Compression d'arbre binaire de recherche
Ouverture

Sorbonne Université

Julien AUCLAIR & Sofiane BELKHIR

Décembre 2020

Table des matières

1 Introduction

Le projet consiste à mettre en place une compression pour les arbres binaire de recherche pour que celui-ci soit plus compacte en mémoire. Un arbre binaire de recherche (ABR) est une structure de données dans laquelle chaque nœud est classé méthodiquement en fonction de sa valeur, de tel manière que chaque nœud du sous-arbre gauche est une valeur inférieure et que chaque nœud du sous-arbre soit supérieure. Ainsi, un arbre binaire de recherche permet des opérations rapides pour rechercher, insérer ou supprimer un nœud. Pour réaliser cette compression, l'idée sera de repérer les structures arborescente identique pour remplacer la deuxième occurrence par un pointeur pointant vers la première.

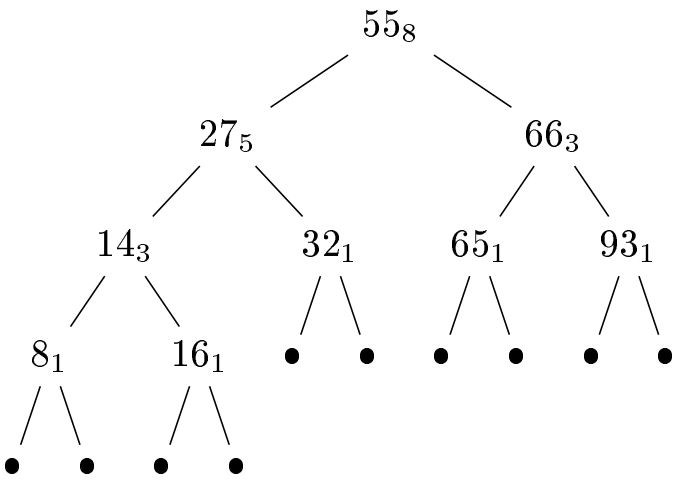

FIGURE 1 – Arbre binaire de recherche

2 Algorithmes de synthèse d'ABR

Dans cette première partie nous allons voir deux algorithmes de mélanges qui vont nous servir à mélanger des listes de valeur pour par la suite nous permettre de construire des arbres de recherche aléatoire pour ensuite les compresser et tester cette compression.

2.1 Algorithme de mélange de Fisher-Yates

Le premier algorithme implémenté ici, est l'Algorithme de mélange de Fisher-Yates cet algorithme va effectuer plusieurs permutations de manière aléatoire pour ainsi mélanger notre liste de valeurs.

2.1.1 Implémentation

Notre fonction appelée ici est "gen_permutation" qui prend en entrée un entier qui est tout simplement la taille de notre liste. Dans un premier temps nous remplissons notre liste de 1 à n. Une fois cela fait, ils nous reste plus qu'a mélanger cette liste, pour cela nous allons utiliser la fonction "extraction_alea" qui va récursivement prendre un élément aléatoirement dans notre liste et l'insérer dans une seconde liste qui commence vide et qui finira remplis de toute nos valeurs mélangées.

```
1  let extraction_alea l p =
2    let len = (List.length l) in
3    let r = (Random.int len) in
4    let rec aux c l1 l2 =
5      match c, l1 with
6      | _, [ ] -> (l, p)
7      | 0, h::t -> (List.rev_append l2 t, h::p)
8      | _, h::t -> aux (c - 1) t (h::l2)
9    in aux r l [ ];;
10
11 let gen_permutation n =
12   let rec aux count list =
13     match count with
14     | 0 -> shuffle  (list, [ ])
15     | _ -> aux (count - 1) (count::list)
16   in aux n [ ];;
17
18 let rec shuffle (l, p) =
19   match l with
20   | [ ] -> p
21   | _ -> shuffle (extraction_alea l p);;
```

