Projet Ouverture

Génération de diagrammes de décision binaires

UE d'Ouverture

RAVINDRAN Rajith
UTHAYAKUMAR Arnaud

Sommaire

Introduction

Dans le cadre de l'UE d'Ouverture du Master 1 en Sciences et Technologies du Langage (STL), ce projet de programmation réalisé en binôme vous propose d'explorer un domaine captivant de l'informatique : la génération de diagrammes de décision binaires, abrégés ZDD. Les ZDD, "Zero-Suppressed Binary Decision Diagram", constituent un concept clé dans la représentation et la manipulation de structures de décision.

La première partie de ce devoir nous amènera à explorer la manipulation de grands entiers en utilisant le langage OCaml. Nous allons concevoir une structure de données pour représenter des entiers en 64 bits, élaborer des fonctions d'insertion, de récupération, de décomposition en bits, de complétion de listes, et de composition d'entiers. Ces opérations sont essentielles pour la génération de nombres aléatoires et la création de tables de vérité.

Ensuite, la seconde partie sera consacrée à la création d'arbres de décision à partir de tables de vérité. Nous allons définir une structure de données pour encoder ces arbres et écrire des fonctions pour créer des arbres de décision équilibrés à partir de tables de vérité. Cette section nous permettra de visualiser les structures d'arbres résultantes.

Pour finir, la troisième partie nous conduira à explorer la compression des arbres de décision en ZDD en utilisant des règles de compression, notamment les règles M et Z. Nous mettrons en place deux approches distinctes pour stocker l'historique de la compression, soit dans une liste, soit dans une structure arborescente. Ces approches nous permettront de convertir efficacement les arbres de décision en ZDD.

Makefile: comment marche notre projet?

Pour compiler notre projet vous pourrez utiliser le Makefile.

Tout d'abord vous pouvez compiler l'ensemble des fichiers en tapant la commande ***make compile.*** Cela créera des fichiers .cmo et .cmi.

Nous avions de plus fait des tests pour chaque module de ce projet (les fichiers test se trouvent dans le dossier "***/src/test".***
Vous pouvez exécuter tout les test en faisant "***make all"*** ou alors vous pouvez les exécuter en faisant un make ***test_$filename où $filename est le nom du fichier*** mis à part pour le fichier ***zdd*** et ***deja_vu*** où on exécute en tapant
.***make test_compressionListe***
.***make test_compressionArbre.***

Lorsque ces makes sont exécutés vous pourrez voir des fichiers "png" apparaître automatiquement grâce au fichier .dot dans le dossier "/src/img", ce sont les résultats de ces tests.

Pour compiler les tests que l'on a fait:
.make expérimentale

Lorsque l'on effectue cette commande, on voit apparaître deux fichiers txt qui seront utiles pour créer des graphes. Ces fichiers txt doivent être utilisés avec le logiciel gnuplot
Pour nettoyer les fichier on a deux commandes possible:
.make mrproper qui supprime seulement les fichier inutiles comme les png les dot, les fichier .cmo générer

.make clear qui supprime tous les fichiers générer mis à part le code source.

BigInt

```
module BigInt :
  sig
    type big_int = int64 list
    val completion : bool list -> int -> bool list
    val auto_completion : bool list -> bool list
    val decomposition : big_int -> bool list
    val composition : bool list -> big_int
    val table : big_int -> int -> bool list
    val gen_alea : int -> big_int
  end
```

Le module BigInt (dans le fichier *big_int.ml*) contient un type qui se compose d'une liste de int64. Cette structure permet de représenter des listes d'entiers de 64 bits pour les opérations et primitives. Le module propose différentes opérations telles que la complétion, l'auto-complétion , la décomposition en bits, la composition à partir de bits, la création de tables de vérité, et la génération d'entiers aléatoires.

Completion

```
val completion : bool_list-> int-> bool_list
```

La fonction completion prend une liste de bits représenté en booléen et un entier naturel. Elle renvoie une liste tronquée ne contenant que ses n premiers éléments, soit la liste complétée à droite, de taille n,complétée par des valeurs false

Auto-completion

```
val auto_completion : bool list -> bool list
```

La fonction auto_completion prend une liste de bits représenté en booléen . Elle renvoie une liste tronquée contenant une taille égale à une puissance de 2 supérieure ou égale à la taille de base de cette liste.

Décomposition en Bits

```
val decomposition : big_int -> bool list
```

La fonction decomposition prend une liste d'entiers de 64 bits et renvoie une liste de bits représentant la décomposition binaire des entiers où “true” correspond à 1 en binaire et false correspond à 0 . Les bits de poids faible sont présentés en tête de la liste.

