Rapport Ouverture

Arbres de Recherches Binaires Compressés

Réalisé par :
HADJEBAR Rabah
LOUNIS Lyes
LEFFAD Karim

27 novembre 2019

Table des matières

Chapitre 1

Introduction

Dans le cadre de ce projet, la problématique d'occupation mémoire dans le stockage de données arborescente nous a été introduite. En effet, les structures arborescentes ont, par nature, une taille mémoire linéaire par rapport au nombre d'éléments *(n + 2n ref)*.

La solution qui nous a été proposé pour répondre à cette problématique, est de factoriser les sous-structures arborescentes qui ont la même topologie. Du fait de la structure récursive des arbres, il existe beaucoup de ressemblances typologiques entre les sous-structures ce qui permet d'effectuer une factorisation efficace et offre la possibilité de créer un algorithme de compression efficace.

Grâce à cette méthode de compression, nous pouvons espérer diminuer l'occupation mémoire des structures arborescentes.

Cependant, l'intérêt des arbres est le temps de recherche qui assez faible en moyenne, en effet les arbres binaire de recherches ont un temps de recherche qui est en O(log(n)). Ainsi cet algorithme de compression ne doit pas trop affecter ce temps de recherche.

Tout ce projet consiste à implémenter différentes versions de cette compression en OCaml(une version utilisant des Liste et une autre avec des map) et d'étudier les résultats obtenus sur l'efficacité de cette compression.

Chapitre 2

Arbres de recherches Binaires

2.1 Structure

```
type arb = Empty | Node of int * arb * arb ;;
```

2.2 Construction

On construit notre arbre binaire de recherche par ajout successifs en feuille à partir d'une liste reçu en paramètre :

1. On construit un nouveau noeud ayant comme clé la valeur de notre élément, Empty pour les deux fils gauche et droit.
2. Pour chaque élément de la liste on insert celui-ci à la feuille de l'arbre.

De cette façon l'insertion se fait toujours au niveau des feuilles et pour une liste donnée il y a qu'un seul arbre possible a construire.

2.3 Recherche

La recherche d'un élément dans l'arbre se fait de la manière suivante :

— si la racine = vide on renvoie false
— si val(racine) = élément on renvoie true
— si val(racine) < élément on continue la recherche dans le sous arbre gauche sinon on continue dans le sous arbre droit.

2.4 Complexité

La complexité de recherche est en O(h) où h est la hauteur de l'arbre

— Dans le pire cas (liste triée) la hauteur égal n –> O(n)
— Dans le cas moyen (arbre équilibré) la hauteur tend vers log(n) –> O(log (n)).

Chapitre 3

Compression des arbres

Dans ce chapitre nous allons présenter la méthode suivie pour compresser un arbre binaires de recherches.

3.1 Structure de l'arbre compressé

```
type ('a,'b) pointeurAbr = Null | Pointeur of ('a, 'b) abrCompresse ref
and ('a,'b) abrCompresse = EmptyCL | NodeC of ('a, 'b) contentNode
and ('a,'b) contentNode  = {
    mutable valuesNode: 'a;
    labelArcsRouge: ('b * 'b);
    abrGauche: ('a,'b) pointeurAbr ;
    abrDroite: ('a,'b) pointeurAbr
};;
```

On a choisi un enregistrement pour représenter notre arbre compressé, pour sa simplicité et la lisibilité du code (abrGauche=...,abrDroit=...). De plus, grâce au mot clé mutable on peut modifier un nœud donc la construction est beaucoup plus facile et ça coûte beaucoup moins en temps (on ne reconstruit pas tout l'arbre a chaque modification d'un noeud).

— contentNode est le seul champ qui est mutable, on modifie ce champ quand on ajoute une valeur et son label par contre les autres champs (abrGauche, abrDroite, labelArcRouge) seront fixés à la construction du nœud et on a pas besoin de les modifier.

— labelArcRouge contient un couple, le premier correspond au label du fils gauche, le deuxième au filsDroit.

Enfin notre structure est générique on l'utilise à la fois pour la version avec liste et map (plus de détaille dans les sections liste et map).

