Sorbonne université

UE Ouverture

Rapport de projet

Laetitia PHAM
Maxence BRUNET
Veyrack LIN

Enseignants : Antoine GENITRINI
Emmanuel CHAILLOUX

27 Novembre 2019

1 Présentation

Dans ce projet, nous essayons de manipuler un modèle de structure de données arborescente, les Arbres Binaires de Recherche (ABR), pour construire une structure compressée suivant une procédure particulière décrite dans la partie deux.

1.1 Synthèse de données

Pour simplifier la synthèse de nos ABR, les données que nous insérons dans l'arbre seront des entiers de 1 à n.

1.1 La fonction *extraction_alea* prend en entrée deux listes d'entiers, notée l et p, et choisit aléatoirement un entier r entre 1 et k (taille de l). Puis elle retourne un couple de listes dont la première est la liste l dans laquelle on a retiré le *r-ième élément* et la deuxième est la liste p dans laquelle on a ajouté le *r-ième élément* extrait de l.

```
let remove e l = List.filter (fun x -> x != e) l;;

let extraction_alea l p = let size = List.length l in
    let r = (int_of_float (Unix.time())) mod size in
      let tmp = List.nth l r in
        ((remove tmp l) , (tmp :: p));;
```

1.2 La fonction *gen_permutation* prend en entrée une valeur entière n et réalise l'algorithme de *shuffle de Fisher-Yates*.

```
let rec interval n m = if n = m
    then [m]
    else n :: interval (n+1) m;;

let gen_permutation n = let l =interval 1 n in
  let p = [] in
    let rec tmp ll pp = if List.length ll = 0
      then pp
      else
          let ext=(extraction_alea ll pp) in
            tmp (fst ext) (snd ext) in
              tmp l p;;
```

1.2 Construction de l'ABR

Un ABR est un arbre binaire dont les nœuds internes sont étiquetés (par des entiers distincts) de tel sorte que la racine possède une étiquette plus grande que toutes les étiquettes du fils gauche, et plus petite que toutes les étiquettes du fils droit. Récursivement, les deux fils de la racine sont des ABR.

1.3 Dans un premier temps, on définit un *type abr* qui représente un nœud de l'ABR.

```
type abr =
  | Node of {etq: int; fg: abr; fd: abr}
  | Empty;;
```

Puis étant donnée une liste d'entiers tous distincts, on construit l'ABR associé à cette liste.

```
let rec inserer e abr = match abr with
  | Empty -> Node {etq = e;fg = Empty;fd= Empty}
  | Node(n) -> if e<n.etq
    then Node {etq=n.etq;fg = (inserer e n.fg); fd = n.fd}
    else Node {etq=n.etq;fg = n.fg ; fd = (inserer e n.fd)};;

let construc l = let empt = Empty in
  let rec aux l abr = if l = []
    then abr
    else aux (List.tl l) (inserer (List.hd l) abr) in aux l empt;;
```

Par exemple, la *figure 1* peut être construit à partir de la liste [4,2,3,8,1,9,6,7,5].

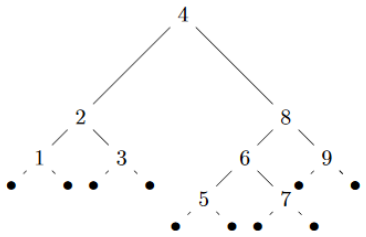


Figure 1: ABR-exemple

2 Compression des ABR

Pour la compression des ABR, nous cherchons à compresser la structure arborescente en repérant les sous-arbres (non réduit à une feuille) ayant une même structure puis en remplaçant la deuxième occurrence du sous-arbre via un pointeur vers le premier sous-arbre (Voir *figure 2*).

2.4 La fonction *parenth* permet de reconnaître si deux arbres (ou sous-arbres) sont isomorphes. En effet, on leur associe une chaîne de caractères construite sur l'alphabet {(,)} de tel sorte que si l'arbre est réduit à une feuille, on lui associe le mot vide "" sinon on lui associe la construction: (chaineFG) chaineFD. Par exemple, pour la *figure 1* on obtient: ((())())((())())().

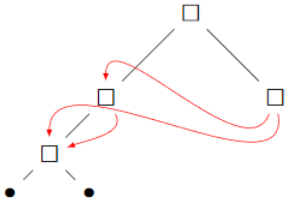


Figure 2: Structure compressé de l'ABR-exemple

```
let parenth abr=
  let rec test tree = match tree with
      |Empty ->""
      |Node(n)->"("^(test n.fg)^")"^(test n.fd) in test abr;;
```

D'après le résultat obtenu par la fonction *parenth*, nous pouvons déterminer la liste des mots que possède l'arbre.