Composition à partir de Bits

```
val composition : bool list -> big_int
```

La fonction composition prend une liste de bits et construit une liste d'entier de 64 bits représentant l'entier sur n bits où n est la taille de la liste de booléen, correspondant à l'écriture binaire des bits.

Création de Tables de Vérité

```
val table : big_int -> int -> bool list
```

La fonction table prend une liste d'entier de 64 bits et une taille de bits, puis renvoie une liste de bits correspondant à la table de vérité de l'entier. Elle utilise les fonctions de décomposition et de complétion pour atteindre la taille spécifiée.

Génération d'Entiers Aléatoires

```
val gen_alea : int -> big_int
```

La fonction gen_alea génère un grand entier aléatoire contenant jusqu'à 64 bits. Elle prend en entrée un nombre n qui représente le nombre de 64 bits à générer, puis renvoie un entier de 64 bits aléatoire. Nous avons essayer d'utiliser Random.int() mais nous avons remarqué que la fonction était incapable de générer après 2^30. Donc nous sommes partis sur Random.bits64() qui satisfait nos besoins.

Fichier Test

Nous avons aussi un fichier test qui permet de tester toutes les fonctions.
Il définit une fonction 'string_of_bool' qui prend en argument une liste de booléens et renvoie une représentation sous forme de chaîne de caractères de cette liste. Cette fonction parcourt la liste et transforme chaque booléen en "true" ou "false", puis concatène les résultats avec des espaces entre eux.
La partie principale du code sert à tester la fonction 'decomposition'. Le code crée un tableau `lst` qui contient deux valeurs, '38L' et '68719476736L', où "L" indique que ces valeurs sont des entiers longs. Ensuite, il obtient la longueur de ce tableau et itère sur chaque élément du tableau.
À chaque itération, le code utilise la fonction 'decomposition' pour décomposer la valeur dans le tableau en une liste de booléens. Ensuite, il affiche la longueur de cette liste (le nombre de bits) suivie de la liste sous forme de chaîne de caractères en utilisant la fonction 'string_of_bool' dans le terminal.

Arbre de décision

```
module DecisionTree :
  sig
    type btree = Leaf of bool | Node of btree * int * btree
    val sub_list : bool list -> int -> bool list
    val cons_arbre : bool list -> btree
    val liste_feuilles : btree -> bool list
  end
```

Le code, qui se trouve dans le fichier *decision_tree.ml*, se concentre sur la manipulation d'arbres de décision. Les arbres de décision sont couramment utilisés dans l'apprentissage automatique et la prise de décision pour modéliser des scénarios de choix basés sur des décisions binaires. L'objectif principal est de créer une structure de données pour représenter un arbre de décision, de construire un arbre de décision à partir d'une liste de booléens et de récupérer les étiquettes à partir d'un nœud de l'arbre.

Structure de données

```
type btree = Leaf of bool | Node of btree * int * btree;;
```

Le code commence par définir une structure de données pour les arbres de décision. L'arbre de décision est défini de manière récursive avec deux types possibles de nœuds :

- *Leaf* : Un nœud terminal qui contient une valeur booléenne.
- *Node* : Un nœud interne qui contient deux sous-arbres et une étiquette entière.

Cette structure de données permet de représenter des arbres de décision binaires où chaque nœud interne a deux enfants qui sont des sous-arbres et chaque nœud externe sont des booléens prenant true si dans la représentation binaire, on a 1 et false sinon .

Construction

```
val cons_arbre : bool list -> btree
```

La construction d'un arbre de décision à partir d'une liste de booléens est réalisée de manière récursive. L'algorithme crée des nœuds internes de type Node pour chaque paire de booléens consécutifs dans la liste. Lorsque la liste ne peut pas être divisée davantage, des nœuds terminaux de type Leaf sont créés avec les booléens restants. La profondeur de l'arbre est également suivie pour chaque nœud.
Cela permet de construire un arbre de décision qui représente de manière hiérarchique les décisions basées sur les valeurs booléennes de la liste d'entrée.

Récupérer les étiquettes

```
val liste_feuilles : btree -> bool list
```

La fonction *liste_feuilles* est utilisée pour récupérer les étiquettes (valeurs booléennes) à partir d'un nœud de l'arbre de décision. La fonction parcourt l'arbre de manière récursive et retourne une liste de toutes les étiquettes présentes dans les nœuds terminaux sous ce nœud.

Compression avec historique stockée

Le code, dans *deja_vu.ml*, présente l'implémentation de deux structures de données pour gérer la mémoire et améliorer l'efficacité de la compression ZDD appliquée aux arbres de décision binaires.