3.2 Avec les listes

Avant de commencer à décrire la méthode suivie, on va rappeler d'abord la manière avec laquelle on associe une chaîne de caractères à un arbre binaire, ainsi que la condition pour que deux arbres sont identiques au sens de la structure arborescente.

3.2.1 Structure arborescente d'un arbre binaire de recherche

on va associer à chaque arbre une chaîne de caractères construite sur l'alphabet $\{(,)\}$, via la construction suivante. Soit A l'arbre binaire à compresser, et φ la fonction suivante :

— Si A est réduit à une feuille, on lui associe le mot vide ε (ainsi $\varphi(\bullet) = \varepsilon$) ;

— Si A a un nœud interne et deux enfants G et D qui sont des arbres binaires alors on associe :

$$\varphi(A) = (\varphi(G))\varphi(D)$$

L'algorithme implémenté pour répondre à ce besoins est comme suit :

```
let chaine_caractere_associe_abr abr=
  let rec loop abr =
```

```
    match abr with
    |Empty->[]
    |Node(cle,g,d)->["("] @ loop g @ [")"] @ loop d
  in String.concat "" (loop abr)
;;
```

Pour savoir si deux arbres sont identiques au sens de structures arborescentes il suffit de comparer les deux chaînes de caractères. Donc on aura l'algorithme suivant

```
let isEqv abr1 abr2=
    chaine_caractere_associe_abr abr1 = chaine_caractere_associe_abr abr2
;;
```

3.2.2 Description de la méthode de compression

D'abord, on commence par définir notre a' et b' de notre structure généralisée ; on prend le b' comme étant un entier pour représenter nos labels(b'=int) et a' comme étant une liste de couples(valeur, liste de labels),c'est-à-dire a'=(int, int list).
L'idée de l'algorithme est la suivante :

1. On initialise comme variables globales un compteur de label "*labelCpt*", et un label par défaut pour dire y a pas de label "*noLabel*".
2. On construit une liste de structures arborescente déjà vues qu'on appellera "*seen*", comme étant liste de couple de chaîne de caractères et référence à l'arbre correspondant.
3. On fait à parcours postfixe sur notre arbre de départ, et pour chaque nœud :
   (a) Si c'est une feuille (Empty) on renvoie(noLabel, EmptyCL)
   (b) Sinon
      i. On associe une chaîne de caractères au nœud, et on vérifie si on a déjà vu une telle structure avec une fonction qui prend en paramètres la chaîne de caractères et notre liste *seen*.
      ii. Si la structure n'est pas déjà vu on fait un appel récursive sur le fils droite et le fils gauche. Et et on insère la nouvelle structure dans la liste *seen* avec référence vers le nœud.
      iii. Sinon on génère avec nouveau label et on insère le contenu de l'arbre dont la racine est ce nœud qu'on est entrain de visiter, dans la référence renvoyer par la liste *seen* à l'aide d'une fonction la liste *seen* à l'aide d'une fonction *insertCompresse* qui prend en paramètre la référence, l'arbre à insérer et le label, et rajoute à la liste de nœud d'arbre référencé la valeur ainsi que le label.

3.2.3 Algorithme

```
let getArbreCompresseListeFromAbr arbre =
  let seen = ref [] in
    let rec aux (abr: arb)=
      match abr with
      | Empty -> ( noLabel, EmptyCL )
      | Node (value , g, d ) ->
        begin
          let chaine = (chaine_caractere_associe_abr abr) in
          let (hs, pointeurABR) = (haveSeen chaine !seen) in
          begin
            if hs then
              let tempL = (newLabel ()) in
                (tempL , (insertCompresse pointeurABR abr tempL) )
            else
              begin
                let (labelG, abrCompG) = (aux g )
                and (labelD, abrComprD) = (aux d ) in
```

```
                    let node = getNewValueNodeList value labelG labelD abrCompG abrCompD in
                      let pointeurNode = getNewPointeurAbrFromAbrCompresseList node in
                      (insertToHaveSeen chaine pointeurNode seen) ; (noLabel, pointeurNode )
                end
            end
          end
        in let (_, res) = aux arbre in getAbrCompresseListFromPointeurAbr res
    ;;
```

3.3 Avec la Map

Dans cette partie nous parlons de la méthode suivie pour compresser un arbre avec une map. Notre méthode est quasiment similaire à celle avec une liste, on présentera les points qui diffèrent de la première méthode.

3.3.1 La structure

On prend le b' comme étant une chaîne de caractère pour représenter nos labels(b'=string) et a' comme étant une map : la clé est de type string qui représente le label, et la valeur est de type int.

3.3.2 La variable global labelCpt

On initialise labelCpt a 0 (let labelCpt = ref 0) Dans la fonction newLabel on incrémente labelCpt et on retourne la valeur de ce dernier convertit en string.

Chapitre 4

Recherche

Après la compression de l'arbre, dans cette partie nous présentons l'algorithme de recherche avec les deux structure liste et map.

4.1 La recherche avec la liste

4.1.1 Description de l'algorithme

L'idée générale de l'algorithme de recherche dans un arbre compressé avec les listes est du même principe que celui des arbre binaires de recherche standard avec quelques extensions. La méthode de recherche est la suivante :

1. L'algorithme prend en entrée l'arbre compressé et une valeur
2. Au début de l'algorithme on associe une liste de labels à vide à la valeur donnée en entrée
3. On accède à la racine de l'arbre, et on procède de la manière suivante :
   (a) Si la liste de labels sauvegardé pendant le chemin de recherche est vide([ ]), alors notre élément est sûrement le premier dans le nœud, vu qu'il est originaire dans le nœud(puisque on a une liste de labels vide => pas d'arcs rouges)
      i. Si le contenu du premier nœud = valeur,alors on renvoie *Vrai*
      ii. Sinon si le contenu < valeur, on fait un appel un appel récursive avec les paramètres *arbDroite*, valeur et la liste des label initial concaténée avec le contenu du label *arcRougeDroite* si c'est pas vide(*noLabel*)
      iii. Sinon c'est le même cas que précédemment avec juste maintenant *arbGauche*.
   (b) Sinon on fait une recherche sur notre liste de labels dans la liste contenue dans le nœud
      i. Si le contenu de la case avec la même liste de la labels que celle du chemin pris, est égal à valeur, alors on renvoi *Vrai*
      ii. Sinon de la même façon on compare le contenu avec la valeur qu'on cherche pour prendre faire un appel récursive, soit le chemin gauche ou droit avec le rajout du label à notre liste de labels.
4. On s'arrête si on arrive à une feuille de l'arbre et renvoie *Faux*

4.1.2 Algorithme implémenté avec *OCaml*

```
let rechercheArbreCompresse arbreCompresse  (elem : int) : bool =
  let rec aux abr labels : bool =
    match abr with
    | EmptyCL -> false
    | NodeC(content) -> match labels with
      |[] ->
          let valNoeud = fst (getFirstElement content.valuesNode )
          and lg = fst content.labelArcsRouge
          and ld = snd content.labelArcsRouge in
          if valNoeud = elem then true
```

```
            else
              if valNoeud < elem then
                aux (getAbrCompresseListFromPointeurAbr
                content.abrDroite) (if ld = noLabel then [] else [ld] )
              else
                aux (getAbrCompresseListFromPointeurAbr
                content.abrGauche) (if lg = noLabel then [] else [lg] )
        | _ ->
            let (found, indiceElement) = (elementIsInList
            content.valuesNode elem labels) in
            if found then true
            else
                let valN = fst (List.nth content.valuesNode indiceElement)
                and lg = fst content.labelArcsRouge
                and ld = snd content.labelArcsRouge in
                if valN < elem then
                  aux (getAbrCompresseListFromPointeurAbr
                  content.abrDroite) (labels @ if ld = noLabel then [] else [ld])
                else
                  aux (getAbrCompresseListFromPointeurAbr
                  content.abrGauche) (labels @ if lg = noLabel then [] else [lg])
    in aux arbreCompresse [];;
```

4.1.3 Complexité

Dans cet algorithme de recherche, nous parcourons une seule fois la profondeur de l'arbre soit en $O(log(n))$ visites de noeud. Pour visiter un noeud, nous devons parcourir la liste des valeurs contenues à l'intérieur et comparer pour chacune les labels soit en $O(n)$ comparaisons de labels (au pire cas, le noeud feuille peut contenir $O(n)$ valeurs). La comparaison de labels revient à une comparaison de liste ou de chaine de caractères, de plus nous savons qu'un élément peut avoir une liste d'au plus $O(log(n))$ label donc en $O(log(n))$ comparaisons. Ce qui donne à notre algorithme une complexité temporelle de $O(n * log(n)^2)$

