Ministère de l’Enseignement Supérieur, de la Recherche et de l’Innovation

Sorbonne Université

Faculté des Sciences et Ingénieries

Département d’Informatique

Rapport du devoir de programmation

« Arbres binaires de recherche : compression et expérimentation »

Présenté par :

Sofiane BRANECI

Amine BENSLIMANE

Encadré par :

M. Emmanuel CHAILLOUX

M. Martin PEPIN

Pour l’UE :

Ouverture

Contexte et Motivations

Un arbre binaire de recherche est une notion classique et fondamentale dans le jargon de l'informatique théorique.

Un ABR est une structure de données qui propose un ordre total sur ses nœuds. Sa particularité et sa puissance résident sur sa facilité d'implémentation mais surtout sur la complexité en nombre de comparaisons (au cas moyen) en $O(\log n)$ effectué pour une recherche.

Si bien que cette structure de données soit performante, le nombre de nœuds d'un ABR peut vite devenir important et ainsi être gourmant en espace mémoire occupé.

Dans ce rapport, nous vous proposons notre contribution pour la compression d'un arbre binaire de recherche.

Nous y définissons le nouveau type de la structure de données compressée, l'algorithme de compression et de recherche et nous finissons par une expérimentation des résultats obtenus.

1.1 Synthèse de données

Dans cette première partie on définit un ensemble de fonction d'utilités qui nous permettra la génération de permutations aléatoires afin de pouvoir construire les arbres correspondant aux listes générées et qui seront utilisés dans les jeux de testes.

Q 1.1 :

EXTRACTION_ALEA :

$$IN : L\, , P = List$$
$$OUT : (L - \{elt\}, \{elt\} \cup P), \qquad 1 \leq elt \leq L.taille$$

Prend en paramètre deux listes L et P et renvoi un couple de listes dont la première est la liste L sans son r^e élément et la deuxième est P avec en tête le r^e élément de L. r étant choisi au hasard ;

Q 1.2 :

GEN_PERMUTATION :

$$IN : \ n = Entier$$
$$OUT : P = Liste\ permutée\ , \forall i\ \in [1..n], \exists j\ \in [1..n]\ tels\ que\ P[j] = i$$

A partir d'un entier n, on définit les listes $L = [1..n]$ et $P = []$. On appelle alors la fonction $EXTRACTION\_ALEA$ sur $L\ et\ P$ jusqu'à vider L. Le résultat est la liste P ;

Voici l'implémentation que nous proposons :

```
(**
renvoie un couple de listes dans lequel le r-ième(noté elt) element de L a été insert en tete de P
params
  l: list
  p: list
return
  (L-{elt}, {elt} UNION P)
*)

let extraction_alea l p =
  Random.self_init ();
  let rIndex =   Random.int (List.length l) in
  let rec extractor l unused elt index =
    match l with
    | [] -> (List.rev unused, elt :: p)
    | h::t -> if index = rIndex then extractor t unused h (index + 1) else extractor t (h::unused) elt (index + 1)

  in extractor l [] (-1) 0

;;

(**
Genere des permutations aléatoires
params
  n: int
return
  p: list des pemutations
*)
let gen_permutation n =
  let l = ref [] in
  for i=1 to (n) do
    l := !l @ [i]
  done;
  let rec aux l p =
    match l with
    |[] -> p
    |_ -> let (x,y) = (extraction_alea l p) in aux x y
  in aux !l []

;;
```

Figure : Implémentation du générateur aléatoire à l'aide de $EXTRACTION\_ALEA$

Q 1.3 :

La complexité de l'algorithme :

- En nombre d'appel au générateur aléatoire :

  $n + (n - 1) + (n - 2) + \cdots + 1 = O(n)$

- En nombre de filtrage de motif :

  $O(n^2)$

1.2 Synthèse de données améliorée

Vu que la complexité du générateur aléatoire est quadratique, l'invocation de cette dernière sur des donnée très grande est extrêmement coûteuse, pour y remédier on propose une nouvelle implémentation qui s'appuie sur le paradigme diviser pour régner dans le but d'améliorer la complexité

Q 1.4 :

INCERCALE :

$$IN : L1\ , L2 = Listes, avec\ n_1 = L_1.taille\ et\ n_2 = L_2.taille$$

$$OUT : \ L = Liste\ intercalée\ , avec\ L.taille = n1 + n2$$

Avec $\frac{n_1}{n_1+n_2}$ de probabilité, la tête de L reçoit celle de L_1, sa queue est obtenue avec $intercale(L_1.queue\ , L_2)$.