DejaVu

```
module type DejaVu =
  sig
    type deja_vu
    val find : deja_vu -> DecisionTree.btree -> DecisionTree.btree *
deja_vu
    val empty : deja_vu
  end
```

Le module DejaVu est une interface qui définit la structure de données *deja_vu*, utilisée pour stocker une structure de données quelconque (par exemple une table de hachage ou un arbre binaire) selon un pointeur d'un arbre de décision. Elle initialise aussi les structures de *find* et *empty*.

ListeDejaVu

```
module type ListeDejaVu :DejaVu
```

Ce module hérite de la signature dela structure *DejaVu* en utilisant une liste. La structure *deja_vu* est une liste de couples **(big_int * btree)** où big_int est la clé pour retrouver le pointeur. Ce module expose deux fonctions principales :

- *find* : Cette fonction effectue une recherche récursive dans la liste pour retrouver le pointeur en fonction de sa clé. Si le pointeur est déjà présent, il est retourné avec la liste inchangée. Sinon, le pointeur est inséré dans la liste, et la nouvelle liste est renvoyée avec le pointeur associé.
- *empty* : Cette fonction renvoie une liste vide pour initialiser la structure *deja_vu*.

ArbreDejaVu

```
module type ArbreDejaVu :DejaVu
```

Ce module hérite de la signature de la structure DejaVu en utilisant un arbre binaire de recherche ou si on parcourt le sous arbre gauche de l’arbre alors ça revient à voir une feuille “false” et sinon on a une feuille égale à “true”. La structure de type

deja_vu est une structure arborescente où chaque nœud Node contient des références vers des arbres de décision (**btree**). Ce module expose trois fonctions principales :

- *creer_abr* : Cette fonction permet de créer un arbre binaire s'il n'existe pas déjà, en utilisant les informations de la liste bool et l'arbre de décision btree. Elle crée un nouvel arbre binaire en suivant le parcours des nœuds, où l'arête gauche est associée à false et l'arête droite à true.
- *find* : Cette fonction effectue une recherche récursive dans l'arbre binaire pour retrouver le pointeur en fonction du parcours de la liste bool. Si le pointeur est déjà présent, il est retourné avec l'arbre inchangé. Sinon, le pointeur est inséré dans l'arbre, et le nouvel arbre est renvoyé.
- *empty* : Cette fonction renvoie une feuille comme élément vide pour initialiser la structure deja_vu.

Compression ZDD de l’arbre de decision

Le code fourni s'attaque au problème de la compression d'arbres de décision en utilisant la structure de données ZDD. Il introduit également trois modules *DejaVu*, *ListeDejaVu* et *ArbreDejaVu* que nous avons vu dans la partie précédente, pour gérer la mémoire des structures déjà rencontrées, améliorant ainsi l'efficacité de la compression. L'objectif étant de comprimer des arbres de décision en suivant les règles de compression ZDD suivant la gestion du stock.

Structure ZDD

La structure ZDD est un concept clé de ce code. Elle permet de représenter les structures de décision binaires sous une forme plus compacte en éliminant les redondances. La compression est basée sur deux règles principales :

- **Règle-M :** Si deux nœuds M et N sont les racines de sous-arbres ayant le même résultat pour *liste_feuilles*, alors les arêtes pointant vers N sont remplacées par des arêtes pointant vers M dans toute la structure ; puis le nœud N est supprimé de la structure.
- **Règle-Z :** Si l’enfant droit de N pointe vers false, alors toutes les arêtes pointant vers N sont remplacées par des arêtes pointant vers l’enfant gauche de N ; puis le nœud N est supprimé de la structure

Module ZDD (DV: DejaVu)

```
module Zdd :
  functor (DV : DejaVu) ->
    sig
      val recuperation_liste_feuilles : btree -> bool list
      val seconde_moitie_false : bool list -> bool
      val compression : btree -> btree
    end
```

Ce module implémente la structure ZDD et est un foncteur qui implement *DejaVu* fourni en argument. Il propose plusieurs fonctions, dont :

- *recuperation_liste_feuilles* : Récupère la liste des feuilles d'un arbre de décision ayant une taille qui est une puissance de 2.
- *seconde_moitie_false* : Vérifie si la deuxième moitié de la liste est constituée de false.

- *compression* : Comprimer un arbre de décision en suivant les règles de compression ZDD. Il utilise la structure DejaVu pour gérer les nœuds déjà rencontrés.

Le code fournit les modules *compressionParListe* et *compressionParArbre*, qui utilisent la structure ZDD pour compresser des arbres de décision. Ces modules sont prêts à être utilisés pour la compression.