Cependant, nous avons pu remarquer que, du fait de la compression, il n'existe en fait que 1 ou 2 noeuds qui possède cet ordre de noeuds ce qui donne en pratique une complexité qui se rapproche plus de $O(n * log(n))$

4.2 La recherche avec la map

4.2.1 Description de l'algorithme

Le principe de l'algorithme de recherche pour les arbres compressés avec des maps est le même que celui avec les listes, avec quelques modifications vue que maintenant on a plus une liste de labels mais plutôt une chaîne de caractères de labels concaténés.
Pour la recherche de le nœud on utilisé la fonction approprie au module *Map* de **OCaml** : *find* on lui donnant en paramètres la chaîne de caractères de labels et le nœud(autant que map).

4.2.2 Complexité

Le princie de cet algorithme étant le même que celui de recherche en liste, nous avons aussi un seul parcours de la profondeur de l'arbre soit $O(log(n))$ visites de noeud. Pour la visite de noeud cette fois-ci, nous faisons uniquement appel à **Map.find** qui effectue la recherche en $O(log(n)$ comparaisons [1] ce qui donne une complexité totale de $O(log(n)^2)$.

Chapitre 5

Expérimentation

Dans ce dernier chapitre du rapport, nous présenterons brièvement, dans un premier temps, les fonctions utilisées pour le calcul du temps pris par l'exécution d'un algorithme sur une donnée particulière, et pour le calcul d'espace mémoire total consommé par une structure de données.

Et en deuxième lieu on effectuera des études expérimental de complexité en temps et on gain mémoire sur l'algorithme de compression et celui de la recherche.

5.1 Fonction pour le calcul du temps

Pour le calcul du temps pris par l'exécution d'un algorithme sur une donnée particulière, on utilise la fonction "*gettimeofday ()*" du module **Unix** [2] de **OCaml** ; une fonction qui nous renvoie l'heure actuelle avec une résolution supérieure à 1 seconde. Il suffit d'appeler la fonction avant et après l'algorithme, la temps donc c'est juste la différence entre les deux.

Pour les résultats, on les met dans un fichier texte dans le but de les utiliser après avec l'application *Gnuplot*[1]

L'implémentation de la fonction en **OCaml** est la suivante :

```
let calculeTime donnees ... =
  let out_channel = open_out "file" in
  ....
    let start = Unix.gettimeofday () in
    (* Exécution de l'algorithme *)
    fprintf out_channel "%fs \n" (Unix.gettimeofday () -. start);
  ...
  close_out out_channel
;;
```

5.2 Fonction pour le calcul d'espace mémoire

Il existe plusieurs façons pour le calcul d'espace mémoire total consommé par une structure de données, parmi elles on trouve le module **Gc**[3] de **OCaml**. Gc présente plusieurs fonctions, mais dans ce rapport on se limitera à la fonction *"Gc.stat()"* qui nous retourne un enregistrement possédant notamment les champs :

- *"minor_ words"* : le nombre de mots stocké dans le tas mineur
- *"major_ words"* : le nombre de mots stocké dans le tas majeur
- *"promoted_ words"* : le nombre de mots qui on été promu du tas mineur au tas majeur

Le calcul de l'occupation mémoire s'effectue de la façon suivante : *heap_ allocation = minor_ words + major_ words - promoted_ words* Les résultats seront comme pour la fonction précédente fournis dans un fichier attaché à l'archive.