Avec $\frac{n_2}{n_1+n_2}$ de probabilité, le cas est analogue ;

Q 1.5 :

GEN_PERMUTATION2 :

$$IN : p, q : Entiers$$

$$OUT : L = Liste, |L| = q - p + 1\ et\ \forall n \in [p, q]\ \exists i\ \in [p, q] tq\ L[i] = n$$

$Si\ p > q\ alors\ L = [\,]$
$Si\ p = q\ alors\ L = [p]$
$Si\ p < q\ (Intervale\ bien\ défini)\ alors :$

$$L_1 = \left[p, p + 1, \dots, \left[\frac{p+q}{2}\right]\right] et\ L_2 = \left[\left[\frac{p+q}{2}\right] + 1\ , \dots, q - 1, q\right]$$

$L = \ intercale(L_1, L_2)$;

Voici l'implémentation que nous proposons :

```
let intercal l p =
  Random.self_init ();
  let rec aux resultat l p n1 n2 =

    match (l,p) with
    |([],_) -> resultat@p
    |(_,[]) -> resultat@l
    |(h1::t1,h2::t2) -> if (Random.float 1.) < ((float_of_int n1)/.(float_of_int (n1+n2))) then
          aux (resultat@[h1]) t1 p (List.length t1) n2
        else
          aux (resultat@[h2]) l t2 n1 (List.length t2)
  in aux [] l p (List.length l) (List.length p);;


(**
Génére des permutations en faisant appelle à intercal
param
    p: int
    q: int
return
List des permutations générées
*)
let rec gen_permutation2 p q =
  if p > q then
    []
  else if p = q then
    [p]
  else
    let l1 = gen_permutation2 p  ((p+q)/2) in
    let l2 = gen_permutation2 (((p+q)/2) + 1) q in
    intercal l1 l2
;;
```

Figure : Implémentation du générateur aléatoire à l'aide de $INTERCALE$

L'aspect du paradigme « Diviser pour régner » est clair, on partitionne le problème en deux sous problèmes de taille $= \frac{n}{2}$;

<u>Q 1.6 :</u>

La complexité de cet algorithme :

- En nombre d'appel au générateur :
  On remarque que : $T(n) = 2 \times T\left(\frac{n}{2}\right) + n$, en vertu du théorème maître, on obtient $T(n) = \Theta(\mathrm{n} \log n)$

- En nombre de filtrage de motif :
  Pareil, $T(n) = \Theta(\mathrm{n} \log n)$

1.3 Construction de l'ABR

Q 1.7 :

On définit le type TREE comme étant :

- Soit une feuille
- Soit composé d'une racine, d'un sous arbre gauche, ss arbre droit et son degré

INSERT :

$IN : arbre = Tree \, ; key = Entier$

$OUT : arbre = Tree \; (avec\ la\ valeur\ key)$

Ajout simple dans un arbre binaire de recherche ;

CONSTRUCT_TREE_FROM_LIST :

$IN : Arbre = Tree \; (vide) \, ; L = Liste$

$OUT : Arbre = Tree \; (obtenu\ à\ partir\ des\ valeurs\ de\ L)$

Permet de construire l'arbre qui correspond à la liste passée en paramètre ;

On s'appuie sur la récursivité suivante :

Si $L = [\,]$ alors renvoyer $Arbre$

Sinon (si $L = h :: t$) alors renvoyer $CONSTRUCT\_TREE\_FROM\_LIST \; (INSERT\ Arbre\ h)\ t$

2. Compression des ABR

Q 2.8 :

On implémente la fonction φ comme suit :

SIGNATURE :