Visualisation des arbres

```
module Dot :
  sig
    val print_dot : out_channel ->btree -> unit
    val generer_arbre : btree -> string -> unit
  end
```

Afin de pouvoir visualiser les arbres, nous avons le fichier *dot.ml*. Le module Dot permet de faire des graphes tout en modélisant les arêtes et les nœuds. La création de ces représentations graphiques permet de mieux comprendre.

print_dot

Cette fonction permet d'ajouter un nœud de l'arbre de décision à un fichier .dot. Elle prend en paramètre un canal de sortie et un arbre de décision.
Si l'arbre est une feuille, un nœud est ajouté avec un label correspondant à la valeur de la feuille.
Si l'arbre est un nœud interne, un nœud est ajouté avec un label correspondant à la profondeur de l'arbre. Des arêtes sont également ajoutées pour relier le nœud parent aux nœuds enfants.

generer_arbre

Cette fonction génère la représentation graphique complète de l'arbre de décision. Elle prend en paramètre l'arbre de décision à représenter et un nom pour le fichier .dot résultant. Elle ouvre un fichier .dot dans le répertoire "img" et utilise la fonction *print_graphe* pour générer la représentation .dot de l'arbre. Une fois la génération terminée, le fichier .dot est fermé.

Le fonctionnement du module Dot repose sur une fonction récursive *print_graphe* qui parcourt l'arbre de décision. Les nœuds de l'arbre sont ajoutés au fichier .dot à l'aide de la fonction *print_dot*. Le parcours est effectué en profondeur, en utilisant les listes bordure et vus pour suivre les nœuds à traiter et éviter les doublons. Enfin, le Makefile va convertir les fichiers .dot en .png.

Analyse de complexité

Dans cette partie, nous allons tout d'abord analyser la taille d'un arbre de décision, puis calculer la complexité naturelle de la compression d'un arbre selon l'algorithme implémenté et pour finir on évaluera la complexité au pire des cas.

Taille d'un arbre de décision

Pour construire un arbre de décision, nous considérons une liste de taille *n* avec *n* une puissance de 2.

D'une part, on aura *n* feuilles à positionner.
D'autre part pour mesurer la profondeur, on aura besoin de (n-1) nœuds.

Conclusion:

On obtient donc: taille(n) = n+(n-1) = 2n-1
On a donc une taille en θ(n).

Complexité naturelle de la compression

Analysons maintenant la complexité de la compression, soit n le nombre de bits stockés dans l'arbre de décision.

```
let compression (g:btree):btree=
let rec aux (g:btree)(lst:DV.deja_vu):btree*DV.deja_vu=
  match g with
  | Leaf x -> (DV.find lst g) (*regle M*)
  | Node (xg,x,xd) -> let rlf = (recuperation_liste_feuilles g) in
                      let smf = (seconde_moitie_false rlf) in
                      if smf=true then (aux xg lst) (*regle Z*)
                      else let btl,lstl= (aux xg lst)  in
                      let btr,lstr=(aux xd lstl) in
                      (DV.find lstr (Node(btl,x,btr))) (*regle M*)
in let x,y= (aux g (DV.empty)) in x;;
```

Pour la compression, nous utilisons 3 fonctions annexes que nous avons implantés:

- ***val*** *recuperation_liste_feuilles : btree -> bool list*

  Cette fonction pour rappel permet de récupérer la liste des feuilles et de compléter cette liste pour qu'on obtient une liste avec une taille d'une puissance de 2

```
let recuperation_liste_feuilles(g:btree):bool list =
  auto_completion (liste_feuilles g);;
```

Nous utilisons deux fonctions:

```
let rec liste_feuilles (bt:btree):bool list=
    let rec aux (bt:btree) (acc:bool list):bool list=
      match bt with
      | Leaf x -> acc@[x]
      | Node(g,x,d)-> aux g acc @ aux d acc
    in aux bt [];;
```

Cette fonction récursif terminal parcourt l'arbre de décision en faisant 2 appels récursif jusqu'à atteindre les feuilles. Le parcours est donc intégral, c'est-à-dire qu'on parcourt tous les nœuds de l'arbre. C'est donc **une complexité en O(n)**.

```
let auto_completion (lst:bool list) : bool list =
  let len = (List.length lst) in
  let rec find_puissance (p:int) (n:int):int=
    if p >= n then p
    else find_puissance (p * 2) n
  in (completion lst (find_puissance 1 len));;
```

Cette fonction cherche la puissance de deux supérieur supérieure ou égale à n. Cette **recherche est en O(1) et la complétion est elle en O(n)**, car on parcourt la liste.
La **complexité de l'auto complétion est donc en O(n)**

Ainsi pour la récupération de la liste des feuilles, on obtient une complexité en O(n).