Ancienne Version: Dans un premier temps, nous sommes partis sur l'idée d'associer et de remplacer chaque mot trouvé dans l'arbre par un entier unique, dans le cas où il y a des mots identiques ils seront représentés par le même entier. Par défaut, on associera le mot () (qui représente les feuilles) avec l'entier 1, et les autres mots suivent le pattern: (a) b , tel que a *et* b sont deux entiers.

```
let unique =
  let last = ref 0 in
  fun () -> incr last ; !last;;

let replace expr mat rep = Str.global_replace mat rep expr;;

let rec parcours hash par =
  if (Str.string_match (Str.regexp "^([0-9]+)[0-9]+$") par 0)
    then (Hashtbl.add hash (unique()) (Str.matched_string par); hash)
    else
      let _ = Str.search_forward (Str.regexp "([0-9]+)[0-9]+") par 0 in
        let tmp = Str.matched_string par in
          let key = unique() in
            Hashtbl.add hash key tmp;
            parcours hash (replace par (Str.regexp_string (Hashtbl.find hash key)) (string_of_int key))

let getHash abr =
  let h = Hashtbl.create (getHauteur abr) in
    let uni=unique() in
      Hashtbl.add h uni "()";
        parcours h pa;;
```

En prenant l'exemple de la *figure 1*, on a : $(((())())((())())()) \rightarrow ((1)1)((1)1)1 \rightarrow (2)(2)1 \rightarrow (2)3 \rightarrow 4$. Ainsi la fonction *getHash* retournera une hashtable contenant les mots: *()* associé à la clef 1, *(1)1* associé à 2, *(2)1* associé à 3 et *(2)3* associé à 4.

A cet instant nous nous sommes rendu compte qu'un problème se posait. En effet cette fonction *getHash* ne marche pas pour des arbres contenant des noeuds qui possèdent seulement qu'un fils droit ou gauche. Nous avons donc décidé de délaisser cette fonction et de continuer uniquement avec l'expression parenthésée.

2.5 Désormais nous devons stocker les informations dans les noeuds internes de notre structure, certains noeuds contiendront plusieurs valeurs et plusieurs parents. Pour pouvoir retrouver le parent d'une valeur dans un noeud, nous rajoutons de l'information lorsque l'on passe à travers les arêtes rouges au niveau du parent et du fils. *Voir Figure 3.*

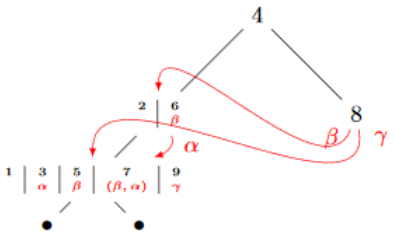


Figure 3: Compression de l'ABR-exemple

Afin d'obtenir une telle compression, via un parcours préfixe, on doit avoir un générateur de symbole unique pour donner un nom aux arêtes rouges.

```
(*Generateur de symbole et son reset*)
let reset_s, generate_symbol = let c = ref 0 in
  ( function () -> c:=0),
  ( function () -> c:=!c+1; "SYMB"^(string_of_int !c) );;
```

Ainsi pour cette première fonction de compression, nous utiliserons une liste pour les couples clés-valeurs dans chaque noeud. D'abord, nous définissons notre nouveau type d'ABR compressé.

```
(*TYPE du noeud d'un arbre compressé*)
type noeud = |Etq of int
             |Couple of {etq : int; liste : string list};;

(*TYPE de l'arbre compressé*)
type comp = |Empty
            |Symbole of string * comp ref
            |NodeC of {etq : noeud list;fg : comp ref ;fd : comp ref };;
```

Pour simplifier notre fonction nous avons crée des sous-fonctions, tels que:
- La fonction *mergeNodes* qui nous permet d'ajouter les clés-valeurs à nos noeuds ou de créer le noeud s'il n'existe pas.
- La fonction *creerFils* qui nous permet de créer une référence vers le noeud fils. Si le lien est indirect, c'est-à-dire si c'est une arête rouge, nous ajoutons un symbole qui nous permettra de retrouver l'étiquette du noeud fils.

```
let mergeNodes h_nodes mot node fg fd symb =
  if (Hashtbl.mem h_nodes mot) then
    let node2 = Hashtbl.find h_nodes mot in let newN = (addInNode !node2 node) in node2:= newN ;
      (symb,h_nodes)
  else
    let newN = (createNode [node] fg fd) in (Hashtbl.add h_nodes mot (ref newN));
      (mot ,h_nodes);;

let creerFils h_nodes s mot= if (Str.string_match (Str.regexp "^SYMB[0-9]+") s 0)
    then ref (Symbole (s, Hashtbl.find h_nodes mot))
    else Hashtbl.find h_nodes mot;;
```