$IN : A = Tree$

$OUT : \varphi(A) \; \in \{\, (\, , \,) \,\}^*$

On s'appuie sur la définition récursive suivante :

$$\begin{cases} si\ A = feuille\ alors\ \varphi(A) = \epsilon\ (mot\ vide) \\ sinon\ (\ A = (\_\,, G, D)\ )\ alors\ \varphi(A) = (\ \varphi(G)\,) + \varphi(D) \end{cases}$$

En voici l'implémentation :

```
(** Implémentation de la fonction phi
  parms
    tree: ABR
  return
  Chaîne de caractères associé à l'arbre en entrée
*)
let rec signature tree =
  match tree with
  | Empty -> ""
  | Tree(_, g, d, _) -> "(" ^ (signature g) ^ ")" ^ signature d;;


(**
```

Figure : Implémentation de la fonction ϕ

Q 2.9 :

PREFIXE :

$IN:\ \ Abr = Tree$

$OUT: T = Tableau\ des\ étiquettes\ de\ Abr\ dans\ l'ordre\ préfixe$

Principe : Effectuer le parcours préfixe de l’arbre en stockant les valeurs dans T ;

Q 2.10 :

On définit le type arbre compressé CTREE comme soit :

- *Une feuille*
- *Un élément avec sous arbre gauche, droit et son degré*
- *Un arbre avec un tableau d’étiquettes (cas de structure similaire)*

Voici les définitions implémentées de $TREE\ et\ CTREE$ (ABR et ABRCompressed) :

```ocaml
(**
definition recursive du type arbe binaire de recherche polymorphique
Un arbre  | est soit une feuille
          | ou bien il est constitué d'un noeud racine, d'un sous arbre gauche et droit qui sont eux même des ABR
*)

type 'a tree =
  | Empty
  | Tree of  'a * 'a tree * 'a tree * int;;


(**
Definition du type d'arbre compressé
à utiliser dans la partie 2
*)

type 'a ctree =
  | CEmpty
  | CTree of 'a * 'a ctree * 'a ctree * int
  | Compressed of 'a tree  * 'a array
;;
```

Figure : Implémentation des types d'arbre normal et compressé

TREE_COMPRESSOR :   (L'algorithme de compression)

$IN : Abr = Tree$

$OUT : AbrC = Ctree$

L'idée derrière la compression est de faire le parcours préfix de l'arbre et de regarder dans une table associative si le nœud courant a été rencontré ou non.

On propose l'algorithme suivant :

Soient A l'arbre à compresser et H la table associative (Hashtbl) de la forme : $N \longrightarrow \phi(N)$

Faire le parcours préfix et pour chaque nœud N dans le parcours faire

1- calculer sa signature en utilisant la fonction phi, on note S cette dernière.

2- chercher S dans H

3- Si S existe dans H alors le sous arbre en raciné en N a été déjà examiné

-> Alors on crée un nœud $COMPRESSED$ ayant comme paramètre P (le tableau contenant les éléments du sous arbre dans le parcours préfix) et le pointeur vers N

4- Sinon si S n'est pas contenu dans H, dans ce cas alors c'est la 1ere fois qu'on explore le nœud

-> Créer un arbre binaire ayant comme valeur la clé de N, son fils gauche est celui de N (respectivement pour le fils droit).

En sortie, on obtient l'arbre compressé.

L'implémentation est comme suit :

```
(**
Construit l'arbre compressé
params
  tree: ABR
  map: Hashmap, signature ---> noeud correspondant
return
  ABR de type CTree qui est l'arbre compressé
*)
let get_key tree =
  match tree with
  | Empty -> -1
  | Tree(v, _,_,_) -> v

let  tree_compressor tree =
  let map = Hashtbl.create 10 in (*init value, grows according to the number of elements*)
    let rec compressor tree map =
      match tree with
    | Empty
 -> CEmpty
    | Tree(v, g, d, deg) ->
        try
          let s = signature tree in
          let pointer = Hashtbl.find map s in
          if deg > 1 then Compressed(pointer, (prefix tree))
          else Compressed(pointer, (Array.make 1 v))
        with Not_found ->
          Hashtbl.add map (signature tree) tree;
          let left = compressor g map in
          let right = compressor d map in
          CTree(v, left, right, deg)
          in compressor tree map;;
```

Figure : Implémentation de l'algorithme de compression

Q 2.11 : Algorithme de recherche dans l'arbre compressé

SEARCH_IN_COMPRESSED :