Code Source 1 – Implémentation de l'algorithme de mélange de Fisher-Yates

2.1.2 Complexité

La complexité de "gen_permutation" est en $O(n)$ pour le nombre d'appels au générateur de nombre aléatoire et en $O(n^3)$ en nombre de filtrage. Quant au mélange ("shuffle" + "extraction_alea") en nombre d'appels au générateur de nombres aléatoire est $O(n)$ où n représente la taille de la liste, en effet nous faisons la génération d'un nombre aléatoire pour choisir une valeur aléatoire dans la liste et ce jusqu'à ce qu'elle soit vide, soit n fois. Quant à la complexité en nombre de filtrage elle est dans le pire cas en $O(n^2)$ puisque

nous devons parcourir la liste en entière et ensuite pour chaque éléments de cette liste aller jusqu'à l'élément choisis aléatoirement qui dans le pire cas est le dernier à chaque fois.

2.2 Algorithme de mélange amélioré

Le deuxième algorithme de mélange implémenté, va permettre des permutations aléatoire uniformes en s'appuyant sur la technique du "diviser pour régner" qui a pour principe de diviser un problème en plusieurs sous-problème plus simple à résoudre. Ainsi chaque sous-problèmes est résolue séparément puis chaque solutions sont combinées.

2.2.1 Implémentation

Dans cet algorithme, nous commençons par créer deux listes composées des entiers de 1 jusqu'à n en effectuant une permutation. Suite à quoi nous allons venir intercaler ces deux listes en calculant aléatoirement la probabilité d'insérer le premier élément d'une des deux listes et cela jusqu'à ce que les deux listes soient vide. Ainsi nous obtenons notre liste de 1 à n mélangée.

```
let intercale l1 l2 =
  let rec loop acc l1 n1 l2 n2 =
    match l1, l2, n1 + n2 with
    | [ ], [ ], _ ->  List.rev acc
    | [ ], hd::tl, n | hd::tl, [ ], n -> loop (hd::acc) tl (n - 1) [ ] 0
    | hd1::tl1, hd2::tl2, n ->
      if Random.int n < n1 then
        loop (hd1::acc) tl1 (n1 - 1) l2 n2
      else
        loop (hd2::acc) l1 n1 tl2 (n2 - 1)
  in loop [ ] l1 (List.length l1) l2 (List.length l2);;

let rec gen_permutation2 p q =
  if (p > q) then
    [ ]
  else if (p = q) then
    [p]
  else
    let perm1 = gen_permutation2 p ((p + q) / 2) and perm2 = gen_permutation2 (((p +
    ↪  q) / 2) + 1) q in
    intercale perm1 perm2;;
```

Code Source 2 – Implémentation du mélange amélioré

2.2.2 Complexité

La complexité de l'algorithme en nombre d'appels au générateur de nombres aléatoire est $O(n)$ car nous avons n - *1* appels au générateur de nombre aléatoire ce qui donne donc une complexité en $O(n)$. Quant à la complexité en nombre de filtrage elle est en $O(n1 + n2)$ où n1 et n2 représente est la taille de chacune des deux listes, cela revient donc à $O(n)$.

3 Compression

Les possibilités de compression d'un arbre binaire de recherche sont assez limitées, en effet, nous ne pouvons pas enlever de valeur, il faut donc manipuler la structure elle même. Le principe mis en place ici est dans l'idée finalement assez simple, nous allons rechercher dans notre arbre des structures arborescente identique. Lorsque des nœuds ont une structure arborescente identique, nous allons réduire l'arborescence des nœuds identique à un pointeur vers la première occurrence ainsi qu'un tableau contenant les valeurs de cette arborescence sous forme de parcours préfixe.

