UE Ouverture

Rapport de Projet

Wenzhuo ZHAO

Zhen HOU

Année 2020/2021

Présentation

Dans ce projet, nous cherchons une solution de manipuler un modèle de structure de données arborescence, les Arbres de Binaires de Recherche puis d'en construire une structure compressée suivant une procédure bien particulière.

Synthèse de données

Question 1.1

La fonction `extraction_alea` prend en entrée deux listes d'entiers, notée `l` et `p`. La fonction choisit aléatoirement un entier `r` entre 1 et la longueur de liste `l`, puis 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 en tête le `r`-ième élément extrait de L.

```
let rec remove_at n l = match l with
  | [] -> []
  | h :: t -> if n = 0 then t else h :: remove_at (n-1) t;;

let extraction_alea (l:int list) (p: int list) : (int list) * (int list) =
  let length = List.length l in
  let r = (Random.int length) in
  let e = List.nth l r in
  ((remove_at r l), (e :: p));;
```

La fonction auxiliaire `remove_at` permet de retirer un élément à la position `n` dans une liste `l`.

Question 1.2

La fonction `gen_permutation` réalise l'algorithme de *shuffle de Fisher-Yates*.

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

let gen_permutation (n: int) : int list =
  let l = interval 1 n in
  let p = [] in
  let rec shuffle s t =
    if List.length s = 0
    then t
    else
      let ea = (extraction_alea s t) in
      shuffle (fst ea) (snd ea) in
  shuffle l p;;
```

La fonction auxiliaire `interval` permet de générer une liste des entiers de `n` à `m`. Dans la fonction `gen_permutation`, nous l'utilisons pour générer la liste L des entiers de 1 à `n`.

Question 1.3

En fonction de `n`, nous avons besoin d'appeler au générateur de nombres aléatoires `n` fois pour que la liste L soit vide et remplit la liste P en appelant `extraction_alea` donc la complexité est $O(n)$.

En nombre de filtrage de motif qui est utilisé dans la fonction `remove_at` de la question 1.1, nous utilisons `match` en parcourant la liste pour trouver l'élément par l'indice donnée par `Random.int`. Supposons cette indice a la valeur `m`, nous avons besoin d'utiliser le filtrage de motif `n * (n-m)`fois donc la complexité est $O(n^2)$.

Construction de l'ABR

Un arbre binaire est :

- soit réduit à une feuille
- soit décomposable en une racine qui est un nœud interne et qui pointe vers deux enfants ordonnés, l'enfant gauche et l'enfant droit.

Question 1.7

En prenant ces deux cas, nous avons défini un type `abr` pour présenter un arbre binaire.

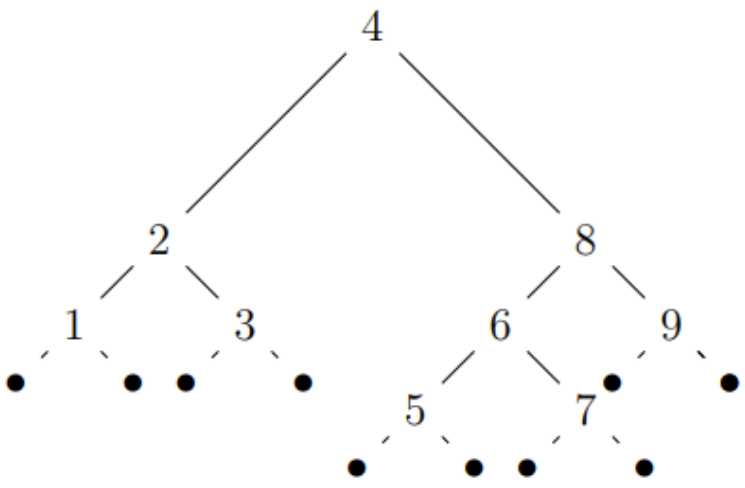
```
type abr =
  | Noeud of {etq: int; fg: abr; fd: abr}
  | Vide;;
```

`etq` présente l'étiquette sur le nœud, `fg` et `fd` représentent le fils gauche et le fils droit respectivement du nœud.

Pour un arbre binaire de recherche, la racine possède une étiquette plus grande que toutes les étiquettes de l'enfant gauche, et plus petite que toutes les étiquettes de l'enfant droit.

Étant donnée une liste d'entiers, nous construisons l'ABR en insérant les entiers.