$IN : abr = Ctree ; K = Entier$

$OUT : bool = Booléen$

Soit K la clé à chercher,

Se positionner sur la racine et comparer sa clé à K

Si $racine.clé == K\ alors$ retourner vrai ;

Sinon :

Si $racine.clé < K$ alors parcourir son sous arbre droit ;

Sinon ($racine.clé > K$ alors parcourir le sous arbre gauche ;

Lorsqu'on rencontre un nœud compressé, on distingue 4 cas :

Soit E la clé en index 0,

1- si E = K alors retourner vrai ;

2- si E > K alors on avance dans le tableau avec un pas de 1 ;

3- si K > E, alors de ce cas on doit sauter tous les éléments qui sont dans le sous arbre gauche

Et pour remédier à cette étape on utilise le pointeur stocké dans le nœud.

Cela découle du fait que les deux nœuds ont la même structure donc même nombre de nœud dans le sous arbre gauche et droit

Donc on fait un appel à la procédure avec le nouvel $index = index + 1 + get\_deg(G)$ et on avance aussi dans le pointeur afin de se positionner sur G, G étant le fils gauche du pointeur stocker au niveau du nœud courant.

R^{que}: *get_deg() retourne le degré du nœud*

4- Si une exception est levée alors K n'existe pas et on retourne faux ;

- Si à un moment donné du parcours on rencontre une feuille alors K ∄ et on retourne faux

Cet algorithme nous permet de ne traiter qu'une partie des valeurs de l'arbre et ainsi a un ordre de complexité de $O(\log n)$;

L'implémentation est la suivante :

```
(**
Recherche d'un élement dans l'arbre compressé
params
  tree: CTree
return
  bool
*)

let search_in_compressed ctree value =
  let rec find ctree value =
    match ctree with
    | CEmpty -> false
    | CTree(v, g, d, deg) -> if value = v then true else
        if v < value then find d value
        else find g value
    | Compressed(p, arr) ->
        (*rechercher la valeur dans le tableau,
        currentP represente le poiteur ayant la meme structure que l'arbre contenu dans le tableau dans l'ordre prefix
        *)
        let rec finder value arr index currentP =
         (*print_int index;*)
          try
            let current = (Array.get arr index) in
            if current = value then true
            else if current < value then finder value arr (index + 1 + (get_degre (get_left currentP))) (get_left currentP)
            else finder value arr (index + 1) currentP
          with Invalid_argument _ ->
          (*print_string "bom\n";*)
          false
        in finder value arr 0 p
  in find ctree value

;;
```

Figure : Algorithme de recherche dans un arbre compressé

Plus formellement, il s'agit d'une :

Recherche classique dans un BST à l'exception d'un pointeur rencontré à un nœud n : on test sur le tableau T qu'il contient, on commence par $i = 0$ (élément racine de n)

si $T[0] = K$ Alors élément trouvé, renvoyer vrai ;

$si\ T[0] > K$ Alors chercher dans la branche gauche de n : $i = i + 1$

si $T[0] < K$ Alors chercher dans la branche droite $i = |fils\_gauche_n| + 1$

sinon Renvoyer faux ;

Étant donné que T contient les étiquettes dans l'ordre préfixe, on est sûr de n'avoir aucune perte de données.

3. Expérimentations : gains ou perte d'efficacité des ABR compressés

Q 3.13 :

Nous définissons la fonction $TIMEIT$ qui permet de calculer le temps pris par l'exécution d'un algorithme sur une structure donnée comme suit :

```
let timeit func tree key =
  let t = Sys.time() in
  let _ = func tree key in
  Sys.time() -. t
```

Figure : Implémentation de la fonction TIMEIT

Q 3.14 :

Etude de complexité en temps de recherche

Nous comparons (sur la même structure de données) le temps d'exécution pris par l'algorithme de recherche dans l'arbre binaire de recherche normal et celui compressé.

