussi créé une fonction nommée treeHeight qui calcule la hauteur. Puisque l'arbre avant compression est un arbre complet, on peut juste calculer sa hauteur et renvoyer sa taille qui est 2^{h+1} -1.
Sachant que le taux de compression est le % de noeud compressé nous avons d'abord calculer la taille de l'arbre avant compression, explicité plus haut, ainsi que la taille de l'arbre après compression avec le nombre d'élément dans la liste de déjà vu pour la compression par liste, et avec nodeCount pour la compression par arbre. Ensuite nous avons fait le rapport entre le nombre de nœuds dans l'arbre après compression et l'arbre avant compression, ainsi on obtient une valeur inférieure à 1.
Pour calculer le temps d'exécution, nous avons utilisé Sys.time() (nommé t) avant la compression et à la fin de la fonction on fait la différence entre t et Sys.time(). C'est ce qui se passe dans nos fonctions

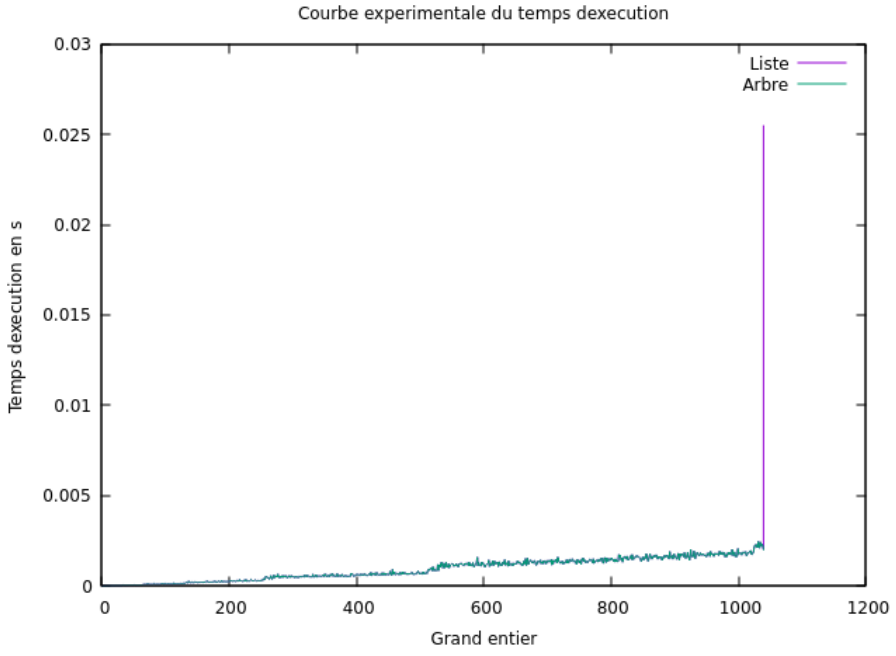
experimentalCurvesTree, et experimentalCurvesList. J'ai oublié de mentionner que ces 2 fonctions renvoient un couple (taux de compression, temps d'exécution).

Pour générer les courbes expérimentaux nous avons 2 buffers qui vont contenir:

