SORBONNE UNIVERSITÉ

UE D'OUVERTURE

Rapport de

Projet ZDD

Enseignants:

Emmanuel CHAILLOUX

Antoine GENITRINI

Ariane ZHANG

Xue YANG

M1 INFO-STL 2023 - 2024

1 Présentation

Le projet porte sur la génération de ZDD (Zero-Suppressed Binary Decision Diagram), un concept introduit par Knuth et originellement défini par Minato. L'objectif principal est de comparer les temps de génération des ZDD obtenus par compression arborescente en fonction des structures de données utilisées.

1.1 Échauffement

Le module Int64 est recommandé pour manipuler des entiers signés sur 64 bits exactement. Lorsqu'on utilise tous les 64 bits, il est nécessaire de manipuler également le bit de signe.

Question 1.1 Représentation des grands entiers

Pour les entiers à grande précision, nous avons choisi d'utiliser des listes d'entiers 64 bits, avec insertion en fin de liste.

Exemple : Pour représenter 2^{100}, deux entiers 64 bits sont nécessaires. Le premier est $2^{100} \bmod 2^{64}$ et le second est $2^{100}/2^{64} = 2^{36}$, ce qui donne la liste [0; 2^{36}].

Code et Implémentation :

Structure de données :

```
(* liste d'entiers a 64 bit *)
type grand_entier = int64 list;;
```

Fonctions primitives :

Insertion (insert) : Cette fonction ajoute une nouvelle valeur à la fin de la liste. Elle est utilisée pour construire progressivement notre grand entier.

```
(* insert : a' list -> a' -> a' list *)
(* insert une valeur en la fin de liste x *)
let insert x valeur =
x @ [valeur]
;;
```

Suppression de tête (remove_head): Cette fonction permet d'éliminer le premier élément de la liste. Cette opération est utile pour manipuler et traiter les parties successives du grand entier.

```
(* remove_head : a' list -> a' list *)
(* enlève le premier element de la liste *)
let remove_head x =
match x with
|[] -> failwith "Il n'y a aucunn élément dans la liste"
|t::s -> s
;;
```

Récupération de tête (get_head): Cette fonction récupère le premier élément de la liste, permettant ainsi d'accéder à la première partie du grand entier.

```
(* get_head : a' list -> a' *)
(* récupère le premier élément de la liste si il existe, sinon erreur *)
let get_head x =
match x with
|[] -> failwith "Il n'y a aucunn élément dans la liste"
|t::_ -> t
;;
```

Affichage (print_grand_entier et print_bits): Ces fonctions permettent d'afficher le grand entier et la liste des bits respectivement, aidant à la visualisation et au débogage.

```
(* print_grand_entier : int64 list -> unit *)
let print_grand_entier x =
Printf.printf "Grand entier : [";
for i = 0 to List.length x - 1 do
Printf.printf "%Ld" (List.nth x i);
if i != (List.length x - 1) then
Printf.printf ";";
done;
Printf.printf "]\n";
;;

(* print_bits : bool list -> unit *)
(* affiche la liste de bits (bool) *)
let print_bits bits =
let rec print_elements lst =
match lst with
| [] -> ()
| true :: s ->
if s = [] then
print_string "true"
else
print_string "true; ";
print_elements s
| false :: s ->
if s = [] then
print_string "false"
else
print_string "false; ";
print_elements s
in
print_string "Bits : [";
print_elements bits;
print_string "]\n";
;;
```

Question 1.2 Décomposition d'un nombre en format binaire

Il est nécessaire de créer une fonction capable de décomposer un entier de taille arbitraire (représenté comme une liste d'entiers) en une liste de bits qui représentent sa décomposition en base 2. Le bit de poids le plus faible doit être présenté en tête de la liste.

Par exemple, pour le nombre 38, sa représentation binaire est 101110. Ainsi, la fonction devrait retourner [false; true; true; false; false; true].