Pour un jeu de tests donné, nous obtenons le graphique suivant :

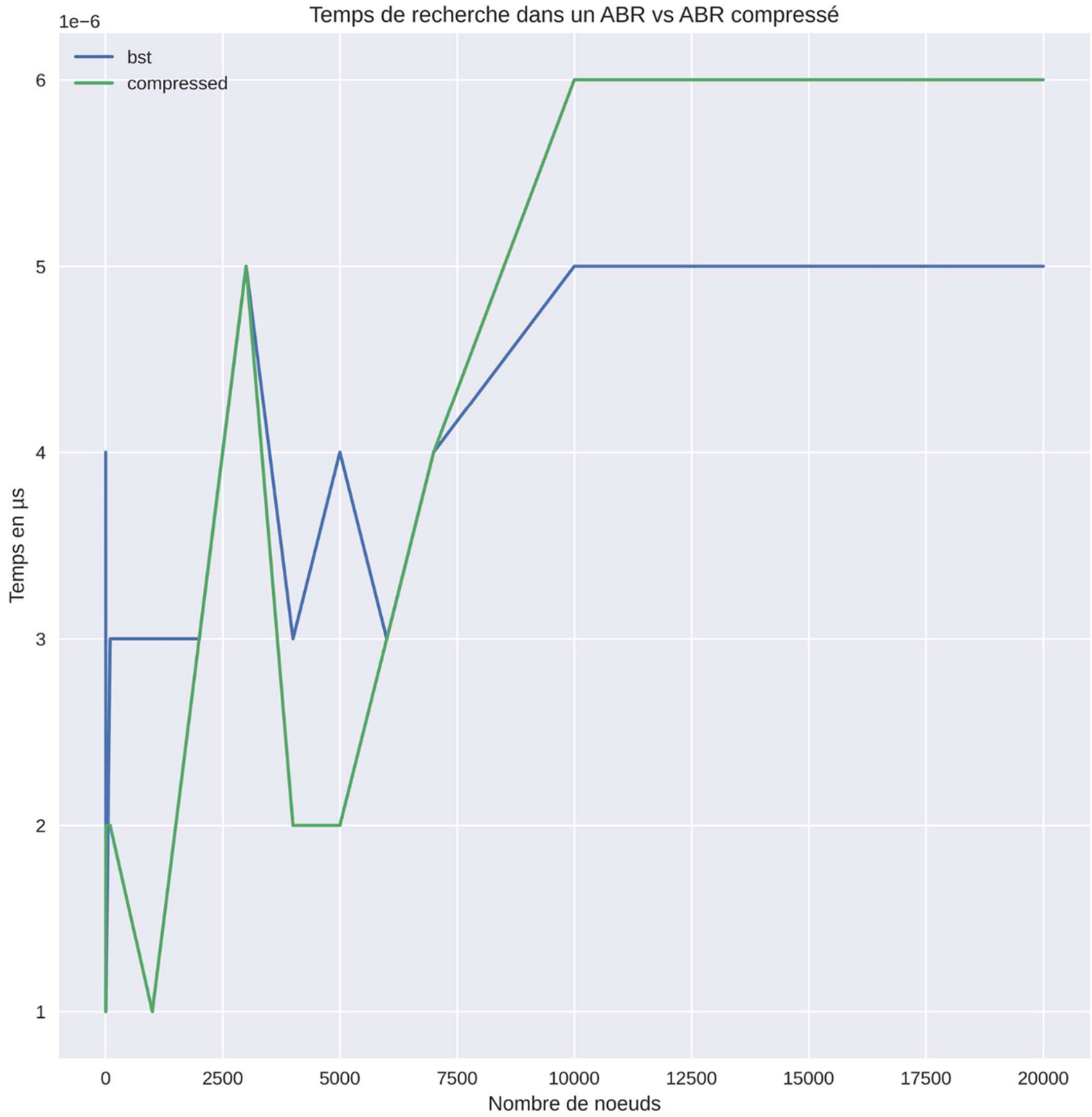

Figure : Temps d'exécution d'une recherche sur un arbre compressé et non compressé

On remarque que pour un nombre de nœuds n :

$n < 7000$: La recherche est équivalente

$n \geq 7000$: La recherche dans un arbre compressé devient plus coûteuse que dans un arbre normal

(6 μs pour un arbre compressé et 5 μs pour un arbre normal au voisinage de $n = 10\ 000$)

NB : Même si la recherche dans un ABRC est plus couteuse, ça reste en $O(\log n)$

Q 3.15 : Etude de complexité en espace mémoire occupée

Nous comparons (sur la même structure de données) l'espace mémoire occupé par l'algorithme de recherche dans l'arbre binaire de recherche normal et celui compressé.