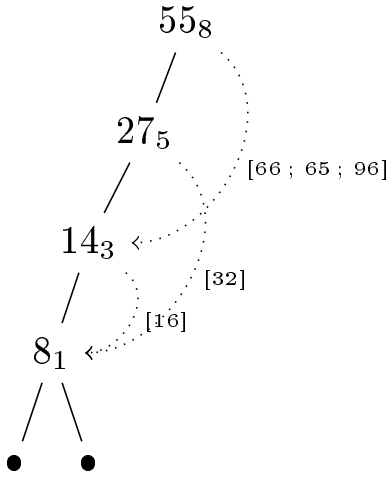

FIGURE 2 – L'ABR de la Figure 1 compressé

3.1 Pseudo code

Algorithm 1: Compression d'un ABR

```
1  Function Compress(tree) :
2      explored ← ∅
3      return CompressAux(tree, explored)

4  Function CompressAux(tree, explored) :
5      if tree.node = Empty then
6          return Empty
7      else
8          left ← CompressAux(tree ⇒ left)
9          right ← CompressAux(tree ⇒ right)
10         s ← size(left) + size(right) + 1
11         p ← Phi(tree)
12         if p in explored then
13             return {explored[p], Prefixe(tree)}
14         else
15             explored[p] ← {tree.node.value, left, right, s}
16             return explored[p]
17         end
18     end
```

3.2 Implémentation

Pour notre arbre compressé nous avons donc soit :
— Un nœud vide
— Un nœud avec une valeur (entier), un sous arbre droit et gauche ainsi que la taille de ses sous-arbre (qui sera indispensable pour la recherche)
— Ou un pointeur vers un nœud et un tableau contenant les valeurs

```
type binary_tree =
  | Empty
  | Node of int * binary_tree * binary_tree;;
type compressed_tree =
  | Empty
  | Node of int * compressed_tree * compressed_tree * int
  | Pointer of compressed_tree ref * int array;;
```

Code Source 3 – Structures utilisées

Pour effectuer cette compression nous allons dans un premier temps faire un parcours suffixe sur l'arbre et donc commencer à visiter les derniers nœuds en premier. Ainsi lorsque nous visitons un nœud nous vérifions dans notre table de hachage si nous avons déjà rencontré une structures arborescente identique à ce nœud, si c'est le cas nous retournons le pointeur stocker dans notre table de hachage ainsi que les valeurs en ordre préfixe de ce nœud sinon nous ajoutons la représentation de cette structures arborescente dans notre table de hachage avec comme valeur un pointeur vers ce nœud et finissons par retourner ce nœud. Nous avons fait le choix ici d'utiliser une table de hachage qui est un container associatif clé-valeur, où ici notre clé sera une chaîne caractère qui représente la structures arborescente d'un nœud et la valeur sera un pointeur vers ce même nœud, ce container nous permet d'avoir un accès en complexité en temps constant.

```
let compress (tree : binary_tree) =
  let table = (Hashtbl.create 5 : (string, compressed_tree ref) Hashtbl.t) in
  let rec aux (tree : binary_tree) =
    match tree with
    | Empty -> Empty
    | (Node (x, left, right)) ->
      let l = aux left in
      let r = aux right in
      let s = (get_size l) + (get_size r) + 1 and p = phi (Node (x, left, right)) in
      if (Hashtbl.mem table p) then
        (Pointer ((Hashtbl.find table p), (prefixe (Node (x, left, right)))))
      else
        let n = (Node (x, l, r, s)) in
        (
          Hashtbl.add table p (ref n);
          n
        )
  in aux tree;;
```

Code Source 4 – Implémentation de la compression

3.3 Expérimentation

L'objectif du projet étant d'avoir un arbre binaire de recherche prenant moins d'espace mémoire, il faut donc s'assurer que cela est le cas avec cette compression. Pour ce faire nous avons utilisé la fonction proposé dans le sujet qui calcule le nombre de mot accessible depuis le pointeur passé en argument. Après quoi nous avons converti ce nombre en Kilo octet. Ces mesures ont été faites sur différente taille d'arbre partant d'un nombre de 10 nœuds jusqu'à 1 000 000 de nœuds.

