Projet Ouverture

Devoir de programmation

Master 1 : STL

LIM Floria
FANG Julien

SCIENCES SORBONNE UNIVERSITÉ

2023/2024

Génération d'un grand entier

```
(*génère un entier aléatoire sur n bits au maximum*)
let genererAleatoire n =
  let random = (Random.int64 Int64.max_int) in
  if Random.bool () then
    Int64.shift_right_logical random (64-n)
  else
    Int64.shift_right_logical (Int64.neg random) (64-n)

(*génère un grand entier aleatoire sur n bits au maximum*)
let genAlea n =
  if (n<=64) then [genererAleatoire n]
  else
    let l = n/64 in (*nombre d'éléments sur 64 bits*)
    let m = n - l*64 in (*le reste est sur m bits*)
    let rec aux i =
      if (i=0) then []
      else (genererAleatoire(64))::(aux (i-1))
    in
  let reste = (genererAleatoire m) in
  if (reste <> 0L) then (aux l)@[reste] (*si le reste est à 0, on ne l'ajoute pas*)
  else (aux l)
```

Décomposition d’un grand entier

```
(*décompose un entier en une liste de booléens*)
let decompositionUnEntier x =
  let signe = Int64.compare x 0L in (*on récupère le signe de x*)
  let rec aux x =
    if ( x = 0L) then
      [false]
    else if ( x = 1L) then
      [true]
    else
      if (Int64.logand x 1L = 0L) then (*on recupère le bit de poids faible*)
        false :: (aux (Int64.shift_right_logical x 1)) (*on décale à droite de 1 bit*)
      else
        true :: (aux (Int64.shift_right_logical x 1)) (*on décale à droite de 1 bit*)
  in
    if (signe < 0) then (completion (aux x) 63)@[true] (*si x est négatif, alors il s'agit d'un entier sur 64 bits*)
    else (aux x)

(*décompose un grand entier en une liste de booléens*)
let rec decomposition l =
  match l with
  | [] -> []
  | h::[] -> (decompositionUnEntier h)
  | h::t -> (completion (decompositionUnEntier h) 64) @ (decomposition t) (*on complète la decomposition de h pour qu'elle soit de taille 64*)
```

Arbre de décision d'un grand entier

```
(*structure de donnée permettant d'encoder des arbres binaires de décision*)
type arbre_ref =
| Feuille of bool (*feuille*)
| Noeud of int * arbre_ref ref * arbre_ref ref (*Noeud (profondeur, fils gauche, fils droit)*)
```

```
(*construit un arbre binaire de décision à partir d'une liste de booléens*)
let cons_arbre l =
  let length = List.length l in
  let rec aux l n =
    match l with
    | [] -> ref (Feuille false) (*arbre vide*)
    | h :: [] -> ref (Feuille h) (*feuille*)
    | h :: t ->
        let (l1, l2) = split l in (*on sépare la liste en deux*)
        ref (Noeud(n, aux l1 (n + 1), aux l2 (n + 1))) (*crée un noeud avec les deux listes en faisant un appel récursif*)
  in aux (completion l (puissanceSup length)) 1 (*on complète la liste l pour que sa longueur soit une puissance de 2*)
```

Structure de ListeDejaVus

```
module ListeDejaVus =

struct
  type t = (int64 list * arbre_ref) list ref (*liste de couple (grand entier, pointeur vers un noeud)*)

  let vide () = ref []

  (*ajoute x à la tête de la liste*)
  let insertTete x l =
    l := x :: !l

  (*recherche un grand entier dans la liste*)
  let rec recherche x l =
    match !l with
    | [] -> None
    | (n, a) :: t ->
      if n = x then Some (n, a)  (*si on trouve le grand entier, on renvoie le couple*)
      else recherche x (ref t)  (*sinon on continue la recherche*)

end
```

Compression d’un arbre de décision (1)

```
let rec aux abr l parent depuisGauche =
```

Cas d’une feuille

```
match !abr with
| Feuille b ->
  let n = (composition [b]) in (*on calcule le grand entier de la feuille*)
  let couple = (ListeDejaVus.recherche n l) in (*on cherche si le grand entier est présent dans la ListeDejaVus*)
  (match couple with
    | None -> ListeDejaVus.insertTete (n, abr) l; (*ajout des deux uniques pointeurs vers les feuilles True et False dans la liste*)
    | Some (_, a) -> (*le noeud parent pointera vers l'unique feuille True ou False du graphe*)
     match !parent with
     | Noeud (pp, gg, dd) ->
       if depuisGauche then parent := Noeud(pp, a, dd)
       else parent := Noeud(pp, gg, a)
     | _ -> ())
```

