PROJET OUVERTURE

RAVINDRAN Rajith

UTHAYAKUMAR Arnaud

BIGINT

- Module *BigInt* (dans le fichier big_int.ml) :
- Type *big_int* : une liste d'entiers int64.
- Fonctions pour la manipulation de grands entiers.
- Fonctions pour la création de tables de vérité et la génération d'entiers aléatoires.

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

GÉNÉRATION D'UN ENTIER ALÉATOIRE

- Impossibilité d'utiliser random.int() --> max 2^30
- Création d'un random
- Gen.alea génère l'entier aléatoire sur n bits

int -> big_int

```
let gen_alea (n : int) : big_int =
  let rec aux (n : int) (acc : big_int) =
    if n <= 1 then acc else (aux (n-1) (Random.bits64()::acc))
  in let l = n/64 in let binf = n mod 64 in
  (aux l [ (Int64.shift_right_logical (Random.bits64()) (64-binf)) ]);;
```

ARBRE DE DÉCISION

- Module *DecisionTree* (decision_tree.ml)
- sub_list-> -Divise la liste en 2 en sous-arbre
- cons_arbre-> Construction de l'arbre à partir d'une liste de booléen
- liste_feuilles -> Savoir convertir l'arbre en liste booléen

```
(*QStrucutre de donnée de l'arbre de decision*)
type btree =
Leaf of bool |
Node of btree * int * btree;;
```

bool list -> btree

```
let cons_arbre(lst:bool list):btree=
  let rec aux (lst:bool list) (depth:int):btree=
    match lst with
    | [] -> failwith "Arbre vide"
    | ng::nd::[] -> Node(Leaf ng, depth, Leaf nd)
    | tl -> let n_leaf = (List.length lst)/2  in
                        let ssg = aux (completion tl n_leaf) (depth+1) in
                        let ssd = aux (sub_list tl n_leaf) (depth+1) in
                        Node(ssg, depth, ssd)
in aux (auto_completion lst) 1 ;;
```

COMPRESSION ZDD DE L'ARBRE DE DÉCISION

- Compression d'arbres de décision en utilisant la structure ZDD.
- Règles de compression M et Z.
- Modules pour gérer l'historique de la compression. (*DejaVu*)

```
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;;
```