- ***val*** *seconde_moitie_false : bool list -> bool*

Pour rappel, dans cette fonction, on parcourt la seconde moitié de liste pour voir si tous les éléments sont true ou false.

```
let seconde_moitie_false (lst:bool list):bool =
  let len = (List.length lst) in
  let rec aux (lst:bool list) (i:int) (n:int):bool =
      if  (List.nth lst i)=true then false
      else if i=n then true
      else aux lst (i-1) n
    in aux lst (len-1) (len/2);;
```

La complexité est en O(n/2) donc O(n) car on parcourt la seconde moitié de la liste.

- ***val*** *find : deja_vu -> btree -> btree * deja_vu*

Pour rappel la fonction find est une fonction de l'interface DejaVu où on l'utilise dans les modules Liste DejaVu et ArbreDejaVu ces deux modules utilisés différemment la fonction.En effet dans le module liste on recherche la clé dans une liste tandis que dans le second, on recherche la clé via un arbre binaire de recherche.

Évaluons maintenant ces deux find.

Pour celle de ListeDejaVu:

```
let rec find (lst:deja_vu)(bt:btree):(btree*deja_vu) =
  let l = composition (liste_feuilles bt) in
  match lst with
  | [] -> bt,[(l,bt)]
  | x::xs -> let bi,bt2=x in
             if bi=l && bt=bt2 then bt2,lst
             else let i,j=(find xs bt) in i,x::j
```

ListeDejaVu est une liste, la complexite moyenne d'une recherche est en O(n) et l'ajout en O(1) or pour ajouter un élément il faut déjà chercher l'élement donc la fonction est en **O(n)**.

Pour ArbreDejaVu:

```
let find (lst:deja_vu)(bt:btree):(btree*deja_vu) =
  let rec aux (l:bool list)(lst:deja_vu)(bt:btree):(btree*deja_vu) =
    match lst with
    | Leaf -> bt,creer_abr l bt
    | Node (a,b,c) -> if l=[] then
                        match b with
                        | None-> bt,Node(a,Some bt,c)
                        | Some bt2 -> if bt=bt2 then bt2,lst else bt,lst
                      else if (List.hd l)=true then let i,j=(aux (List.tl l) c bt) in i,Node(a,b,j)
                      else let i,j=(aux (List.tl l) a bt) in i,Node(j,b,c)
  in aux (liste_feuilles bt) lst bt
```

ArbreDejavu est un ABR, la complexité moyenne d'une recherche ou un ajout dans un ABR est en **O(log n)**.

La complexité moyenne dans une compression dépend ainsi de la structure utilisée pour stocker l'historique:

- Si la structure de donnée utilisé est une liste la complexité naturelle est: $c(n) = (O(n)*O(n))=O(n^2)$

- Si la structure de donnée utilisé est une liste la complexité naturelle est: $c(n) = (O(n)*O(n))=O(n*log(n))$

Complexité pire cas de la compression:

Dans le pire des cas, la complexité ne change pas pour la structure en liste mais cependant pour la complexité de la structure ABR, on a une complexité en $\Omega(n)$ pour la fonction find, car on insère dans le pire des cas la liste des mots. On a donc une complexité en $\Omega(n^2)$ pour chacune des structures lors de la compression.

Courbes expérimentales

Le code a pour objectif de mesurer et de comparer les performances de deux algorithmes de compression d'arbres de décision, *compressionParArbre* et *compressionParListe*, en fonction de la taille de l'arbre. Les performances sont mesurées en termes de temps d'exécution et de la taille moyenne de l'arbre compressé.

Le code génère des arbres de décision aléatoires de différentes tailles, les compresse avec les deux algorithmes, mesure les temps d'exécution et la taille des arbres compressés, puis enregistre ces données dans des fichiers.

Calcul de la Taille de l'Arbre

La fonction *tree_size* calcule la taille d'un arbre de décision en float. Elle parcourt récursivement l'arbre et compte le nombre de nœuds, en ajoutant 1 pour chaque nœud de décision rencontré.

Mesure du Temps

La fonction *diff_temps* mesure le temps nécessaire pour exécuter une fonction passée en argument. Elle enregistre l'heure de début, exécute la fonction (ici, la compression), puis enregistre l'heure de fin. La différence entre ces deux moments donne le temps d'exécution.

Génération de Liste

La fonction *generation_liste* génère une liste d'entiers de 1 à n. Cette liste est utilisée pour l'axe des abscisses dans les fichiers de données.

Écriture des Données

La fonction *ecriture_donnees* permet d'écrire les données dans un fichier. Elle prend en entrée un fichier, deux listes de données à écrire, et la liste d'incréments correspondant à l'axe des abscisses. Elle écrit ces données dans le fichier avec un format spécifique.