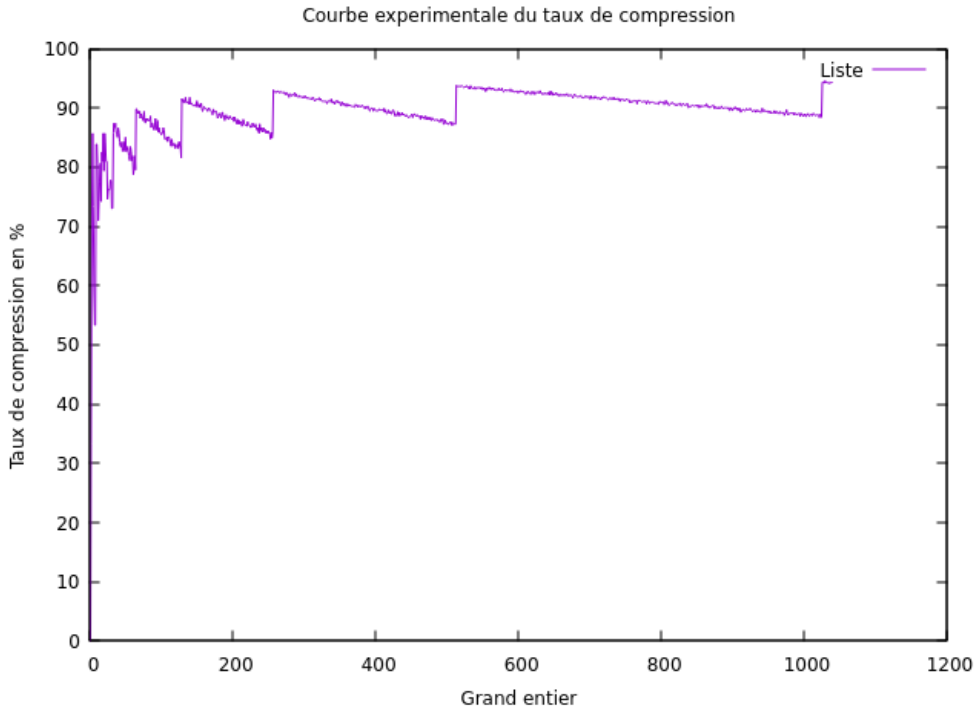
- un entier n qui est la taille de l'entier généré par (gen_alea n)
- le taux de compression qui se calcule par 1 - nombre de noeuds après compression / nombre de noeuds avant compression
- le temps d'exécution.

Les buffers qui contiennent les données sur le taux de compression et le temps d'exécution sont mis dans un fichier txt, et avec ces données nous avons pu tracer les graphiques.

Pour la courbe du temps d'exécution on peut constater que globalement la compression par liste et par arbre prennent autant l'un que l'autre pour s'exécuter mais à des valeurs très grandes la disparité entre les 2 méthodes se verra beaucoup plus, comme on le constate sur le graphique. Cela se traduit par le fait que au maximum l'arbre devra parcourir n/2 pour atteindre l'étiquette de la racine mais c'est un cas qui n'arrivera jamais car après avoir traité la racine, l'algorithme est fini. Contrairement à l'arbre, la liste de déjà est parcouru en entière à chaque fois qu'un nœud n'est pas présent dedans, donc même si l'insertion se fait en O(1) la recherche fait que cette méthode est moins efficace que l'arbre.



Pour la courbe du taux de compression, on peut constater que 99% des arbres ont été réduit de moitié après la compression, cela se traduit par la probabilité d'appliquer la règle Z ou la règle M qui augmente fortement plus l'arbre est grand. Ainsi on aura plus de chance de trouver des doublons (règle M) ou de se débarrasser des nœuds qui ont des fils droit composés uniquement de false dans leur liste de feuilles (règle M). On peut aussi apercevoir qu'à chaque fois que n est une puissance de 2 le taux de compression est plus élevé car dans mon implémentation à chaque fois que n > nombre de feuilles alors j'augmente la hauteur de l'arbre de 1, et donc le nombre de feuilles est multiplié par 2, et donc la règle Z s'appliquera pour les feuilles de remplissage (feuille se trouvant à droite de l'arbre et ayant la valeur false).



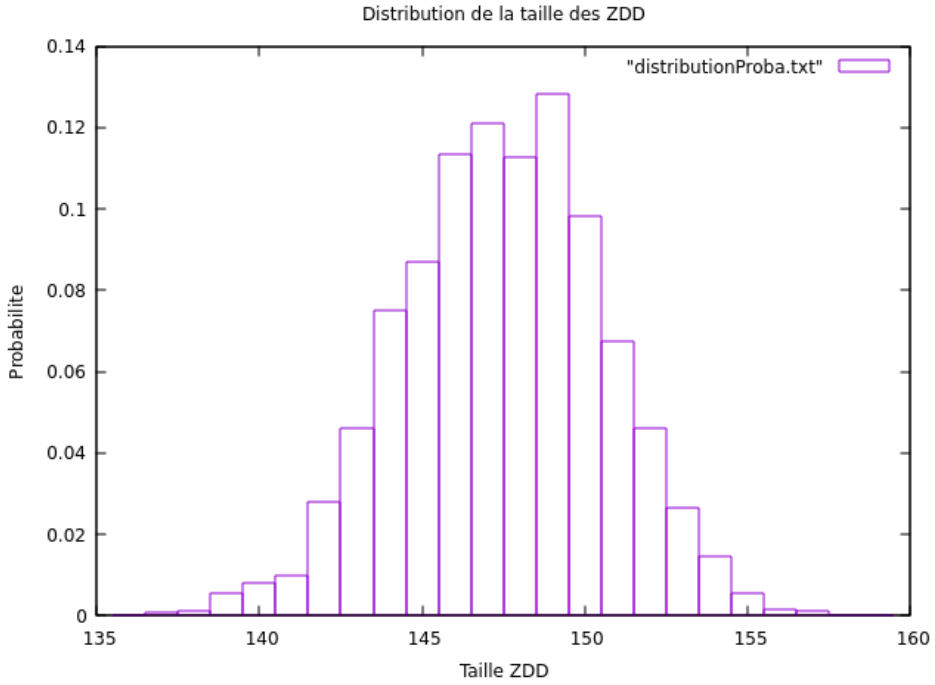
Question 6.21

Pour calculer une approximation de la distribution des probabilités de la taille des ZDD, nous écrit dans un fichier texte une liste de couple (tailleZDD, fréquence). À partir de ces données nous générons un histogramme représentant la distribution des probabilités de la taille des ZDD.
Tout d'abord nous avons créé plusieurs fonctions nous permettant de construire la liste de couple de (tailleZDD, nombre d'occurrence). Pour cela nous avons plusieurs fonctions:

- nb_Feuilles qui prend en argument un entier **n** (le nombre de bits de l'entier généré), cet fonction renvoie la puissance de 2 qui est supérieur à n, on en aura besoin pour la fonction table
- rechercheListe qui prend en argument une liste de couple (tailleZDD, nombre d'occurence) et un entier x (taille du ZDD). On recherche dans la liste si x un des couples contient x, on incrémente son nombre d'occurrence, sinon on ajoute un nouveau couple avec (x, 1)

- distributionOcc prend en argument un entier n (nombre de bits du grand entier généré), nbIteration et une référence vers une liste de couple (tailleZDD, nombre d'occurrence). À chaque itération on construit notre table de vérité, en générant notre grand entier avec gen_alea avec le nombre de bit en entrée, ainsi qu'une puissance de 2 supérieur à n avec nb_Feuilles (ça nous permettra de représenter le grand entier dans l'arbre). On construit l'arbre compressé à partir de la table de vérité. On compte ensuite, le nombre de nœuds de l'arbre compressé. On appelle la fonction recherche_liste décrite ci dessus, avec la liste des occurences donnée en entrée ainsi que la taille de l'arbre courant
- distributionProba qui prend la liste des occurrences complétées par la fonction distributionOcc, le nbIteration ainsi que la liste des probabilités que l'on remplira à chaque itérations.
  À chaque itérations, on calcule la probabilité de chaque élément = $\frac{nombre\ d'occurrences\ de\ l'élément}{nombre\ total\ d'occurrences}$, et on insère un couple (taille de ZDD, probabilité) dans la liste de probabilité à remplir

On inscrit notre liste des probabilités dans un fichier txt avec n = 300 et 3000 échantillons (arbres générés), que l'on utilisera pour générer notre histogramme suivant :



Comme on peut le constater la répartition de la taille des ZDD forme une gausienne, avec la moyenne qui se situe à 147, et l'écart-type est de 3, où au moins la moitié des valeurs sont concentrées entre 144 et 150.