COMPRESSION AVEC HISTORIQUE STOCKÉ

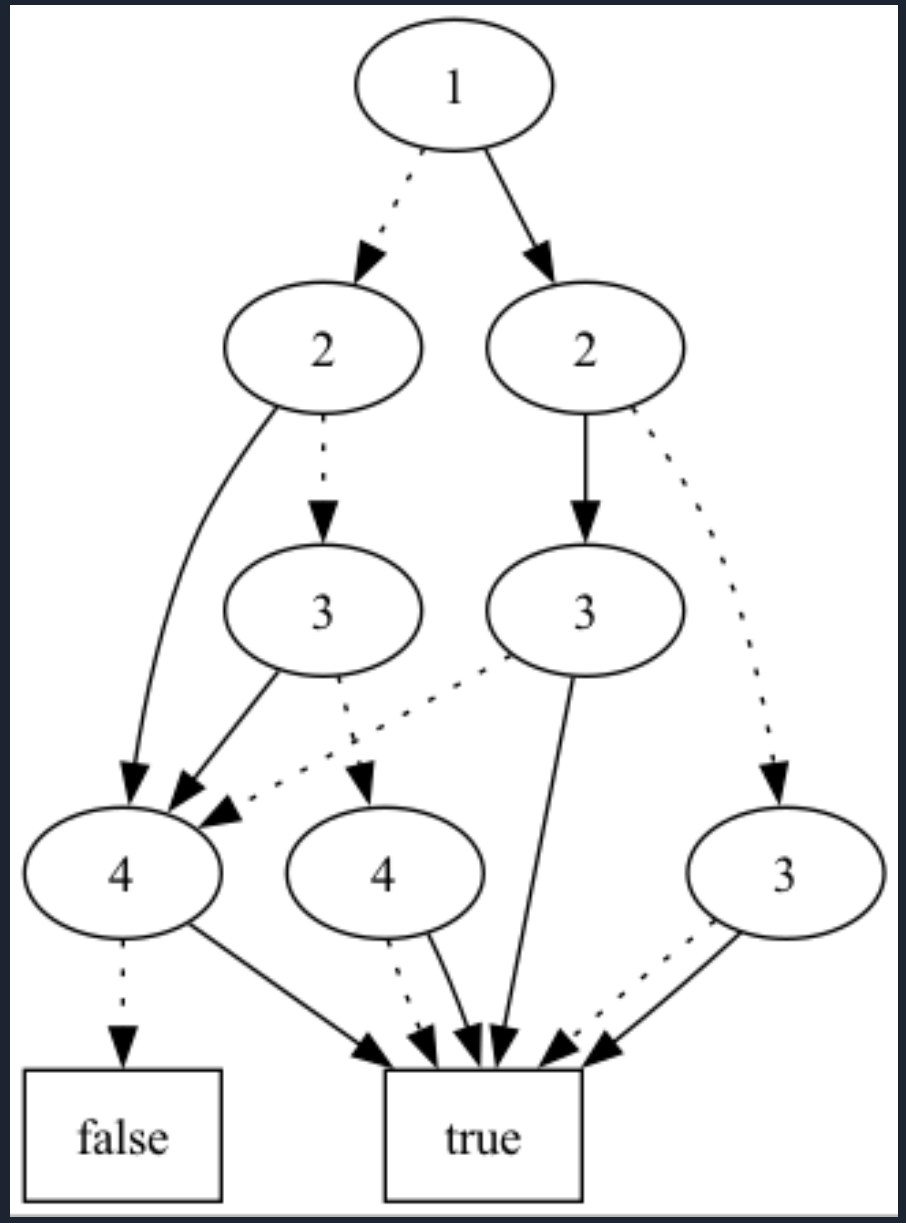
- Deux structures de données : stocker en mémoire et améliorer l'efficacité de la recherche
- *DejaVu* : module typé
- *ListDejaVu* : Stockage de l'historique en utilisant une liste comme clée et un pointeur de l'arbre en valeur implantant le module DejaVu.
- *ArbreDejaVu* : Stockage de l'historique en utilisant un arbre binaire implantant le module DejaVu.

```
(*Module Deja_vu qui est utiliser pour la compression zdd*)
module type DejaVu = sig

  (*
  Structure de donnée permettant de stocké une structure de donnée
  quelconque (par exemple une table de hachage ou un arbre binaire)
  selon un pointeur d'un arbre de decision
  *)
  type deja_vu

  (*
    fonction permettant soit d'inserer un pointeur
    dans la structure de donnée soit de retourner
    ce pointeur si il est deja present
  *)
  val find : deja_vu -> btree -> btree*deja_vu

  (*Utile pour initialiser la structure*)
  deja_vu
  val empty : deja_vu
end;;
```



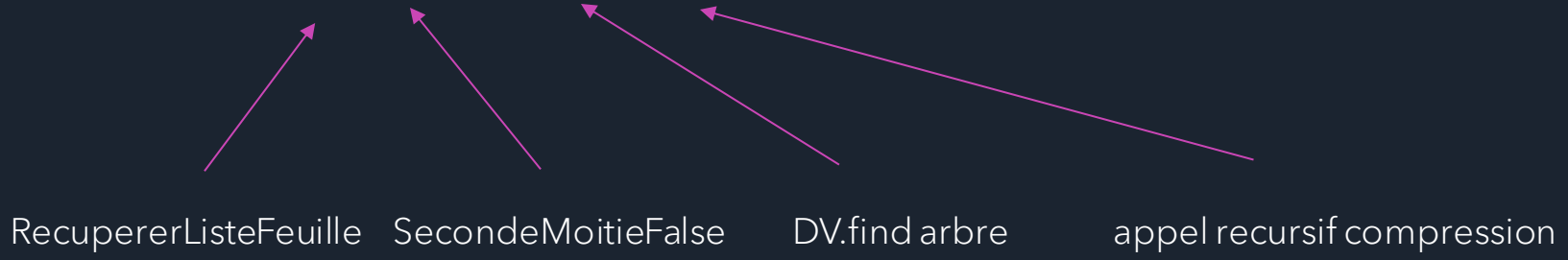
VISUALISATION DES ARBRES

- 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 *Makefile* va convertir les fichiers .dot en .png.