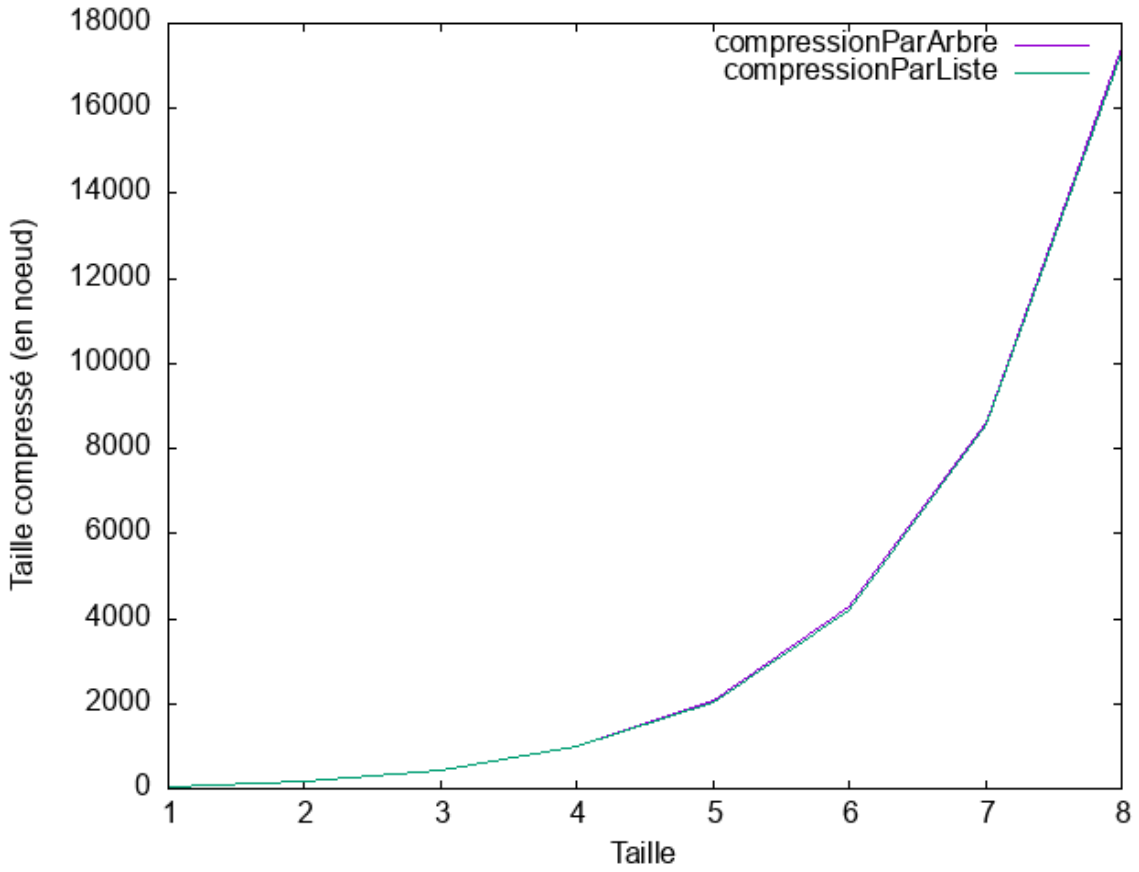
Fonction de Compression

La fonction *compression_test* prend en entrée deux algorithmes de compression (algo1 et algo2), le nombre de tests n, et la taille maximale de l'arbre max. Elle génère des arbres de décision de tailles croissantes (de 2 à max) et mesure le temps d'exécution de la compression pour chaque taille. Les résultats sont stockés dans une liste.