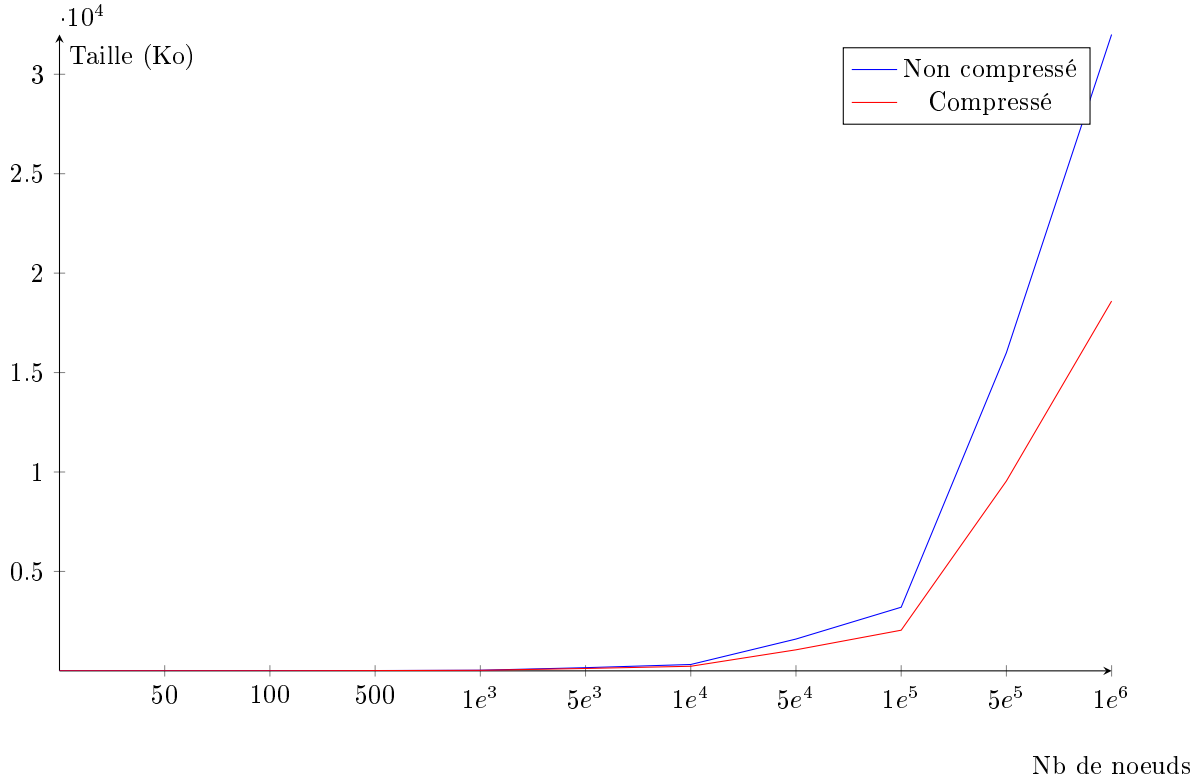

FIGURE 3 – Comparaison ABR non compressé et compressé

Nous pouvons remarquer que la compression est effective seulement lorsque que l'arbre atteint une certaine taille, ici entre 100 et 500 nœuds mais (bien que cela soit difficile à voir ici) en dessus de ce palier la taille de l'arbre "compressé" est plus volumineuse que l'arbre non compressé.

4 Recherche

La recherche est l'opération principale des arbres binaire de recherches. En effet, cette structure de données organise ses valeurs à l'insertion pour permettre une recherche rapide par la suite, il est donc important ici qu'après compression de notre arbre celui-ci permette toujours une recherche efficace.

4.1 Pseudo code

Algorithm 2: Recherche dans un ABR compressé

```
1  Function Search(tree, x) :
2      if tree.node = Empty then
3          return false
4      else if isPointer(tree.node) then
5          return SearchAux(tree, x, tree.labels, 0)
6      else
7          if x = tree.node.value then
8              return true
9          else if x > tree.node.value then
10             return Search(tree ⇒ right)
11         else
12             return Search(tree ⇒ left)
13         end
14     end

15 Function SearchAux(tree, x, labels,i) :
16     if tree.node = Empty then
17         return false
18     else if isPointer(tree.node) then
19         return SearchAux(tree ⇒ node, x, labels, i)
20     else
21         if x = labels[i] then
22             return true
23         else if x > labels[i] then
24             return SearchAux(tree ⇒ right, x, labels, (i + 1))
25         else
26             return SearchAux(tree ⇒ right, x, labels, (i + size(tree ⇒ left) + 1))
27         end
28     end
```

4.2 Implémentation

Nous avons 3 cas possibles pour la recherche dans un arbre binaire soit le nœuds est vide dans ce cas là nous retournons "false", soit il s'agit d'un nœud classique dans ce cas si la valeur du nœud est celle que nous recherchons nous retournons "true" sinon valeur recherche est plus petite que celle du nœud dans ce cas nous explorons le sous arbre gauche dans le cas contraire nous explorons le sous arbre droit.