```
let rec inserer (e: int) (a: abr) : abr = match a with
  | Vide -> Noeud {etq = e; fg = Vide; fd = Vide}
  | Noeud(n) ->
    if (e < n.etq)
    then Noeud {etq = n.etq; fg = (inserer e n.fg); fd = n.fd }
    else Noeud {etq = n.etq; fg = n.fg; fd = (inserer e n.fd) }

let construction (l: int list) =
  let vide = Vide in
  let rec aux (l: int list) (a: abr) =
    if l = [] then a
    else aux (List.tl l) (inserer (List.hd l) a)
  in aux l vide;;
```



Une version imprimée du figure au dessus est la suivante:

```
( 4 ( 2 ( 1 ϵ ϵ ) ( 3 ϵ ϵ ) ) ( 8 ( 6 ( 5 ϵ ϵ ) ( 7 ϵ ϵ ) ) ( 9 ϵ ϵ ) ) )
```

Compression des ABR

Afin de représenter l'ABR de manière plus compacte en mémoire, l'idée globale consiste à repérer les sous-arbres (non réduit à une feuille) ayant la même structure arborescente (en oubliant l'étiquetage) puis en remplaçant la deuxième occurrence du sous-arbre via un pointeur vers le premier sous-arbre.

Question 2.8

En associant une chaîne de caractères construite sur l'alphabet `{(,)}` à chaque arbre, nous pouvons identifier si deux arbres sont isomorphes.

```
(*La fonction Ø qui se lit "phi" en Grec*)
let rec phi (a: abr) : string = match a with
  | Vide -> ""
  | Noeud(n) -> "(" ^ (phi n.fg) ^ ")" ^ (phi n.fd);;
```

Question 2.9

Nous calculons un tableau contenant les étiquettes de l'arbre rangées en ordre préfixe.

```
let rec prefixe (a: abr) : int list = match a with
  | Vide -> []
  | Noeud(n) -> (n.etq)::(prefixe n.fg)@(prefixe n.fd);;
```

Question 2.10

Pour compresser un ABR, il est nécessaire de définir une nouvelle structure de données.

```
type abr_comp =
  | VideComp
  | NoeudComp of {etq: int; fg: abr_comp; fd: abr_comp; }
  | Pointeur of {etqs: int list; point: abr_comp ref};;
```

Le record `Pointeur` représente un sous-arbre par un pointeur qui stocke les étiquettes contenues dans ce sous-arbre dans le tableau `etqs`. L'attribut `point` représente une référence vers un sous-arbre ayant le même structure.

Pour ce nouveau type, nous devons réécrire les méthodes `phi` et `prefixe`(voir dans le fichier abr.ml). La construction d'un ABR compressé commence par initialiser un ABR compressé sans `Pointeur` à partir d'un ABR non compressé.

```
let rec init (a: abr) : abr_comp = match a with
  | Vide -> VideComp
  | Noeud(x) -> NoeudComp {etq = x.etq; fg = (init x.fg); fd = (init x.fd)};;
```

Puis nous cherchons tous les sous-arbres dans cet ABR compressé. Dans le figure au dessus, nous avons 9 sous-arbres. En ordre préfixe ils sont présentés comme les suivants:

- 1
- 3
- 2-1-3
- 5
- 7
- 6-5-7
- 9
- 8-6-5-7-9
- 4-2-1-3-8-6-5-7-9

```
let rec arbres (a: abr_comp) : (abr_comp ref list) = match a with
  | VideComp -> []
  | NoeudComp(n) -> (arbres n.fg)@(arbres n.fd)@[(ref a)]
  | Pointeur(n) -> [];;
```

Pour remplacer un nœud par un `Pointeur`, nous devons trouver un ABR compressé qui a une même structure comme ce nœud dans la liste renvoyée par la fonction `arbres`.

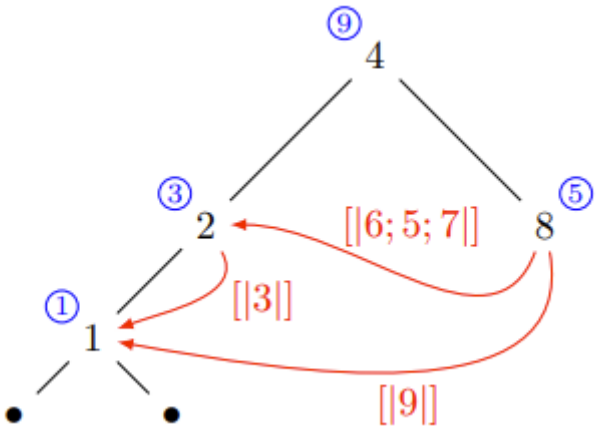
```
let rec find (a: abr_comp) (l: abr_comp ref list) : (abr_comp ref) =
  match l with
  | [] -> (ref VideComp)
  | x::xs -> if (egal_structure a !x)  then x else (find a xs)
```

La fonction `egal_structure` permet de savoir l'égalité de structures de deux ABR compressés.

Ayant fait toute la préparation, on peut commencer à remplacer des nœuds par `Pointeur` en parcourant l'ABR compressé initialisé par `init`.

```
let rec construction_comp (a: abr) : abr_comp = match a with
  | Vide -> VideComp
  | Noeud(n) ->
    let ab = (init a) in
    let l = (arbres ab) in
    let rec replace (a: abr_comp) =
      let e = (find a l) in
      let flag_found = (VideComp != !e) in
      if flag_found = true then
        if (identique a !e) = true
        then match a with
          | VideComp -> VideComp
          | NoeudComp(x) -> NoeudComp {etq = x.etq; fg = (replace x.fg ); fd = (replace x.fd )}
          | Pointeur(x) -> Pointeur {etqs = x.etqs; point = x.point}
        else match a with
          | VideComp -> VideComp
          | _ -> Pointeur {etqs = (prefixe_comp a); point = e}
      else
        match a with
        | VideComp -> VideComp
        | NoeudComp(x) -> NoeudComp {etq = x.etq; fg = (replace x.fg ); fd = (replace x.fd )}
        | Pointeur(x) -> Pointeur {etqs = x.etqs; point = x.point}
    in (replace ab );;
```

On vérifie que le ABR compressé parcouru `a` est identique avec un autre ABR compressé `e` trouvé par `find` dans la liste `l` renvoyée par `arbres` (les résultats de préfixe sont identiques). S'ils ne sont pas identiques, alors `a` peut être remplacé par une référence vers `e`.



Une version imprimée du figure au dessus est la suivante:

```
( 4 ( 2 ( 1 ϵ ϵ ) [ 3 ]->( 1 ϵ ϵ ) ) ( 8 [ 6 5 7 ]->( 2 ( 1 ϵ ϵ ) ( 3 ϵ ϵ ) ) [ 9 ]->( 1 ϵ ϵ ) ) )
```

Question 2.11

Pour implémenter une fonction de recherche de valeur dans un ABR compressé, il faut avoir l'accès à fils gauche/droit d'un ABR compressé qui peut être sous forme `Pointeur`

```
(*fils gauche d'un abr comp*)
let filsGauche (a: abr_comp ref) : abr_comp ref = match !a with
  | VideComp -> ref VideComp
  | NoeudComp(n) -> ref n.fg
  | Pointeur(n) -> ref VideComp;;

(*fils droit d'un abr comp*)
let filsDroit (a: abr_comp ref) : abr_comp ref = match !a with
  | VideComp -> ref VideComp
  | NoeudComp(n) -> ref n.fd
  | Pointeur(n) -> ref VideComp;;
```

Pour manipuler le tableau d'étiquettes dans `Pointeur`(voir la question 2.10), nous avons une fonction `slice` (voir le fichier abr.ml) qui peut extraire une partie contenant les éléments entre i-ème et k-ème (i et k sont inclus) dans ce tableau.

Si on tombe dans la cherche d'un élément `e` dans un tableau d'étiquettes, sachant que le premier élément `x` présente la racine de l'ABR. En comparant `e` et `x`, on peut savoir l'élément `e` est dans le fils droit ou gauche puis nous utilisons la fonction `slice` pour nous aider à sauter dans **la partie correspondante** puis le chercher récursivement.

La partie correspondante dépend la position de l'élément x:

- Si dans le fils droit, cette partie sera de 2ème élément au `d`- ème élément (dont `d` est la taille de fils droit)
- Si dans le fils gauche, cette partie sera de `d`+1 élément jusqu'à la fin du tableau.

Question 2.13

La complexité en moyenne de la recherche dans un ABR compressé dépend de la fonction `slice`. Elle utilise l'algorithme `take` et `drop` qui ont une complexité $O(n)$, donc la complexité de notre algorithme de la fonction *chercher* est en $O(n)$.

Expérimentations

Question 3.13

Nous proposons la fonction suivante pour calculer le temps pris par l'exécution d'un algorithme.