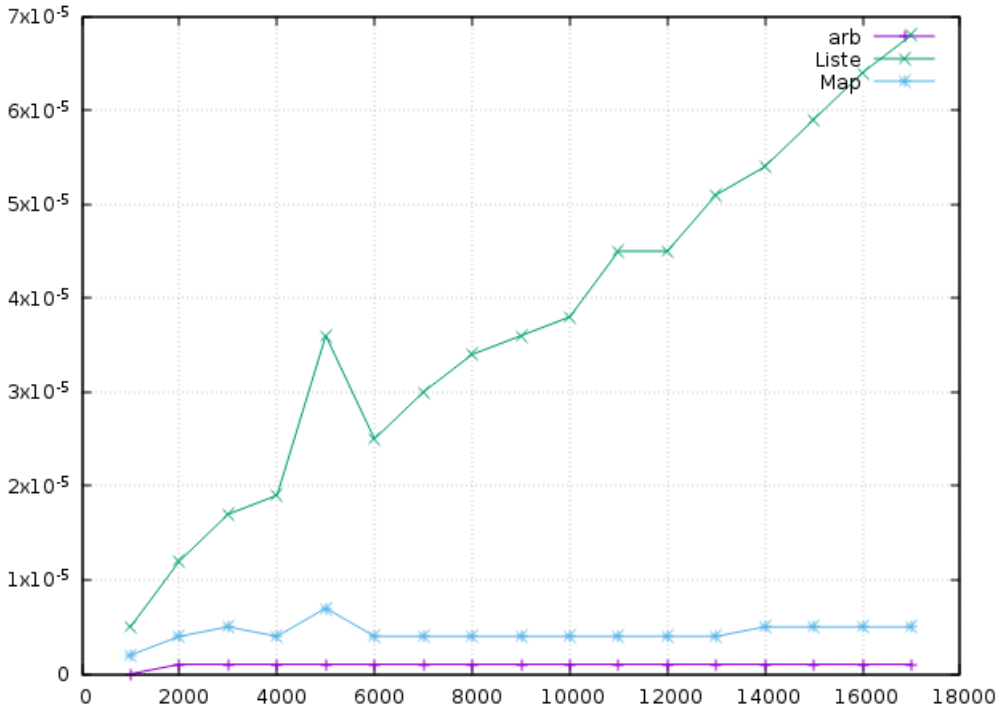
1. Gnuplot est un logiciel qui sert à produire des représentations graphiques en deux ou trois dimensions de fonctions numériques ou de données

L'implémentation est la suivante :

```
let calculeMemory donnees ... =
  let out_channel = open_out "file" in
  ....
    let start = ( (Gc.stat()).minor_words +. (Gc.stat()).major_words
    -. (Gc.stat()).promoted_words ) in
        (* Construction/affectation de la structure de données *)
    let end = ( (Gc.stat()).minor_words +. (Gc.stat()).major_words
    -. (Gc.stat()).promoted_words ) in
    let m =  int_of_float(end -. start) in
    fprintf out_channel "%fs \n" (m);
  ...
  close_out out_channel
;;
```

5.3 Étude expérimentale de complexité en temps des algorithmes de recherche

Dans cette étude, on effectuera (2 x (La taille de l'ABR)) recherche et on prendra à la fin la moyen des temps de recherche de la totalité des recherches ; avec cette approche on garantit *presque* la correction du résultat Pour les tailles des arbres à générer, ça sera des arbre de 1000 à nb * 1000 avec "nb" une donnée pour l'algorithme d'expérimentation. Par exemple avec nb=18 on aura le résultat suivant :