Implémentation

int64_to_bits : traiter chaque élément de type int64 de notre liste. Elle prend un nombre et renvoie sa décomposition binaire sous forme d'une liste de valeurs booléennes. Le processus est récursif: à chaque étape, le reste de la division par 2 est ajouté à la liste, puis le processus est répété avec le quotient.

```
(* int64_to_bits : int64 -> bool list *)
(* convertie un nombre int64 en bits false : 0 et true : 1 *)
let rec int64_to_bits x =
match x with
|0L -> []
|_ -> (Int64.rem x 2L = 1L) :: int64_to_bits (Int64.div x 2L)
;;
```

decomposition : parcourir la liste d'entiers fournie et utiliser la fonction auxiliaire précédente pour décomposer chaque entier. La fonction gère également le cas où l'entier est zéro, ajoutant une liste de bits false de longueur appropriée.

```
(* decomposition : grand_entier -> bool list *)
(* convertie un grand entier en bits *)
let rec decomposition x =
match x with
|[] -> []
(* concat une liste de taille 64 bits remplit de false a une decomposition de la suite de la liste*)
|0L::s -> (create_false_list 64) @ (decomposition s)
|t::s ->
if s=[] then
(* convertie x (le dernier élément de la liste ) : int64 en binaire booléene *)
(int64_to_bits t) @ (decomposition s)
else
(* convertie x en binaire de 64bits *)
(completion (int64_to_bits t) 64) @ (decomposition s)
;;
```

Question 1.3 Complétion d'une liste de bits

L'objectif de cette partie est de pouvoir manipuler une liste de bits pour l'adapter à une taille spécifiée n. Deux cas principaux se présentent :

Si la liste de bits est plus longue que n, alors elle est tronquée pour n'avoir que ses n premiers éléments.

Si la liste est plus courte que n, elle est complétée par des valeurs false pour atteindre la taille souhaitée.

Implémentation

create_false_list : créer une liste remplie de valeurs false. La longueur de cette liste est définie par l'argument n.

```
(* create_false_list : int -> bool list *)
(* créer une liste de false de taille n *)
let rec create_false_list n =
if n <= 0 then
[]
else
false :: create_false_list (n - 1)
;;
```

completion : s'occuper de la manipulation principale de la liste de bits. En fonction de la longueur de la liste initiale et de la valeur de n, elle décide soit de tronquer la liste soit de l'étendre en utilisant la fonction create_false_list.

```
(* completion : bool list -> int -> bool list *)
(* si la liste bits contient au moins n élement alors retourne *)
(* les n ler éléments sinon complète les bits manquantes par des false *)
let rec completion bits n =
match bits,n with
|[],_ -> create_false_list n
|_,0 -> []
|x::s,_ -> x::(completion s (n-1))
;;
```

Question 1.4 convertie bits en grand entier

composition : Transformer une liste de bits en une suite d'entiers en base 10. Convertir des segments de 64 bits, gérer les segments de zéros et assembler le tout en une liste d'entiers qui représente le nombre binaire intégral.

```
(* composition : bool list -> grand_entier *)
(* convertie bits en grand entier *)
let rec composition bits =
match bits with
| [] -> [0L] (* Si la liste est vide, renvoie [0L] *)
| _ ->
(* Si la liste dépasse 64 bits *)
if List.length bits > 64 then
(* prend les 64 premier bits de la liste *)
let head = completion bits 64 in
(* supprime les 64 premier bits de la liste *)
let suite = remove_nb bits 64 in
(* Si les 64 premiers bits sont tous faux, renvoyer [0L] suivi de la composition du reste de la liste *)
if List.for_all (fun x -> x = false) head then
0L :: (composition suite)
else
(* Sinon, convertir les bits en int64 et les ajouter à la composition du reste de la liste *)
(int64_of_binary_list head) :: (composition suite)
else
[int64_of_binary_list bits]
;;
```

Question 1.5 Génération et complétion de Table

table : Transformer un grand entier en séquence de bits sous forme de liste de valeurs booléennes (true ou false) et étendre cette séquence pour obtenir une liste de taille n.

```
(* table : grand_entier -> int -> bool list *)
(* convertie un grand entier en bits et complète a n bits *)
let table x n =
let bits = decomposition x in
completion bits n
;;
```