Ensuite, nous avons implémenté notre fonction de compression : *compTree* qui sera appelé par ABR-compress-listes. Celle-ci prend en paramètre un ABR non compressé, une hashtable qui stockera nos structures le temps de la création et finalement une liste de symbole.

```
let rec compTree  abr h_nodes lSymb = let mot = parenth abr in
    match abr with
    | Empty -> if (Hashtbl.mem h_nodes mot)
                then (mot, h_nodes)
                else begin Hashtbl.add h_nodes mot (ref Empty); (mot, h_nodes) end
    | Node(n) -> (    (*Notre ancien noeud*)
     if (n.fg = Empty && n.fd = Empty) (*ici on traite le noeud parent de 2 feuilles*)
        then mergeNodes h_nodes mot (if lSymb = [] then Etq n.etq
                else Couple {etq=n.etq; liste=lSymb}) (ref Empty) (ref Empty) (if lSymb=[]
                                                                    then "" else (List.hd lSymb))
        else if (Hashtbl.mem h_nodes mot) (*Noeud existe deja*)
            then let tmp = Hashtbl.find h_nodes mot in match !tmp with
                |NodeC(n2) ->  let _ = (match !(n2.fg) with
                                        |Empty -> compTree n.fg h_nodes lSymb
                                        |Symbole(s,h) -> compTree n.fg h_nodes (s::lSymb)
                                        |NodeC(n3) -> compTree n.fg h_nodes lSymb) in
                               let _ = (match !(n2.fd) with
                                        |Empty -> compTree n.fd h_nodes lSymb
                                        |Symbole(s,h) -> compTree n.fd h_nodes (s::lSymb)
                                        |NodeC(n4) -> compTree n.fd h_nodes lSymb) in
                            mergeNodes h_nodes mot (Couple{etq=n.etq; liste=lSymb}) (ref Empty)
                               (ref Empty) (if lSymb=[] then "" else (List.hd lSymb))
                  |_ -> ("Err",h_nodes)
                else (*Noeud n'existe pas*)
                let fg = compTree n.fg h_nodes (if Hashtbl.mem h_nodes (parenth n.fg)
                                                 then generate_symbol()::[] else lSymb ) in
                  let fd = compTree n.fd h_nodes (if Hashtbl.mem h_nodes (parenth n.fd)
                                                then generate_symbol()::[] else lSymb ) in
                mergeNodes h_nodes mot (Etq n.etq) (creerFils h_nodes (fst fg)
                    (parenth n.fg)) (creerFils h_nodes (fst fd) (parenth n.fd)) ""  );;
(*Fonction Principale qui compresse l'arbre*)
let ABR-compress-listes abr = let nodes = Hashtbl.create (getHauteur abr) in
    let c = compTree abr nodes [] in !(Hashtbl.find (snd c) (parenth abr)) ;;
```

2.6 Pour la fonction de recherche, nous fournissons un ABR compressé ainsi que l'élément recherché. Pour chaque noeud de l'arbre nous regardons son étiquette qui est une liste de Noeud. Pour cela nous utilisons une fonction auxiliaire *checkList* qui va vérifier si l'élément *e* est contenu dans cette liste, si oui nous retournons *true* sinon nous utilisons le résultat de la sous fonction pour savoir s'il faut chercher dans le sous arbre gauche ou droit.

```
(*Check si e est dans la liste l
renvoi 0 si oui,sinon si e < à l'etiquette 1, -1 sinon*)
let rec checkList l e = match l with
  | [] -> 2
  | h::t -> match h with
    | Etq(n) -> if n=e then 0 else if n<e then 1 else -1
    | Couple(n) -> if e=n.etq then 0 else checkList t e


(*Recherche a partir du noeud compT si e est contenue dans l'arbre*)
let rec search compT e = match compT with
  | Empty -> false
  | NodeC(n) -> let tmp = checkList n.etq e in
    if tmp = 0
        then true
        else if tmp = 1
            then search !(n.fd) e
            else search !(n.fg) e
  | Symbole(n,c) -> search !c e;;
```

La fonction *checkList* réalise une recherche séquentiel sur la liste du noeud, la complexité est donc linéaire. La fonction *search* parcours chaque noeud dans lequel nous utilisons pour chacun d'entre eux la fonction *checkList*. Le parcours d'un ABR est en O(log(n)) mais si l'arbre est dégénéré et qu'il possède que des fils gauches (ou droits), le parcours de l'arbre ce fait en O(n) mais la liste d'un noeud sera composé d'un seul élément. Nous obtenons donc une complexité dans le pire de cas en $O(n)$.