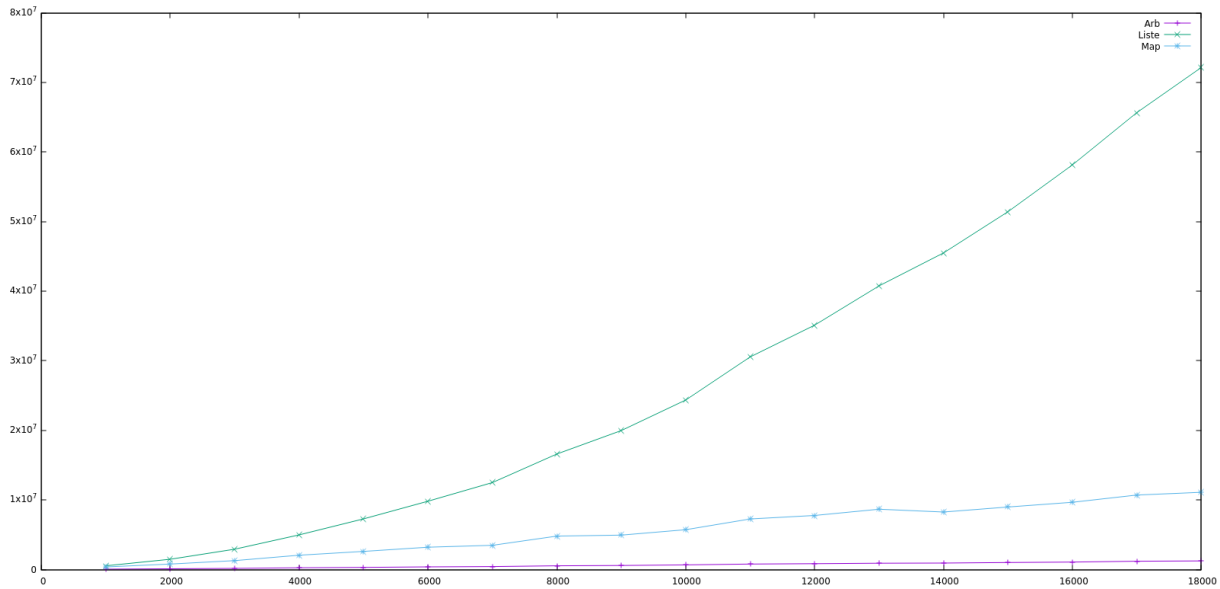


Le temps des algorithmes de recherche

On remarque, avec ce graphe que la complexité de recherche théorique correspond à peu près aux résultats que l'on obtient. En effet avec l'implémentation en liste, On perd beaucoup en efficacité de recherche contrairement à l'implémentation à l'aide d'une Map ou la recherche n'est pas très impactée.

5.4 Étude expérimentale de complexité en espace

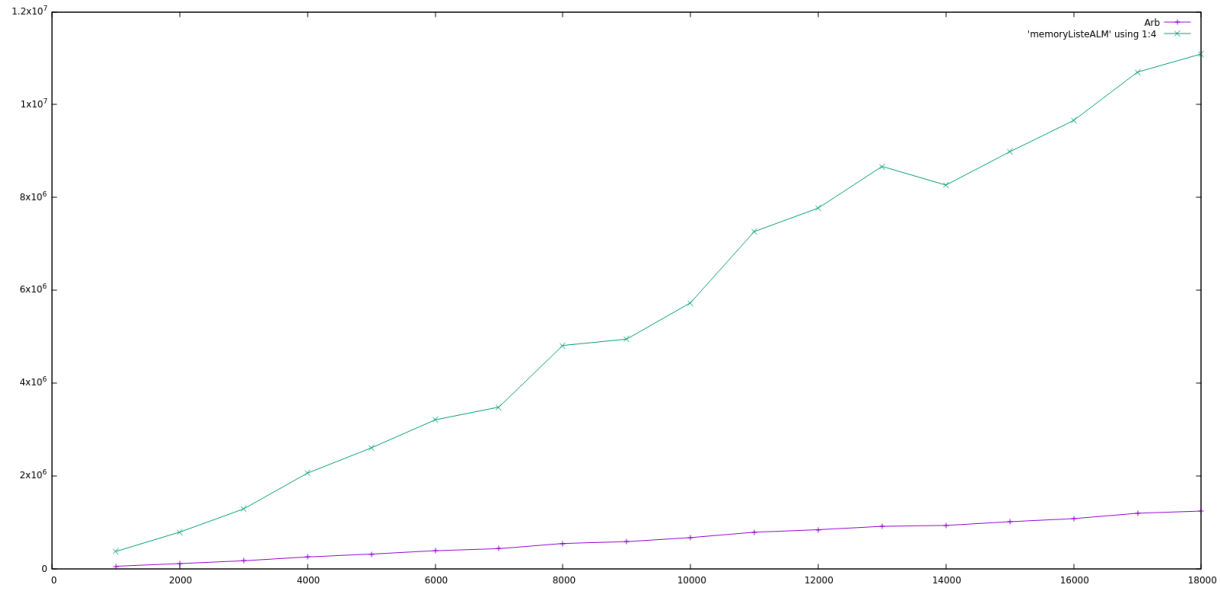
Dans cette partie, on s'intéresse à la consommation mémoire de nos structures de données. Notre étude se fera sur des arbres de recherches de taille 1000 à 180000, et nous étudierons l'occupation dans le tas comme décrit dans la partie précédente.