Notre troisième et dernier cas est plus subtile, en effet lorsqu'il s'agit d'un pointeur, nous ne pouvons pas juste parcourir les sous arbres mais nous allons faire tout comme, en plus de cela nous allons parcourir le tableau qui contient les valeurs stockées dans un ordre préfixe. Sachant que les valeurs sont en ordre préfixe cela nous facilite le parcours, nous commençons a comparer la valeur rechercher avec la première valeur du tableau si la valeur attendu est plus petite dans ce que cas là, nous devons incrémenter l'index de 1 puisque le sous arbre gauche est parcourus en premier dans un parcours préfixe et dans le cas où la valeur est plus grande nous incrémentons de 1 qui correspond au nœud actuel à quoi nous additionnons la taille du sous

arbre gauche ainsi nous arriverons dans le tableau à l'index correspondant au sous arbre droit et ainsi de suite jusqu'à ce que la valeur soit trouvée ou non.

```
let rec search_compressed (tree: compressed_tree) (n : int) : bool =
  match tree with
  | Empty -> false
  | (Node (x, left, right, _)) ->
    if n = x then
      true
    else if n < x then
      search_compressed left n
    else
      search_compressed right n
  | (Pointer (ptr, array)) ->
    let rec aux tree l i =
      match tree with
      | Empty -> false
      | (Node (x, left, right, s)) ->
        if n = (Array.get l i) then
          true
        else if n < (Array.get l i) then
          aux left l (i + 1)
        else
          aux right l (i + (get_size left) + 1)
      | (Pointer (ptr, array)) -> aux !ptr l i
    in aux !ptr array 0
```

Code Source 5 – Implémentation de la recherche dans un ABR compressé

4.3 Complexité

La complexité de la recherche est dans le pire cas en $O(n)$ dans le cas où les nœuds seraient tous situés dans le sous arbre gauche ou droit (donc inséré en ordre croissant ou décroissant) et que nous rechercherions le nœuds le plus profond, nous parcourerions donc l'arbre en entier. La complexité en moyenne est quant à elle en $O(\log n)$, en effet la recherche dans un arbre compressé s'effectue de la même manière que dans un arbre non compressé, la seule chose qui diffère est le faite qu'il faut récupérer la taille du sous arbre gauche dans le cas où nous devons aller visiter le sous arbre droit lorsque la valeur est plus grande, mais cela est en $O(1)$.

4.4 Expérimentation

Après nous être assuré que l'arbre compressé était efficace en terme de complexité en espace, il nous faut maintenant savoir si la recherche d'une valeur dans ce même arbre reste tout aussi efficace en terme de complexité de temps. Le temps d'une seule recherche étant minime, nous avons donc fait 1 000 000 de recherche en boucle en prenant à chaque fois une clé aléatoire. Les temps sont exprimés en milliseconde et ont été mesurés sur des taille d'arbre différente allant comme pour l'Expérimentation de la compression d'un arbre de 10 nœuds à 1 000 000 de nœuds.