Compression d’un arbre de décision (2)

```
let rec aux abr l parent depuisGauche =
```

Cas d’un noeud : Règle-Z

```
| Noeud (p, g, d) ->
  let liste_bool = (liste_feuilles abr) in (*on calcule la liste_feuilles associé au noeud*)
  let (_, moitie) = (split liste_bool) in (*on recupere la deuxieme moitié de la liste de booléens*)
  if (List.for_all (fun x -> x = false) moitie) then (*on verifie si la deuxième moitié de la liste ne contient que des valeurs false*)
    match !parent with (*on change le pointeur vers son enfant gauche*)
      | Noeud (pp, gg, dd) ->
        if depuisGauche then
          begin
            parent := Noeud(pp, g, dd);
            (aux g l parent true); (*on continue le parcours*)
          end
        else
          begin
            parent := Noeud(pp, gg, g);
            (aux g l parent false) (*on continue le parcours*)
          end
      | _ -> ()
```

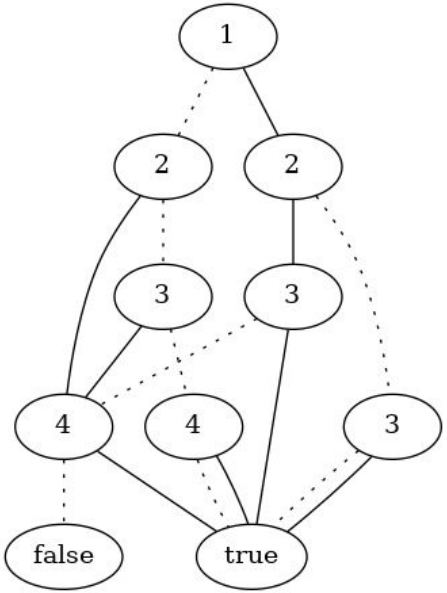
Compression d’un arbre de décision (3)

```
let rec aux abr l parent depuisGauche =
```

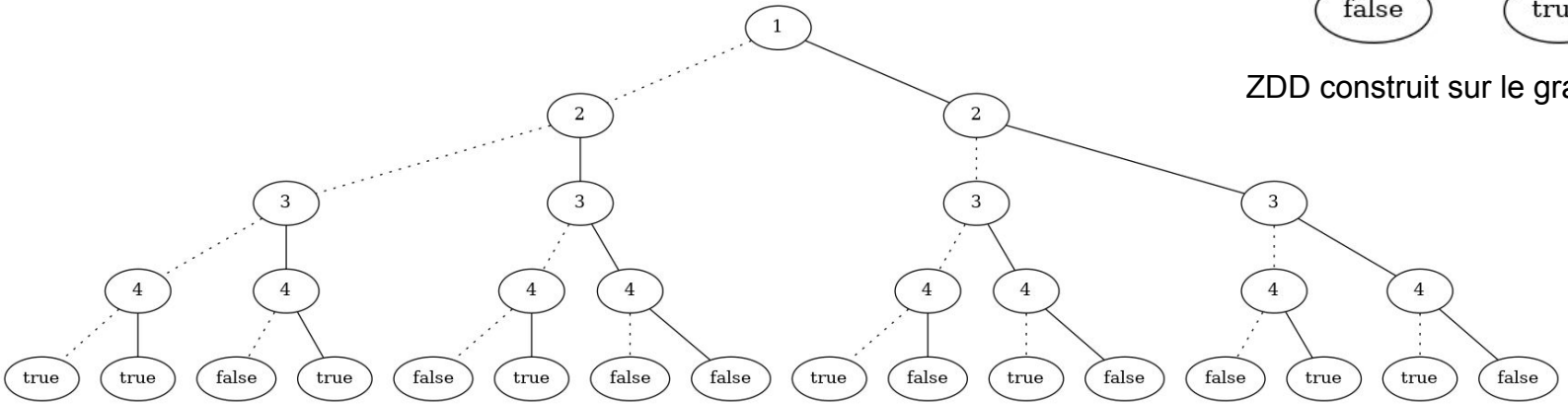
Cas d’un noeud : Règle-M

```
| Noeud (p, g, d) ->
  let liste_bool = (liste_feuilles abr) in (*on calcule la liste_feuilles associé au noeud*)
  (...)
    begin
      let n = (composition liste_bool) in (*on calcule le grand entier associé au noeud*)
      let couple = (ListeDejaVus.recherche n l) in (*on cherche si le grand entier est présent dans la ListeDejaVus*)
      (match couple with
        | None -> (*le grand entier n'est pas dans la ListeDejaVus, on l'ajoute*)
          (ListeDejaVus.insertTete (n, abr) l);
          (aux g l abr true);
          (aux d l abr false);
        | Some (_, a) -> (*le grand entier est dans la ListeDejaVus, on change le pointeur vers le noeud depuis le parent*)
          match !parent with
          | Noeud (pp, gg, dd) ->
            if depuisGauche then parent := Noeud(pp, a, dd)
            else parent := Noeud(pp, gg, a);
          | _ -> ())
    end
```