Mémoire utilisée par structure

Contrairement à nos attentes, la consommation mémoire est du loin supérieure pour un arbre compressé (que ce soit avec des listes ou des maps) que pour un arbre binaire standard.

Dans ce qui suit, on s'intéressera plus aux arbres compressés avec des maps et aux ABR.



Mémoire utilisée par la map et l'ABR

Comme on peut le voir plus précisément dans ce graphe, il y a une grande différence entre notre implémentation des arbres compressés avec des Map et et celle des ABR.

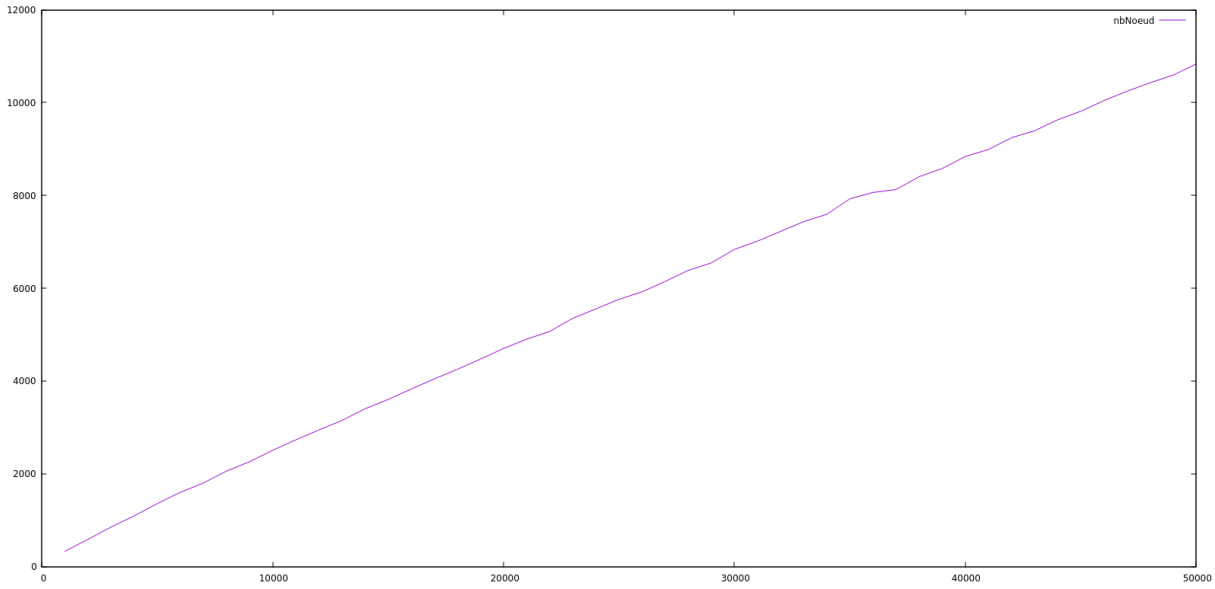
5.5 Étude expérimentale sur le taux de compression

5.5.1 Nombre de nœuds internes

Cette fois-ci, on étudiera de près le nombre de nœuds internes dans les arbres compressés avec des listes obtenus avec le jeu de test donné [2].

Nombres de valeurs	Nombres de noeuds internes
100	100
150	59
500	176
750	258
1000	1000
10000	2497
50000	10846

En plus de cela, nous avons effectuer des tests sur des données aléatoires, présentés dans le graphe ci-dessous.