Question 1.6 Génération aléatoire de grands entiers

genere_aleatoire : générer une séquence d'entiers aléatoires pour composer un grand entier de n bits au maximum. Si n dépasse 64, la fonction crée récursivement des entiers de 64 bits et les empile jusqu'à ce que la taille totale atteigne presque n. Pour les bits restants (moins de 64 bits), elle génère un dernier entier aléatoire qui complète la taille désirée.

```
(* genere_aleatoire : int -> grand_entier *)
(* génère aléatoirement un grand entier de au moins n bits *)
let rec genere_aleatoire n =
Random.self_init ();
if n > 64 then begin
let head = Random.int64 Int64.max_int in
head :: (genere_aleatoire (n - 64))
end else
let x = Random.int64 (Int64.sub (Int64.shift_left 1L n) 1L) in
[x]
;;
```

2 Arbre de décision

Question 2.7 Structure de données

```
type arbre_decision =
| Feuille of bool
| Noeud of int * arbre_decision * arbre_decision
;;
```

Question 2.8 Construire d'un arbre décision

cons_arbre : prendre une table de vérité en entrée et construire un arbre de décision binaire équilibré. Les nœuds internes de l'arbre sont étiquetés avec la profondeur à laquelle ils se trouvent dans l'arbre, tandis que les feuilles sont marquées avec les valeurs de la table de vérité, assignées selon un parcours préfixe.

```
(* cons_arbre : bool list -> arbre *)
(* construit d'un arbre a partir d'une table de vérité *)
let cons_arbre table =
(* aux : bool list -> int -> arbre *)
let rec aux table profondeur =
match table with
| [] -> failwith "La table de vérité est vide.";
| [x] -> Feuille x (* Si la table contient un seul élément, on crée une feuille *)
| _ ->
(* construction en préfixe : créer d'abord la racine puis enfant gauche et enfant droite *)
let noeud = Noeud(profondeur, (Feuille false),(Feuille false)) in
let mid = (List.length table) / 2 in
let gauche = (aux (take mid table) (profondeur + 1)) in (* Récupérer la première moitié *)
let droite = (aux (drop mid table) (profondeur + 1)) in (* Récupérer la seconde moitié *)
let noeud = set_fils_gauche_arbre noeud gauche in
let noeud = set_fils_droite_arbre noeud droite in
noeud
in
let arbre = aux table 1 in
arbre
;;
```

Question 2.9 convertir un arbre en table de vérité

liste_feuilles : parcourir un arbre de décision et compiler une liste ordonnée des valeurs booléennes des feuilles, reflétant les résultats finaux pour chaque chemin dans l'arbre. Elle effectue un parcours de l'arbre, ajoutant les étiquettes des feuilles à la liste dans l'ordre où elles apparaissent lors d'un parcours en largeur, de gauche à droite.

```
(* liste_feuilles : arbre -> bool list *)
(* convertie un arbre en table de vérité *)
let rec liste_feuilles arbre =
match arbre with
| Feuille x -> [x]
| Noeud (_, g, d) -> (liste_feuilles g) @ (liste_feuilles d)
;;
```

3 Compression de l'arbre de décision et ZDD

3.1 Compression avec historique stocké dans une liste

Question 3.10

La structure "liste_deja_vus" est définie de la manière suivante.

```
type liste_deja_vus = (grand_entier * arbre_decision) list
```

Question 3.11 compression par liste

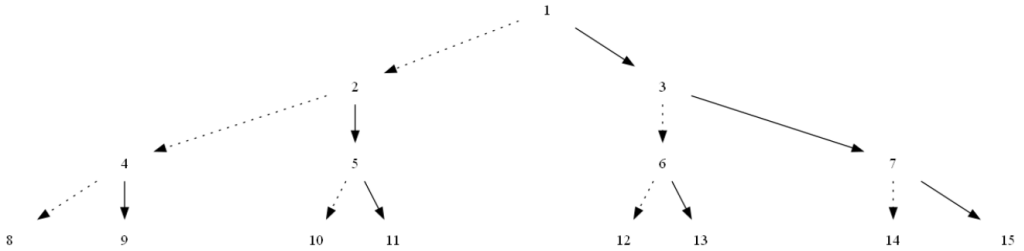
La fonctionnalité compression permet de compresser un arbre grâce à deux règles.