Pour un jeu de tests donné, nous obtenons le graphique suivant :

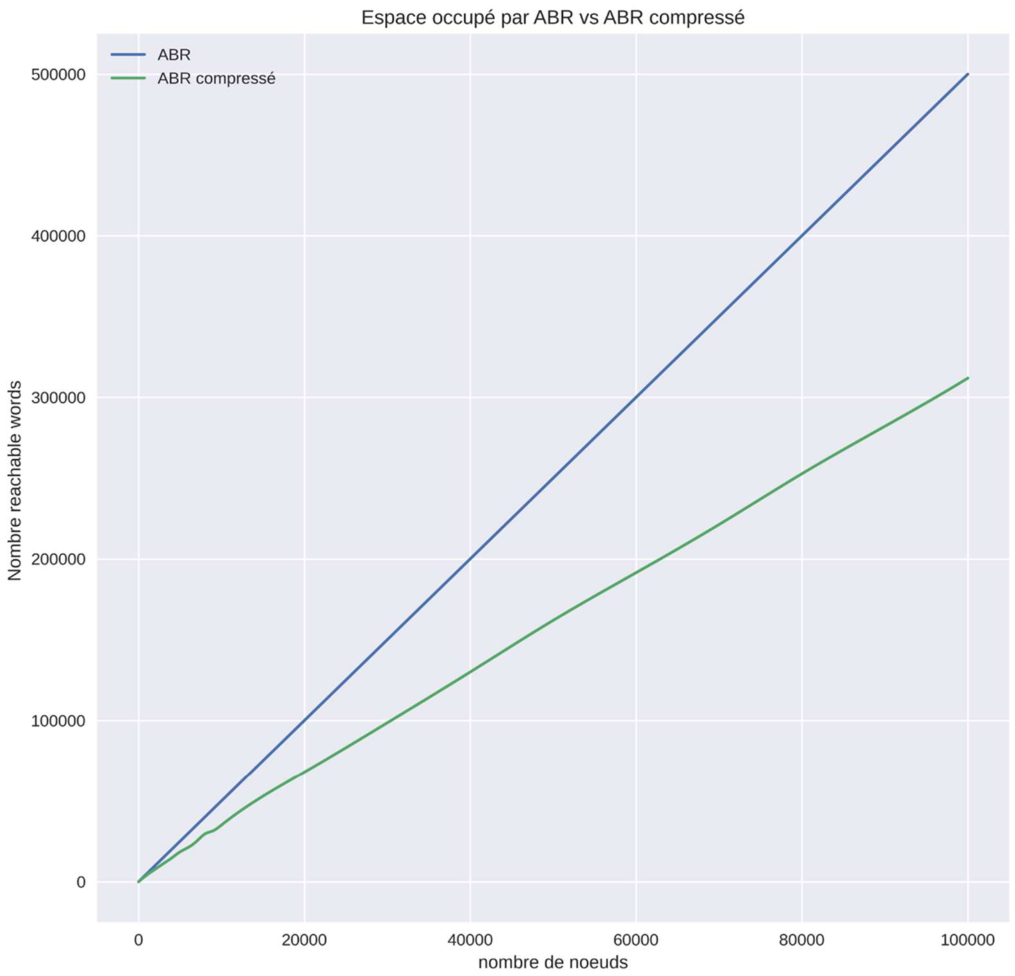

Figure : Espace mémoire occupé d'une recherche sur arbre binaire compressé et non compressé

Comme prévu, l'algorithme de compression permet d'optimiser l'espace mémoire occupé. Le contraire serait inattendu vu que $\forall n$, $\# n_{stockés\ dans\ ABR} < \# n_{stockés\ dans\ ABRC}$.

Conclusion :

L'algorithme de compression optimise l'utilisation de l'espace mémoire en dépit de la recherche qui devient coûteuse $\propto n$, n étant le nombre de nœuds.