Nombre de nœuds internes

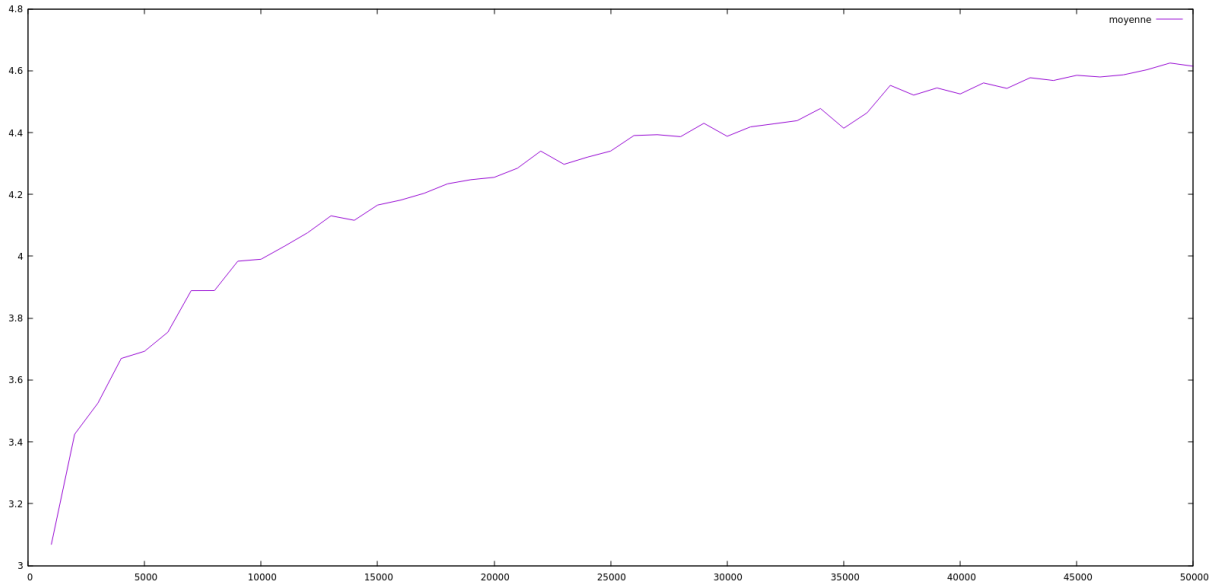
5.5.2 Nombre moyen de clés par nœud interne

On étudiera maintenant le nombre moyen de clés par nœud interne dans les arbres compressés avec des lites, obtenus avec le jeu de test donné.

Nombres de valeurs	Nombres de valeurs par noeud
100	1.0
150	2.5
500	2.8
750	2.9
1000	1.0
10000	4.0
50000	4.6

2. `https://www-apr.lip6.fr/~genitrini/doc_ens/Jeu_de_tests.tar.gz`

Comme précédemment, on a aussi fait d'autres tests sur des arbres aléatoires :



Nombre moyen de clés par nœud interne

Chapitre 6

Conclusion

Durant ce projet, nous avons essayé d'implémenter une compression d'arbre binaire de recherche susceptible d'améliorer les performances mémoires sans trop diminuer les performances en temps de calcul. En effet, théoriquement cette implémentation aurait dû donner des résultats assez clairs mais malheureusement, cela n'a été le cas dans nos expérimentations :

- un arbre binaire de recherche standard de 40000 éléments on a une mémoire de *120000* (n + 2n ref)
- En prenant en compte le nombre moyen de valeurs dans un noeud *moy* et le nombre de noeud total *nb* pour un arbre de taille 40000 on obtient *moy = 4,3* et *nb = 8500*.
- Et la taille de notre structure peut se calculer de cette manière : *nbElem + nbRef + nbLabels* où :
  - *nbElem = nb * moy = 36550*
  - *nbRef = 2 * nb * (ref + label) = 34000* (en effet chaque arc a une référence et un label
  - *nbLabels = n * labels = 520000* où *n* est le nombre d'elements et *labels* sont les labels de chaque éléments soit au pire cas *labels = log(nb)*

  ce qui donne : *memoire = (nb * moy) + 2 * nb (ref + label) + log(nb) * n = 590550*

En regardant ces résultats, on se rend compte que le plus gros problème de mémoire est la façon de stocker les labels correspondant à chaque valeur. En effet, on stocke la liste entière des labels pour chaque valeur. Pour remédier à cela, une idée serait de ne stocker que le label du père et non toute la liste. cette solution diminuerait grandement la mémoire de notre structure car *nbLabels = n* seulement ce qui donnerait un total de *110550*.

Bibliographie

[1] Module map de ocaml. https ://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html.
[2] Module unix de ocaml. https ://caml.inria.fr/pub/docs/manual-ocaml/libref/Unix.html.
[3] Module gc de ocaml. https ://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html.