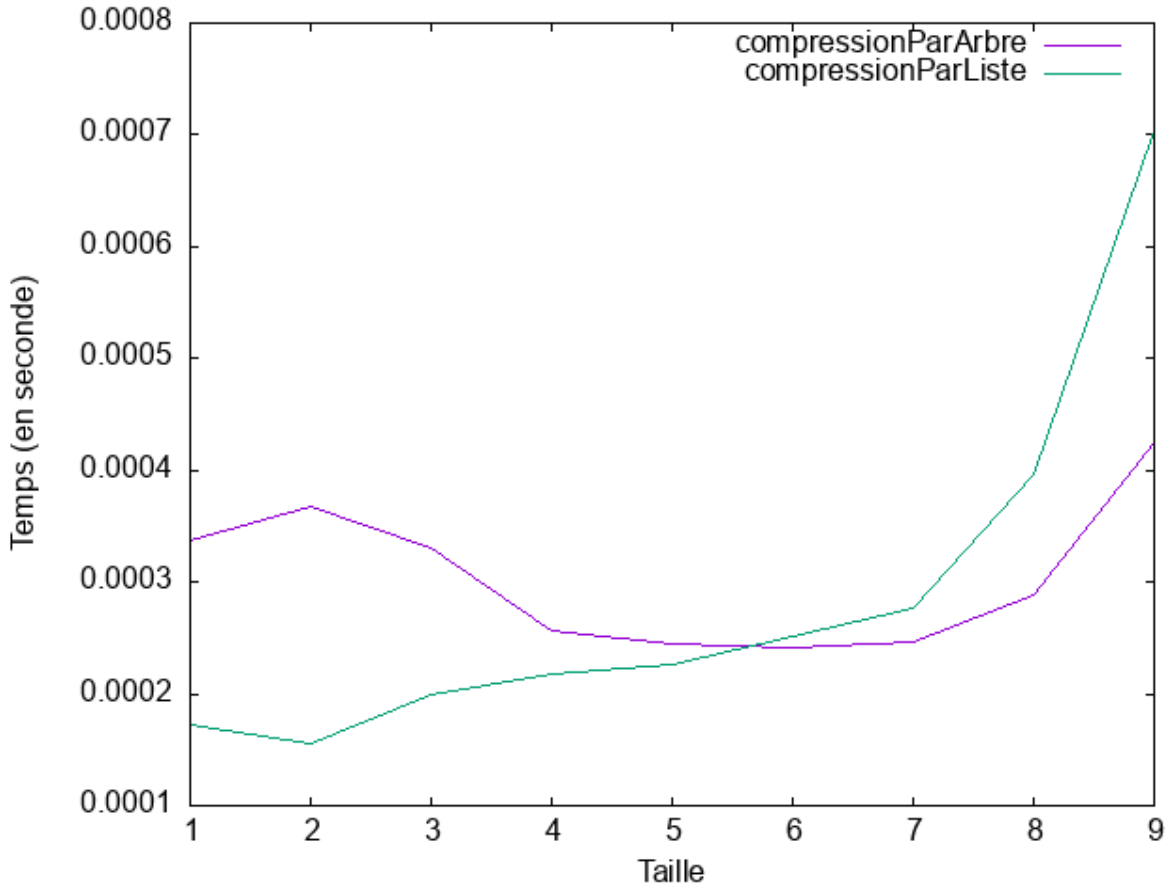
Application

Les deux algorithmes de compression sont testés en utilisant la fonction *compression_test*. Les résultats sont enregistrés dans deux fichiers distincts : "comparaison_vitesse.txt" pour les mesures de temps et "taille_moy.txt" pour les mesures de taille d'arbre.

La visualisation graphique se situe dans le dossier *img* après avoir exécuter le module *gnuplot* dans le terminal. Elle s'obtient en exécutant la fonction *load "graph.gnu".*



Courbe de l'estimation de la taille moyenne de l'arbre compressé



Courbe de la vitesse d'exécution des algorithmes de compression

Distribution de probabilité des tailles des ZDD

Le code qui se situe dans *distribution.ml* évoque l'analyse de la distribution des tailles résultant des tables de vérité aléatoires. Il reprend la majorité des modules que nous avons codés.

La taille de l'arbre

La fonction *tree_size* calcule la taille d'un arbre de décision en entier. Elle parcourt récursivement l'arbre et compte le nombre de nœuds, en ajoutant 1 pour chaque nœud de décision rencontré.