```
let time f x: float =
  let t = Sys.time() in
  let fx = f x in
  fx;
  (Sys.time() -. t);;
```

L'argument `f` est la fonction à tester alors l'argument `x` est l'argument de `f`.

Pour calculer l'espace mémoire occupé par une structure, nous avons trouvé ces deux fonctions:

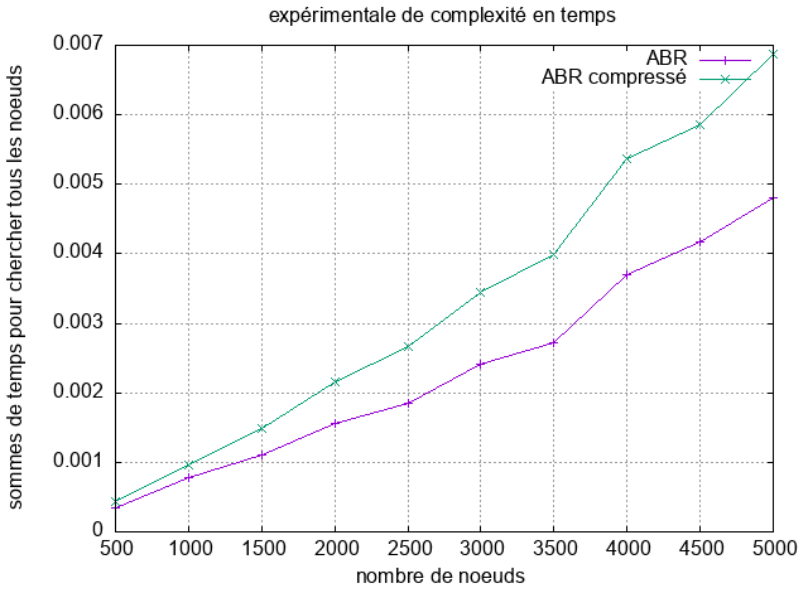
```
let size_abr (a: abr) : int = sizeof a;;

let rec size_abr_comp (a: abr_comp) : int = match a with
  | VideComp -> sizeof VideComp
  | NoeudComp(n) -> 4 + (size_abr_comp n.fg) + (size_abr_comp n.fd)
  | Pointeur(n) -> (sizeof n.etqs) + 1 ;;
```

En testant la fonction `size_abr`, on a eu le résultat qu'un ABR non compressé ayant seulement un nœud interne (racine) a la taille de 4 mots qui sont

- 3 valeurs: étiquette, fils gauche, fils droit
- son en-tête

Donc dans la fonction `size_abr_comp`, on peut en déduire qu'un nœud dans un ABR compressé a aussi au moins une taille de 4 mots. Dans le record `Pointeur`, selon la description d'OCaml, un pointeur a la même taille qu'un entier qui est un mot en machine, donc on peut en déduire qu'un `Pointeur` a la taille `(sizeof n.etqs) + 1` .

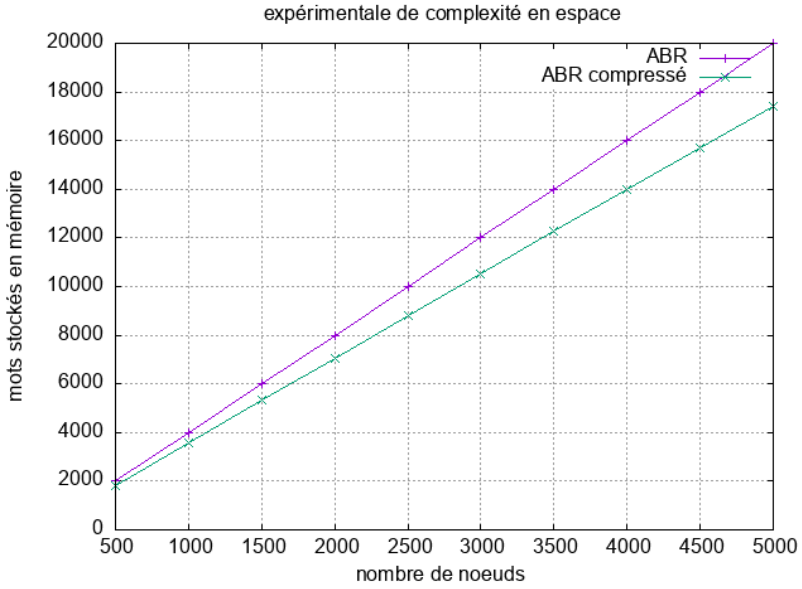
Question 3.14



L'axe des ordonnées est la somme de temps pour chercher tous les nœuds dans un ABR. S'il y a `n` entiers dans un ABR `a`, alors la valeur est `(time_chercher chercher a 1) +(time_chercher chercher a 2) + ... + (time_chercher chercher a n)`.

La fonction *chercher* dans un *ABR compressé* doit manipuler le tableau donc nous observons que *ABR compressé* prends plus de temps sur le même nombre de nœuds que *ABR* mais les deux fonctions *chercher* dans *ABR* et dans *ABR compressé* ont la même complexité.

Question 3.15



D'après le graphe, nous pouvons remarquer que l'ABR compressé est plus optimisé en terme d'espace mémoire. Cette optimisation est plus évidente si le nombre de nœuds `n` est assez grand.