ANALYSE DE COMPLÉXITÉ

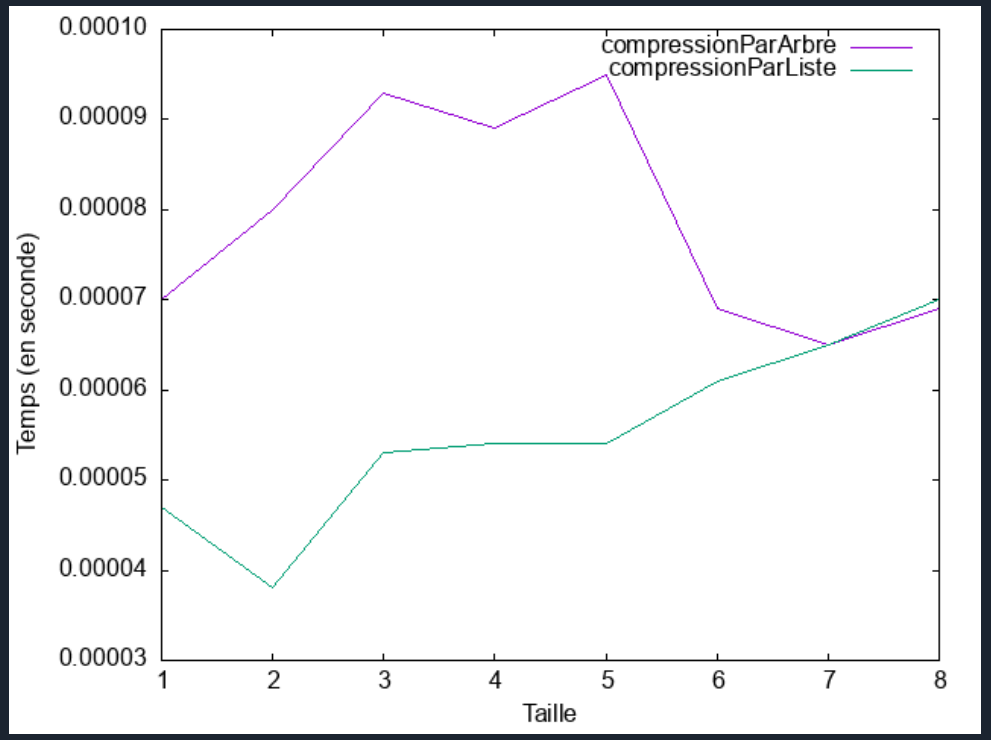
- *La complexité naturelle total pour la compression avec historique stocké en liste:*
- *c(n) = (0(n)+0(n)+0(n**2))log(n)=0(nlog(n))*

- *La complexité naturelle total pour la compression avec historique stocké en tant que structure arborescente:*
- *c(n) = (0(n)+0(n)+0(nlog(n)))*log(n)=0(nlog(n))*

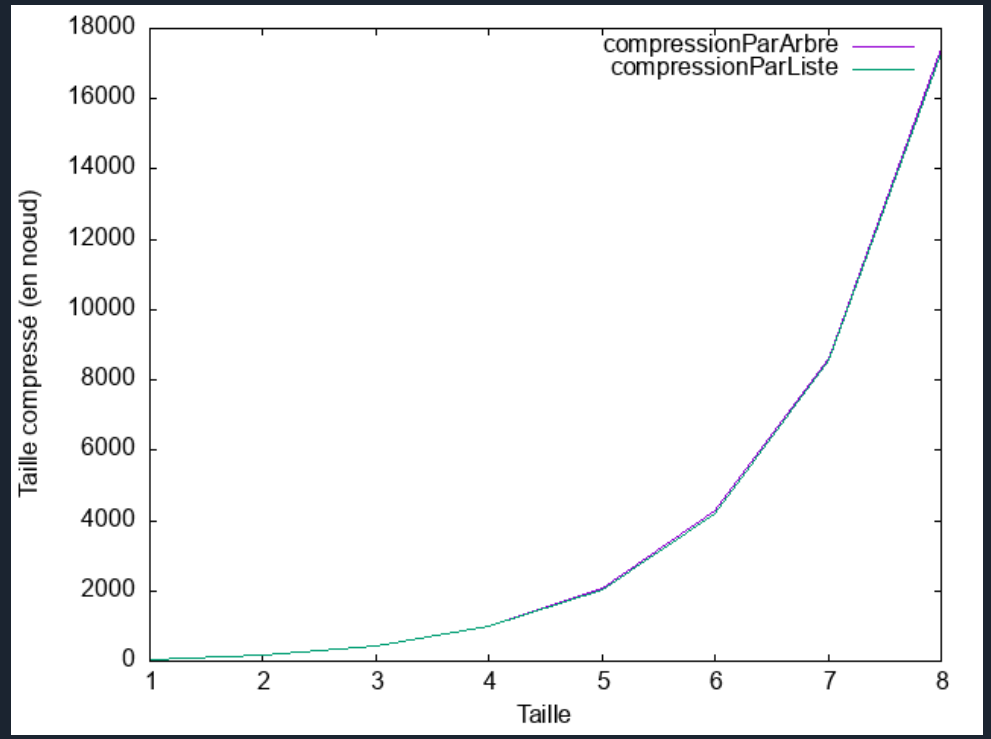


COURBES EXPÉRIMENTALES

- Mesure de performances des algorithmes de compression en fonction de la taille de l'arbre.
- Mesure du temps d'exécution et de la taille moyenne de l'arbre compressé.
- Visualisation graphique des résultats.