Nous avons séparé cette fonctionnalité en trois fonctions "regle_M", "regle_Z" et "compression_par_liste" qui les utilise. "regle_M" prend en paramètre un arbre de décision et une référence à une liste déjà vue et retourne un arbre de décision compressé. On applique un parcours suffixe. Et on traite chaque cas, lorsque enfant gauche est à enfant droit, si le grand entier de ces deux arbre sont déjà vu avec la méthode "contient". Si c'est bien le cas, "get_pointeur_liste_deja_vus" permet de retourner le pointeur vers l'arbre correspondant aux grand entier recherché. S'ils ne sont pas dans la liste, alors on les ajoute en tête de liste. Le fait que la liste déjà vue est une référence facilite cet ajout.

Question 3.12 dot

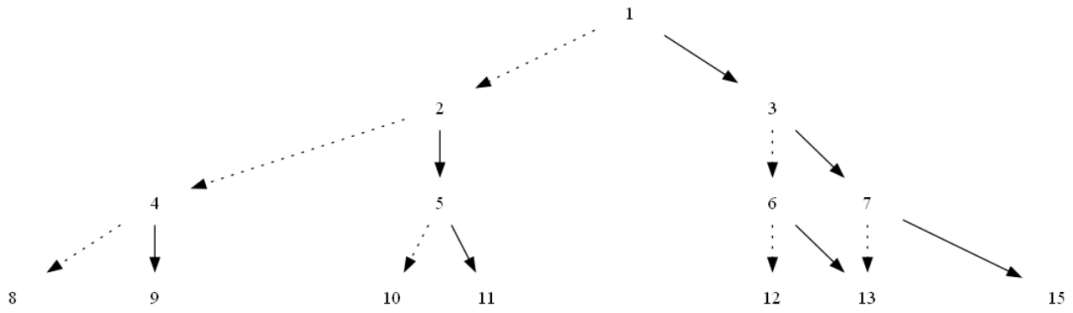
Nous cherchons à convertir notre arbre en langage dot qui puisse générer une image png.

Tout d'abord,dans le fichier dot, chaque nœud possède un id, par exemple la racine de l'arbre est Noeud1, l'enfant gauche de la racine est Noeud 2, l'enfant droit de la racine est



Noeud 3 et ainsi de suite, comme sur l'image suivante..

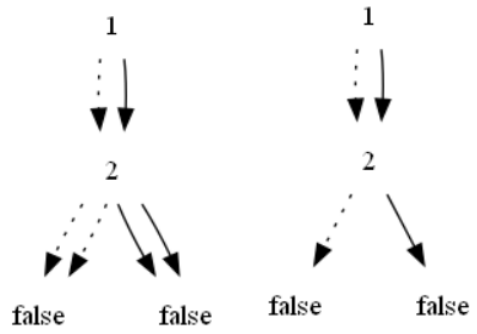
Nous remarquons une règle, lorsque nous avons un noeud courant de id i (ex:5), alors l'enfant gauche est de id i*2 (5*2=10) et l'enfant droit pour i*2+1 (5*2+1=11). Cette règle est appliquée pour la création d'un tableau. Ce tableau contient une paire dont le premier contenant est un arbre de décision et le second un entier. Les index servent a représenté le numéro de nœud, le contenu de l'élément de cette index représente un pointeur vers un arbre de décision correspondant au numéro de l'index et un id qui sera utilisé plus tard dans la génération de code dot. Par exemple t.(5) = (Noeud(5,10,11),5). Par contre t.(i) ne contient pas toujours i comme index. On peut voir dans l'exemple suivant t.(7) a 13 pour id.



Le tableau permet de stocker le pointeur d'un nœud et le id de chaque nœud d'un arbre. Nous avons expliqué précédemment que t.(1) représente la racine alors t.(0) ne sera pas utilisé. Et puis, nous avons initialisé un tableau de taille d'une puissance de 2 supérieur aux nombre de nœuds de l'arbre donnée afin d'éviter les débordements de tableau. Par exemple, un arbre de 15 nœuds va créer un tableau de longueur 16, un arbre de 5 nœuds créer un tableau de longueur 8. Ainsi, les méthode "calcul_nb_noeud", "arrondi_puissance2_superieur", "mise_a_jour_tableau", "tableau" sont définies.