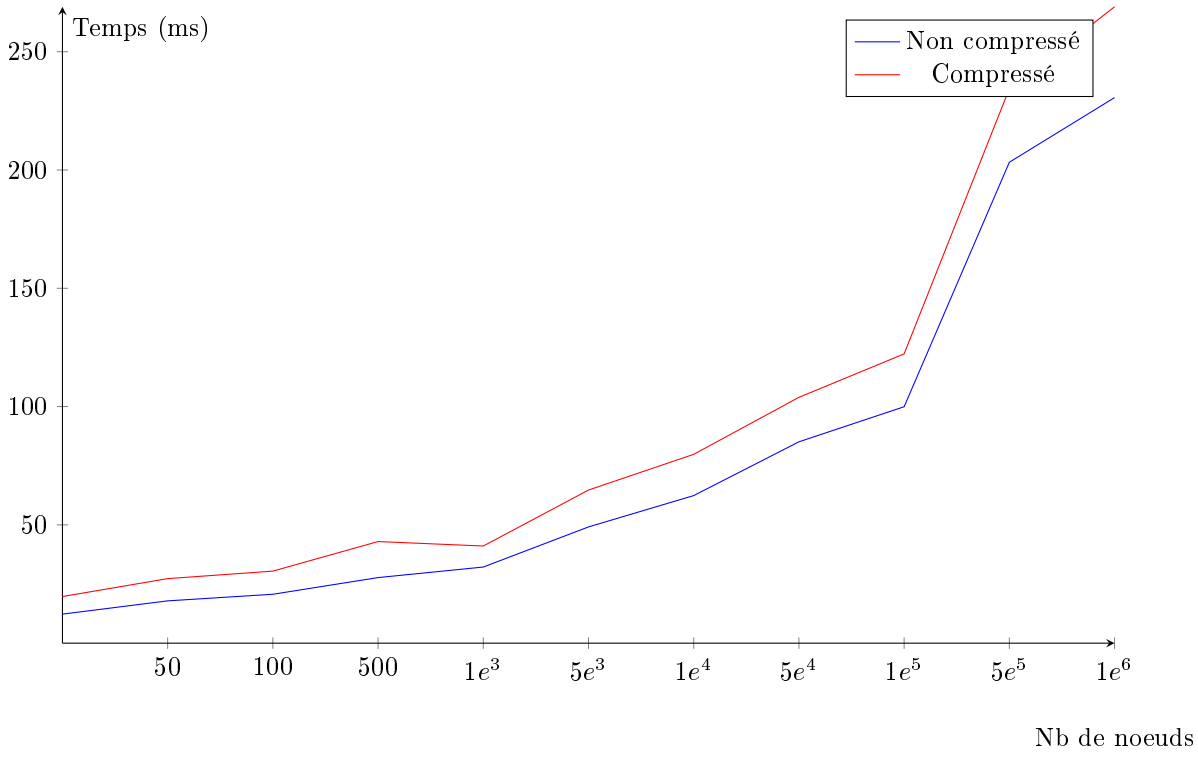

FIGURE 4 – Comparaison recherche ABR non compressé et compressé

Pour avoir une comparaison la plus fidèle possible, nous avons dans un premier temps générer une liste composée de 1 000 000 de nombres aléatoire pouvant aller de 1 à 2 fois la taille de l'arbre (choix arbitraire qui nous permettra d'avoir la possibilité d'avoir des clés présentent ou non dans l'arbre), qui nous servira comme clé à chercher dans les arbres. Dans un second temps nous construisons notre arbre composé de x nœuds en utilisant l' Algorithme de mélange amélioré. Ces premières étapes permettent, d'effectuer des recherches de clé identique dans des arbres eux aussi identique et ainsi éviter qu'une recherche ne soit avantagée par la construction d'un arbre différent.
Ensuite, dans un troisième et dernier temps nous avons plus qu'à parcourir notre liste de clé aléatoire et effectuer la recherche de ces mêmes clés tout en mesurant le temps que cela prend. Et cela pour chaque arbre de taille différente.

Ainsi en observant, le graphique de la Figure 4 on peut s'apercevoir que la recherche dans un arbre binaire compressé est légèrement plus conséquent que dans un arbre binaire non compressé.

5 Conclusion

Pour conclure, nous avons pu observer grâce aux différentes expérimentations que la compression est plutôt probante. En effet, bien que celle-ci ne l'est pas sur de petit arbre, elle est reste tout de même très intéressante sur des arbres plus volumineux et obtient des résultats entre 1.5 et 2 fois plus compacte en mémoire. Quant à la recherche les temps mesurés peuvent être quelques peu décevant car en effet les temps pour un arbre binaire de recherche compressé sont légèrement plus long que dans un arbre non compressé, de l'ordre de quelque milliseconde à plusieurs dizaine de milliseconde pour les arbre plus volumineux.