Graphe produit en langage dot



ZDD construit sur le grand entier [25899]



Arbre de décision issu de la table de vérité de taille 16 construite sur le grand entier [25899]

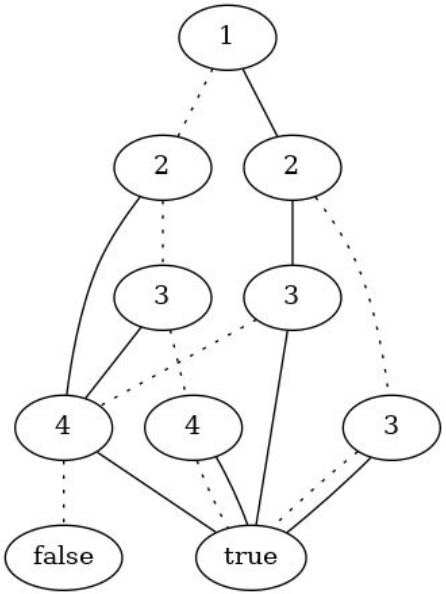
Structure de ArbreDejaVus (1)

```
module ArbreDejaVus =

struct

  type arbreDV =
  | Noeud of arbre_ref ref option * arbreDV ref * arbreDV ref (*Noeud (pointeur vers un noeud, fils gauche, fils droit)*)
  | Feuille (* Feuille *)

  let vide() = ref Feuille  (*arbre vide*)

  (...)

end ;;
```

Structure de ArbreDejaVus (2)

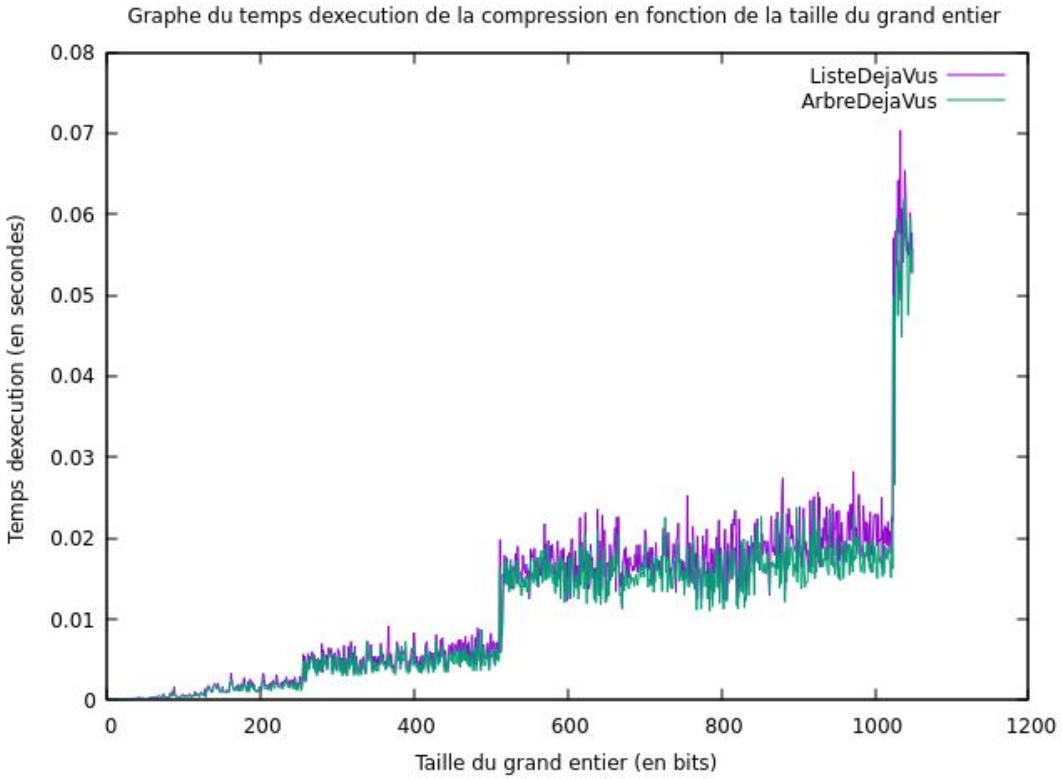
```
(*recherche si il existe un pointeur au bout du parcours de la liste de booléens*)
let rec recherche liste arbre =
  match (liste, !arbre) with
  | [], Feuille -> None
  | [], Noeud (None, _, _) -> None
  | [], Noeud (Some p, _, _) -> Some p
  | h::t, Feuille -> None
  | h::t, Noeud (_, g, d) -> if h then (recherche t d) else (recherche t g)

(*insère un pointeur au bout du parcours de la liste de booléens*)
let rec inserer liste pointeur arbre =
  match (liste, !arbre) with
  | [], Feuille -> (*on arrive à la fin de la liste et on est sur une feuille*)
    arbre := Noeud(Some pointeur, vide(), vide()); (*on crée un noeud ayant pour étiquette le pointeur*)
  | [], Noeud (p, g, d) -> (*on arrive à la fin de la liste*)
    arbre := Noeud (Some pointeur, g, d) (*l'étiquette du noeud contient maintenant le pointeur*)
  | (h::t, Feuille) -> (*la liste n'est pas finie, mais on arrive à une feuille*)
    arbre := Noeud(None, vide(), vide()); (*on crée un nouveau noeud puis on continue le parcours de la liste*)
    inserer liste pointeur arbre
  | h::t, Noeud (_, g, d) ->
    if h then inserer t pointeur d
    else inserer t pointeur g
```