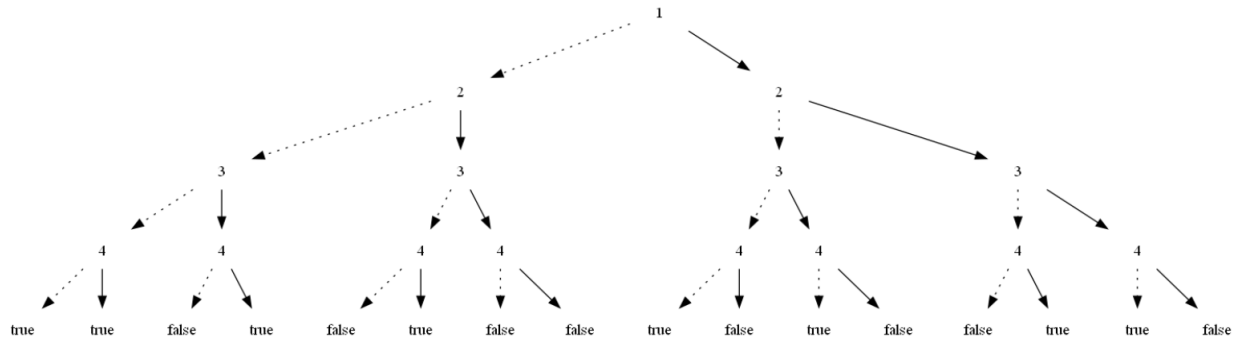
Le tableau est ensuite utilisé pour générer le code en dot grâce à la fonction "generate_dot_arbre" et "dot". Cependant, l'image de l'arbre affiche des flèche redondant. C'est pourquoi, un fichier "file_intermediaire.dot" sert a stocker ce code dot intermédiaire. Ce fichier

sera ensuite traité par la fonction “remove_duplicate_lines” pour enlever les lignes redondant et gardé qu’une seule fois cette ligne.



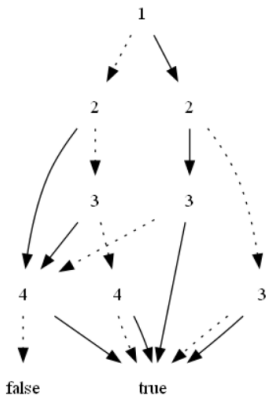
Question 3.13 Figure 1 de l’arbre

Nous obtenons bien en résultat l’arbre demandé sur le grand entier [25899]



Question 3.14 Figure 2 de l’arbre

Enfin, nous obtenons bien l'arbre compressé suivant sur le grand entier [25899].



4 Compression avec historique stocké dans une structure arborescente

Question 4.15 Structure ArbreDejaVus

Nous avons créé et utilisé la structure de données ArbreDejaVus, qui est un arbre de recherche binaire enregistrant des pointeurs vers les nœuds d'un graphe selon un chemin booléen. La fonction add_noeud_arbre_deja_vus nous permet d'ajouter de nouveaux nœuds à cet arbre, tandis que les fonctions get_element_arbre_deja_vus et contient_arbre_deja_vus servent respectivement à retrouver un nœud via un chemin spécifique et à vérifier la présence d'un nœud. De plus, nous utilisons print_arbre_deja_vus pour afficher la structure de l'arbre. Ensemble, ces fonctionnalités soutiennent un algorithme de recherche et de compression efficace, permettant une récupération rapide des informations.

```
type arbre_deja_vus =
| Leaf
| Node of arbre_decision option * arbre_deja_vus * arbre_deja_vus
;;
```

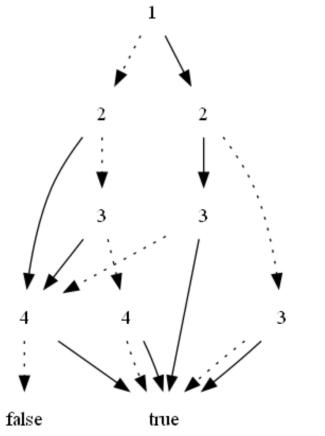
Question 4.16 Optimisation d'Algorithme via ArbreDejaVus

Adapter un algorithme de base afin qu'il utilise la structure ArbreDejaVus pour le stockage et la recherche des nœuds déjà visités. La fonction regle_M_bis est une implémentation récursive qui modifie l'arbre binaire en remplaçant les sous-arbres répétitifs par des pointeurs vers le premier sous-arbre correspondant trouvé. Lorsqu'un nouveau nœud est examiné, la fonction vérifie si les sous-arbres gauche et droit ont déjà été visités à l'aide de contient_arbre_deja_vus. Si c'est le cas, elle remplace ces sous-arbres par des pointeurs récupérés via get_element_arbre_deja_vus. Sinon, elle ajoute les nouveaux sous-arbres à ArbreDejaVus en utilisant add_noeud_arbre_deja_vus. Ce processus optimise l'utilisation de l'espace mémoire et améliore l'efficacité de l'algorithme en évitant les répétitions inutiles.

Question 4.17 Compression Par Arbre

```
(* compression_par_arbre : arbre_decision -> arbre_deja_vus ref -> arbre_decision *)
let compression_par_arbre arbre arbre_deja_vue =
let nouveau_arbre = copier_arbre arbre in
let nouveau_arbre = regle_Z nouveau_arbre in
let nouveau_arbre = regle_M_bis nouveau_arbre arbre_deja_vue in
nouveau_arbre
;;
```

Question 4.18 Résultats



5 Analyse de complexité

Question 5.19

Mesure de complexité naturelle:

La mesure de complexité la plus intuitive pour le problème de compression en ZDD est le nombre de nœuds n dans le ZDD. Le nombre de nœuds capte la taille globale du problème.

Analyse des fonctions:

liste_feuilles : Elle parcourt l'ensemble de l'arbre pour récupérer toutes les feuilles. Sa complexité est O(n) où n est le nombre total de nœuds.

contient et get_pointeur_liste_deja_vue: Ces fonctions parcourent la liste liste_deja_vue de taille maximale m. Leur complexité est donc O(m).

Complexité de regle_M:

La fonction regle_M parcourt chaque nœud de l'arbre (au pire des cas) et effectue un certain nombre d'opérations pour chaque nœud :
Il appelle liste_feuilles deux fois, soit 2×O(n).

Il appelle contient deux fois, soit 2×O(m).

Dans le pire des cas, il pourrait appeler get_pointeur_liste_deja_vue deux fois, soit 2×O(m).

La complexité pour un seul nœud est alors O(n+m). Mais comme cette opération est effectuée pour chaque nœud, la complexité totale est $O(n^2+nm)$.

Complexité de regle_Z:

La fonction regle_Z parcourt chaque nœud de l'arbre et pour chaque nœud, elle appelle la fonction liste_feuilles une fois. Sa complexité est donc $O(n^2)$.

Compression_par_liste:

Cette fonction appelle les fonctions copier_arbre, regle_Z et regle_M séquentiellement.

La complexité de copier_arbre est O(n) car elle parcourt et copie chaque nœud une fois.

La complexité de regle_Z est $O(n^2)$.

La complexité de regle_M est $O(n^2+nm)$.

La complexité totale est donc $O(n^2+nm)$.

Algorithme compression_par_arbre :

Le pire cas se produit lorsque l'arbre de décision est un arbre complet et que chaque chemin unique conduit à un nouvel ajout dans l'arbre des déjà vus.

La complexité en temps de l'ajout à arbre_deja_vus est O(h) où h est la hauteur de l'arbre, ce qui, dans le pire cas d'un arbre équilibré, serait O(log n) où n est le nombre de nœuds dans l'arbre des déjà vus.

À chaque nœud, l'algorithme doit potentiellement ajouter deux nouveaux nœuds à l'arbre déjà vus (un pour chaque enfant gauche et droit), donc cela est fait au plus 2n fois.

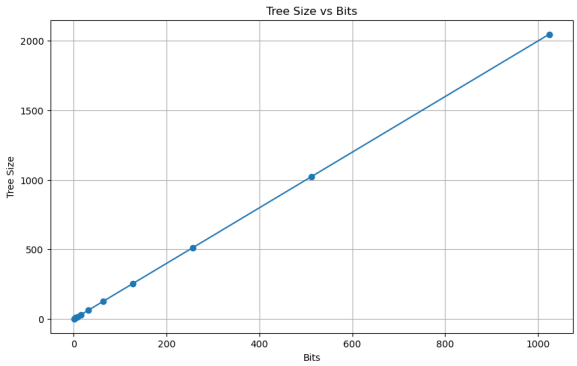
Donc, la complexité en temps de regle_M_bis est O(n log n) dans le pire cas.