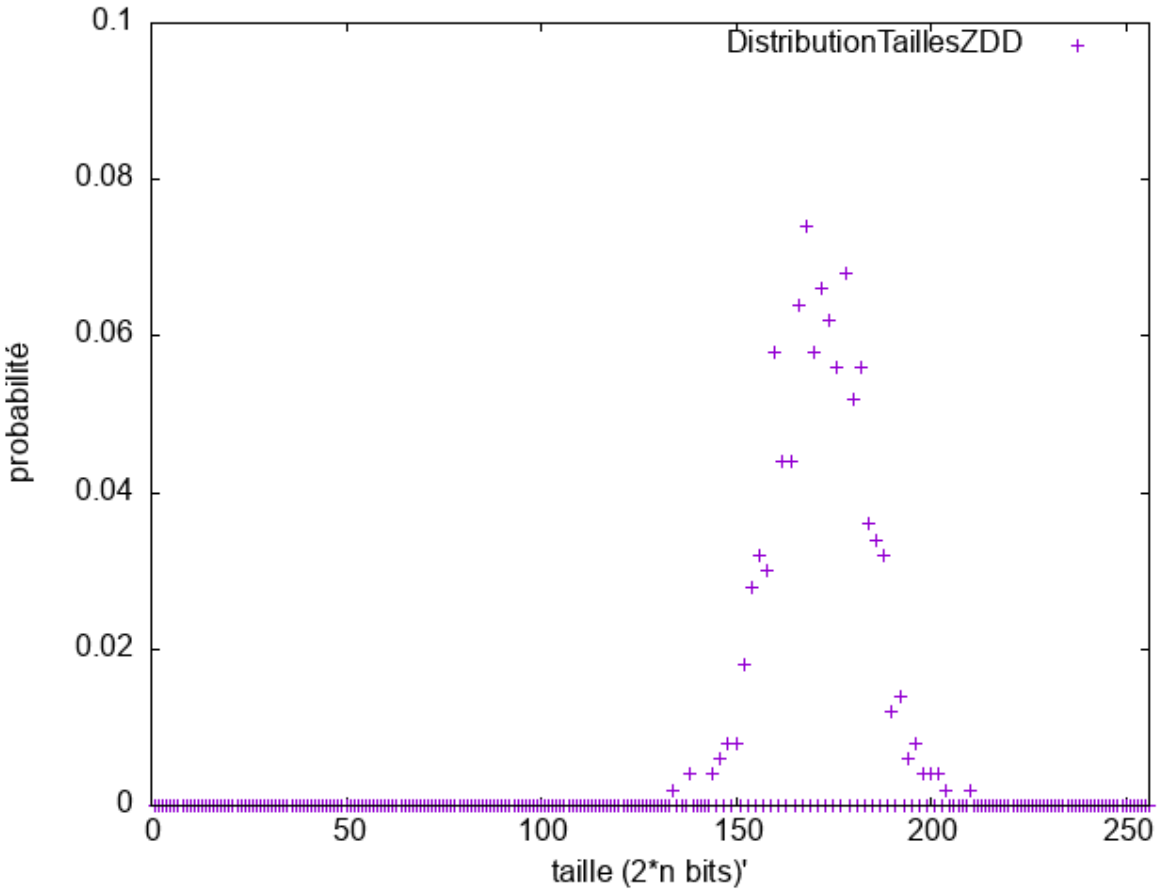
Génération des tables de vérité

La fonction *aux_distribution* génère des tables de vérité aléatoires, construit des ZDD à partir de ces tables. Les paramètres *n* et *acc* correspondent à la taille de la table et au nombre d'itérations. La fonction renvoie un tableau d'entiers représentant la distribution des tailles des ZDD.

Distribution des probabilités

La fonction *distribution* appelle la fonction précédente pour obtenir la distribution des tailles des ZDD. Ensuite, elle écrit la distribution dans le fichier "distribution.txt". Chaque ligne du fichier contient une taille de ZDD et la probabilité correspondante.

La visualisation graphique se situe dans le dossier *img* après avoir exécuter le module *gnuplot* dans le terminal. Elle s'obtient en exécutant la fonction *load "graph.gnu".*



Distribution des probabilités des tailles de ZDD

Conclusion

Ce projet de programmation en OCaml nous a permis de découvrir de la structure de données et de la théorie des graphes en se penchant sur la génération de diagrammes de décision binaires (ZDD) en utilisant des arbres de décision.

Dans la première partie, nous avons commencé par l'échauffement en travaillant sur la manipulation de grands entiers en créant le module BigInt. Nous avons élaboré des fonctions pour l'insertion, la récupération, la suppression, la décomposition en bits, la complétion de listes et la composition d'entiers. Ces opérations sont importantes pour la génération de nombres aléatoires et la création de tables de vérité.

La deuxième partie du projet est la création d'arbres de décision à partir de tables de vérité. Nous avons conçu une structure de données pour encoder ces arbres et avons écrit des fonctions pour créer des arbres de décision équilibrés à partir de tables de vérité. Cette partie nous a permis de visualiser les structures d'arbres résultantes.

La troisième partie a été consacrée à la compression des arbres de décision en ZDD en utilisant des règles de compression, en particulier les règles M et Z. Nous avons mis en place deux approches différentes pour stocker l'historique de la compression, soit dans une liste, soit dans une structure arborescente. Ces approches nous ont permis de convertir efficacement les arbres de décision en ZDD.

Nous avons également abordé la question de la complexité algorithmique des deux méthodes de compression, en calculant la complexité au pire des cas.

Finalement, l'étude expérimentale a été suggérée pour explorer des notions telles que le taux de compression, la vitesse d'exécution et l'utilisation de la mémoire, en utilisant des données générées aléatoirement.

Ce devoir a non seulement approfondi nos compétences en programmation en OCaml, mais il nous a également exposés à des concepts avancés de structures de données, de manipulation de bits, de théorie des graphes et de complexité algorithmique. Il nous a encouragés à réfléchir de manière critique. Enfin, il représente une étape cruciale dans notre parcours académique en informatique.