2.7 Pour notre seconde fonction de compression : *ABR-compress-maps*, nous avons créé un nouveau type où à la place d'une liste de noeud nous avons mis une hashtable pour stocker de façon plus efficace l'ensemble des clés.

```
(*TYPE de l'arbre compressé map*)
type comp_map =
    |Empty
    |SymboleM of string * comp_map ref
    |NodeM of {etq : (int, string list) Hashtbl.t ;fg : comp_map ref ;fd : comp_map ref};;
```

Notre fonction ABR-compress-maps est aussi similaire à notre fonction sur les listes, nous l'avons juste adapté à notre nouveau type. (*cf abr.ml*).

2.8 Pour la recherche dans les arbres compressés-map nous n'avons pas réussi à récupérer les clés des hashtable qui nous aurait permis de diriger la recherche. En outre, nous regardons de manière naïve le fils gauche puis le fils droit afin d'examiner leur hashtable associée si c'est un noeud compressé, puis nous testons avec la fonction *mem* des Hashtbl pour connaître l'existence de la valeur recherchée.

```
(*Fonction de recherche dans un arbre compressé map*)
let rec searchMap compT e = match compT with
  | EmptyM -> false
  | NodeM(n) -> if (Hashtbl.mem n.etq e)
    then true
    else
        if not (searchMap !(n.fg) e)
            then searchMap !(n.fd) e
            else true
  | SymboleM(n,c) -> searchMap !c e;;
```

Dans le pire cas, nous avons un arbre dégénéré. Dans ce cas là, la fonction de recherche va parcourir tous les noeuds. Ceux-ci contiennent des hashtables vides étant donné que l'ABR n'est pas compressé. On effectue donc au maximum n parcours.

3 Expérimentation: gains ou perte d'efficacité des ABR compressés

3.1 Analyse

3.10 La fonction permettant de calculer le temps d'exécution de notre algorithme est la suivante:

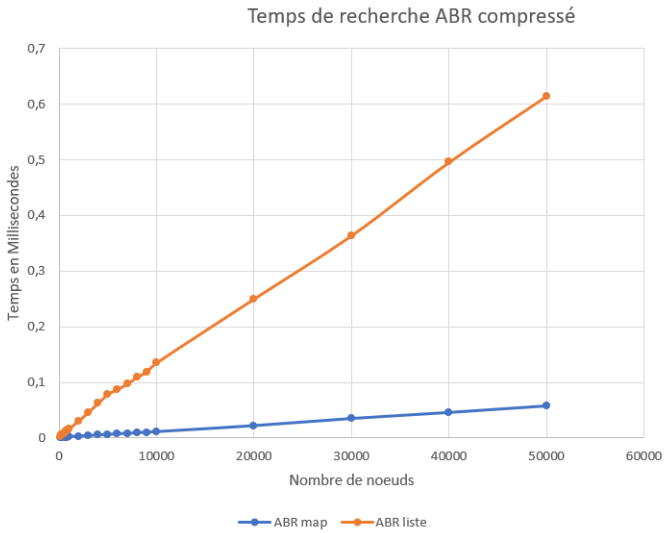
```
let time_test n = let r = list_of_rand n(*valeur max*) n(*nb valeur a tester*) in
        let mytree = construc (gen_permutation n(*taille de l'arbre*)) in
        let comp = compresser mytree in
          average_comp comp r;;
```

Cette fonction construit un ABR compressé de n noeuds et calcule son temps d'exécution. La fonction *average_comp* permet d'obtenir une moyenne de ce temps.

```
let compress_size_test tt=
  Gc.compact ();
  let s = Gc.allocated_bytes () in
  let _ = compresser tt in
  Gc.compact ();
  Gc.allocated_bytes () -. s;;
```