Compression d’un arbre de décision avec ArbreDejaVus

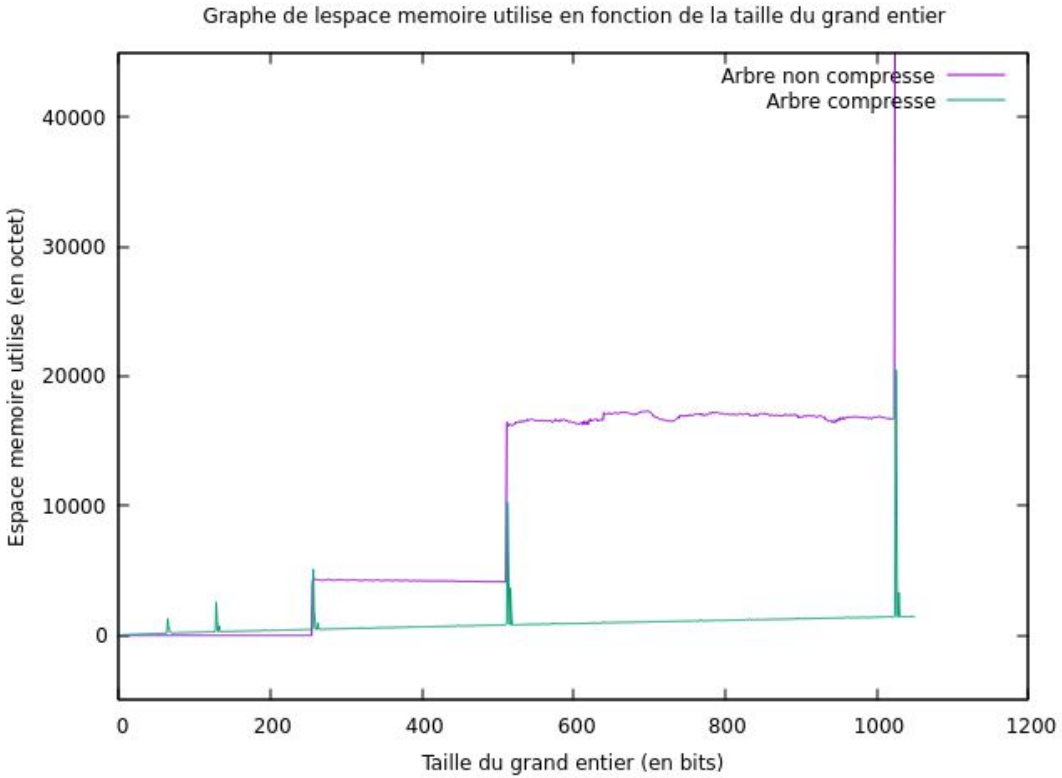


ZDD construit sur le grand entier [25899] à l’aide de ArbreDejaVus

Graphe du temps d'exécution des fonctions de compression



Graphe de l’espace mémoire utilisé



Graphe du taux de compression