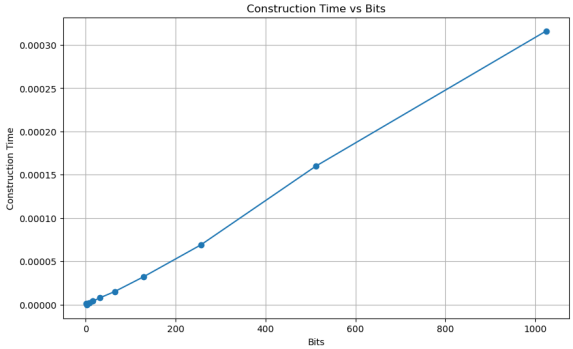
Algorithme compression_par_arbre:

Cet algorithme copie d'abord l'arbre (O(n) pour n noeuds), applique la règle Z (qui pourrait être O(n) dans le pire cas), et ensuite regle_M_bis (O(n log n) dans le pire cas).

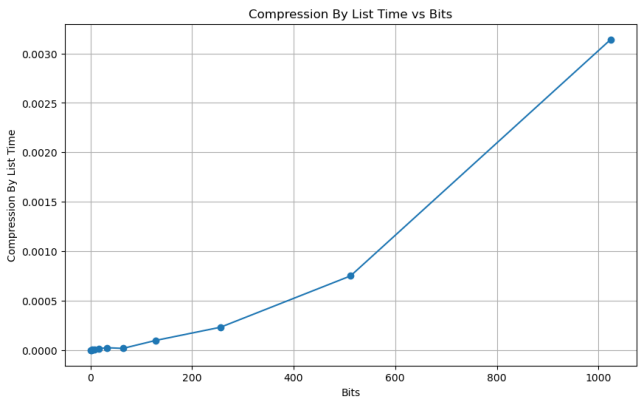
La complexité totale de compression_par_arbre serait alors O(n) + O(n) + O(n log n), ce qui est dominé par O(n log n) dans le pire cas.

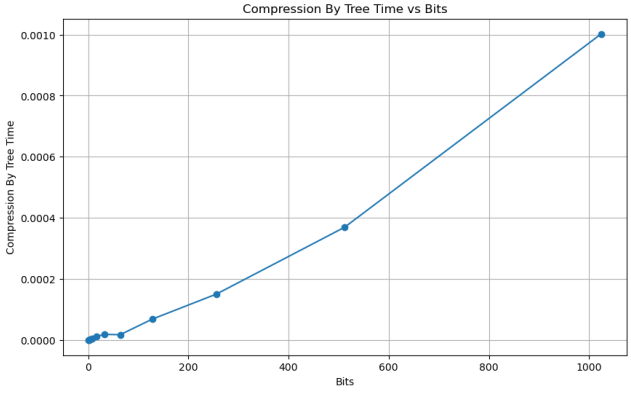
6 Étude expérimentale





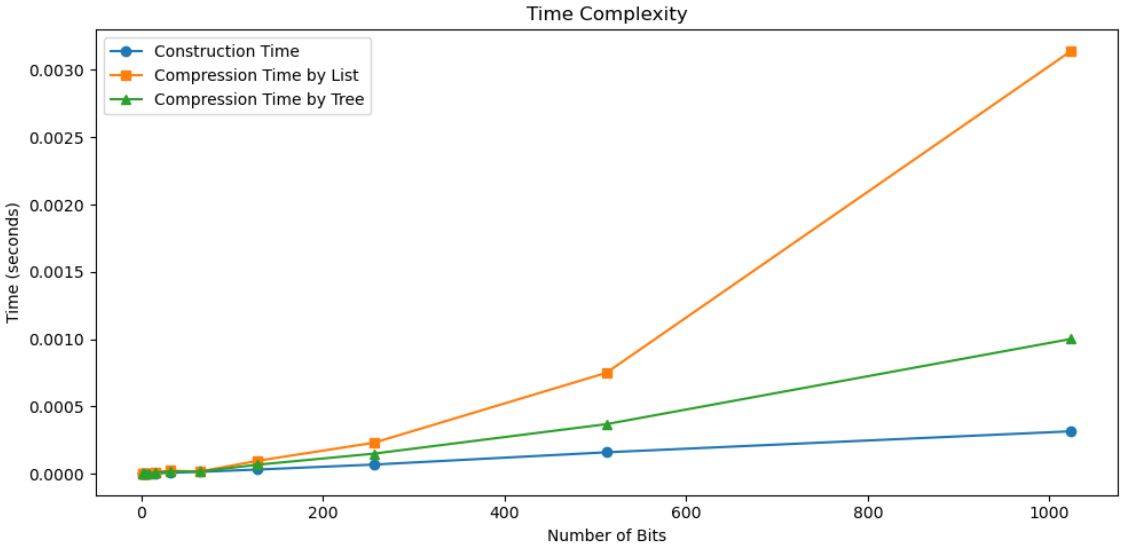
"Taille de l'arbre vs Bits" montre une augmentation directement proportionnelle de la taille de l'arbre à mesure que le nombre de bits augmente, indiquant une mise à l'échelle linéaire prévisible de la taille de la structure de données avec l'augmentation de la taille d'entrée.



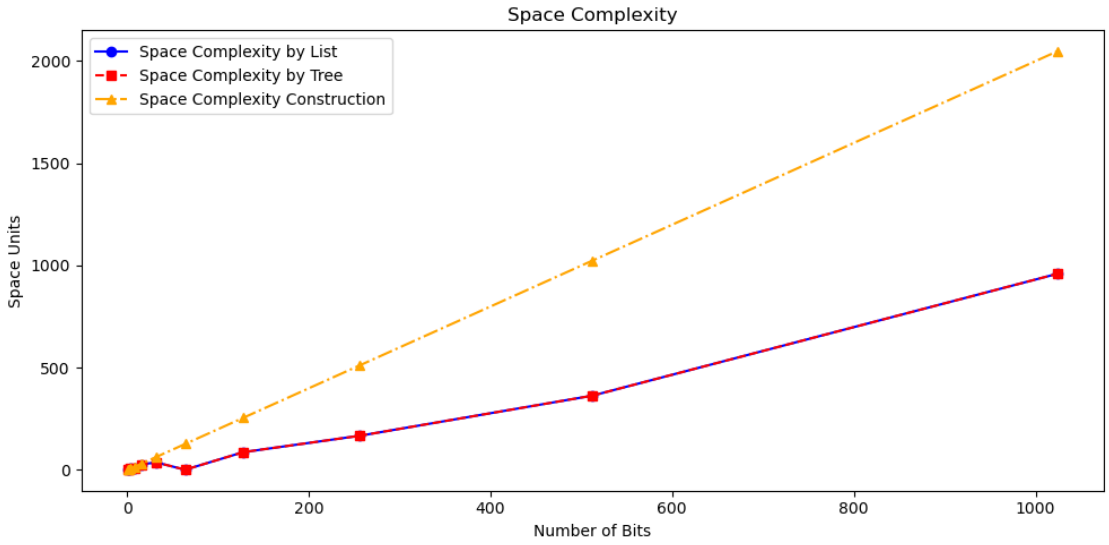


Dans le graphique "Temps de construction vs Bits", on remarque une tendance linéaire évidente où le temps de construction s'accroît plus rapidement à mesure que le nombre de

bits augmente, soulignant un coût computationnel plus élevé associé à la construction de diagrammes de décision binaires plus grands.



Les graphiques illustrent tous deux une tendance à la hausse du temps de compression à mesure que le nombre de bits augmente. Notamment, le temps de compression par arbre reste significativement inférieur à la compression par liste pour la plupart, ce qui s'aligne avec un processus de compression plus efficace lors de l'utilisation de l'approche basée sur l'arbre.



Le graphique illustre que la complexité spatiale est équivalente pour les méthodes de compression par liste et par arbre, ce qui indique qu'elles nécessitent une quantité d'espace équivalente par rapport au nombre de bits. De plus, les représentations compressées occupent moins d'espace que la taille de l'arbre original, ce qui suggère une réduction efficace de l'espace grâce à la compression.