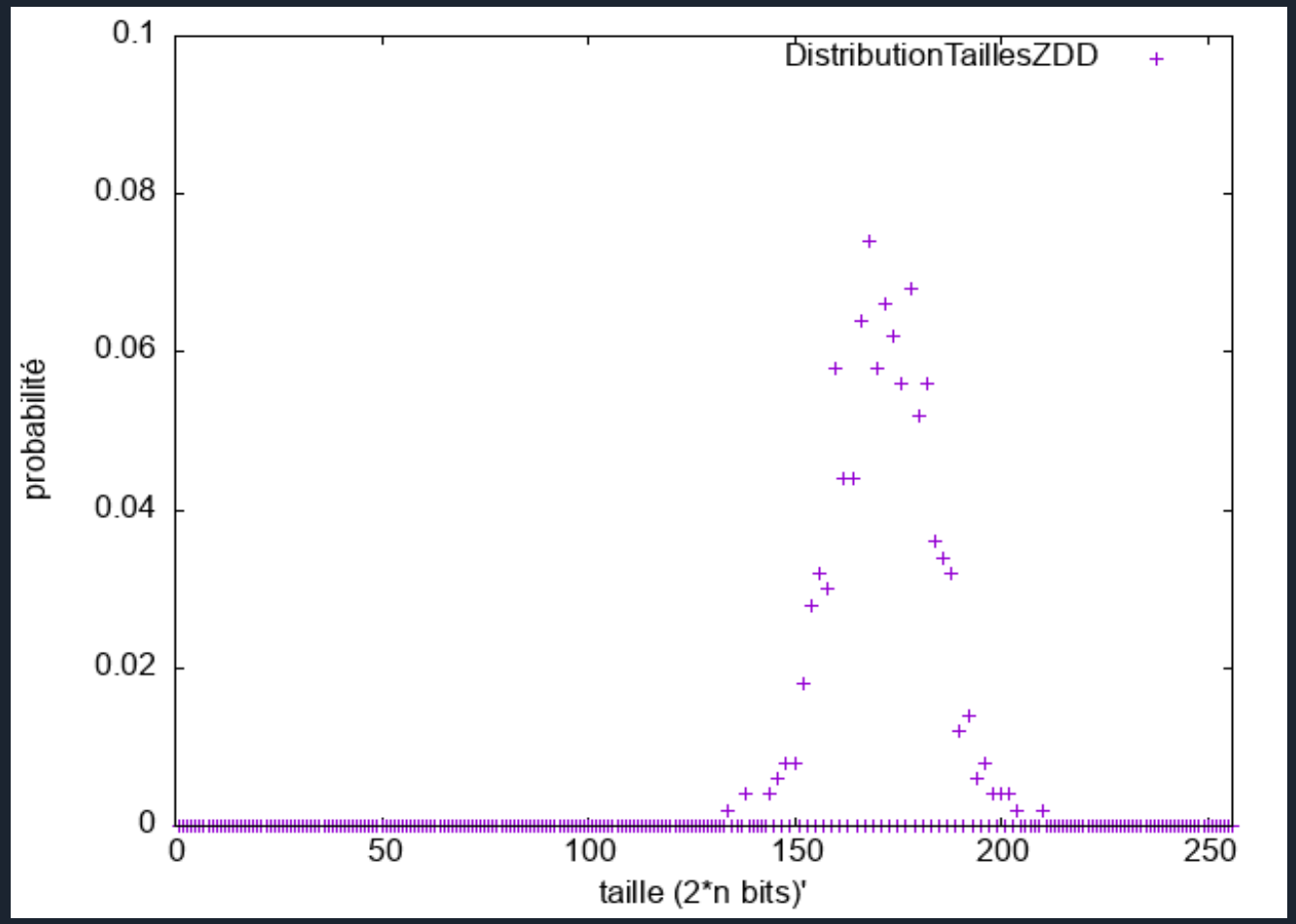
Courbe de la vitesse d'exécution des algorithmes de compression



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

DISTRIBUTION DES PROBABILITÉS

- Analyse de la distribution des tailles des ZDD résultant des tables de vérité aléatoires.
- Calcul de la taille de l'arbre.
- Génération des tables de vérité et distribution des probabilités.



Distribution de probabilité des tailles des ZDD