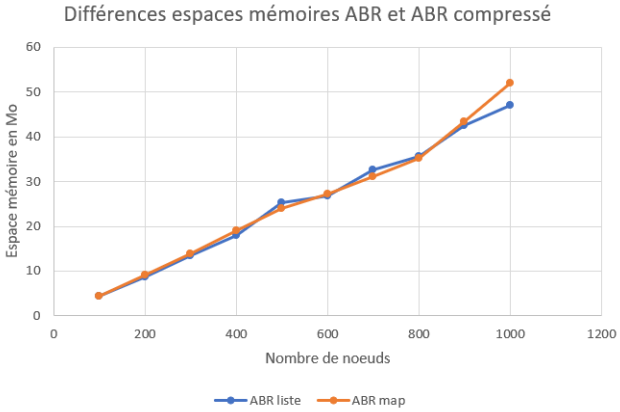
La fonction *compress_size_test* permet d'obtenir l'espace mémoire en octet de l'ABR compressé.
A partir de ces fonctions, nous avons déterminé les graphes du temps d'exécution de la recherche d'un élément dans un ABR compressé, du temps de compression ainsi que l'espace mémoire engendré par cette compression.

3.11



D'après le graphe, nous observons que la recherche d'un élément dans un ABR compressé pour l'utilisation d'une liste est supérieure à la recherche avec des maps au niveau du temps d'exécution. En effet, on stocke de manière plus efficace les clés-valeurs dans un maps donc la recherche est plus efficace.

3.12



D'après le graphe nous pouvons remarquer qu'en terme d'espace mémoire, l'ARB non compressé croît plus rapidement que les ABR compressés (cf *Annexes - Graphe1*). De plus, l'espace mémoire est assez similaire pour l'utilisation d'une liste ou d'une map (pour les clés-valeurs). D'après les résultats obtenus, nous pouvons remarquer que l'ABR compressé est plus optimisé en terme d'espace mémoire, mais pénalisant en termes de temps de recherche.

3.13 Nous avons implémenté *calcul* et *calculN* qui permettent de calculer le nombre de nœuds internes et le nombre moyen de clés par nœud interne

```
let rec calcul abr nbN moy = match abr with
    | Empty -> ()
    | Symbole(n,e) -> ()
    | NodeC(n) -> nbN:= !nbN + 1;
      moy :=!moy + (List.length n.etq);
      let _ = calcul !(n.fg) nbN moy in
      let _ = calcul !(n.fd) nbN moy in ();;

let calculN abr = let nbN = ref 0 and moy = ref 0 in
  let _ = calcul abr nbN moy in (!nbN,((float_of_int !moy) /. (float_of_int !nbN)));;
```

Pour chacun des fichiers de tests, nous avons obtenu ces résultats:

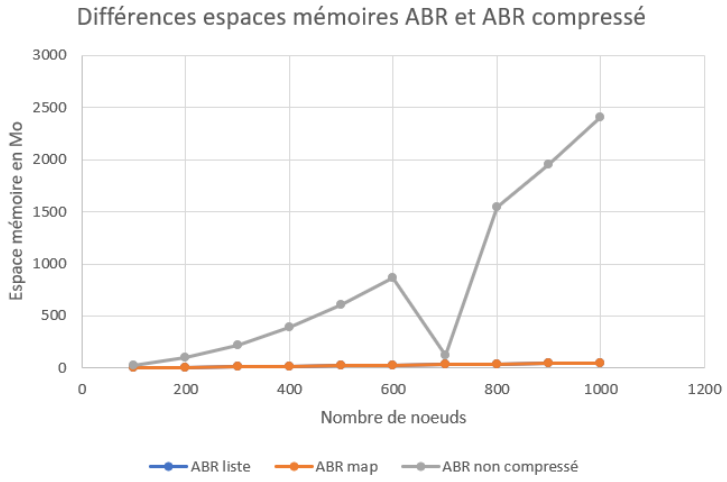
Nombre de noeuds	Nombre de noeuds internes	Nombre moyen de clés par noeuds internes
100	100	1.000000
150	59	2.542373
500	176	2.8409093
750	258	2.906977
1 000	1 000	1.000000
10 000	2497	4.004806
50 000	10846	4.609994

3.2 Conclusion

Durant ce projet, nous avons réussi à obtenir une structure compressé avec des listes ou des maps à la place des noeuds d'un ABR classique. Cette compression nous a permis d'obtenir les résultats présentés dans ce rapport. Malgré cela, certains choix que nous avons fait auraient pu être optimisés comme le fait de créer des symboles uniques en utilisant des entiers à la place de nos chaînes de caractères, ou bien le fait d'utiliser des champs mutables pour nos types aux niveaux des fils gauches/droits au lieu des références. Outre cela, ce projet nous a permis d'apprendre le langage OCaml et d'approfondir nos connaissances quant aux arbres binaires.

4 Annexes

Graphe1:



Graphe2: