

Le beurre et le Tarjan du beurre (Union-Find Persistant)

Sylvain Conchon et Jean-Christophe Filliâtre
Université Paris Sud / CNRS

JFLA 2007



Le problème des classes disjointes

maintenir dans une structure de données une partition de $\{0, 1, \dots, n - 1\}$

structure impérative :

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

Le problème des classes disjointes

maintenir dans une structure de données une partition de $\{0, 1, \dots, n - 1\}$

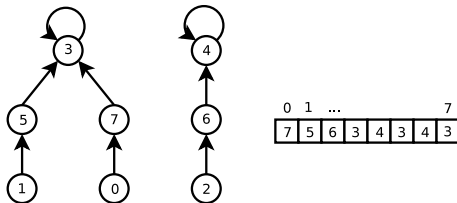
structure impérative :

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

Solution optimale : l'algorithme de Tarjan

(solution en réalité due à M. D. McIlroy et R. Morris, et seulement analysée par Tarjan)

dans chaque classe, les éléments sont chaînés jusqu'au représentant



deux idées clés :

- la **compression de chemin** pendant l'opération `find`
- l'utilisation de **niveaux** pendant l'opération `union`

Union-Find persistant

la solution de Tarjan est inadaptée au **backtracking**

- pas d'opération undo immédiate
- la copie de la structure serait catastrophique

une solution serait une structure **persistante** :

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

Union-Find persistant

la solution de Tarjan est inadaptée au **backtracking**

- pas d'opération undo immédiate
- la copie de la structure serait catastrophique

une solution serait une structure **persistante** :

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

Une solution naïve

```
module M = Map.Make(struct type t = int let compare = compare end)

type t = int M.t

let create n = M.empty

let find m i =
  let rec lookup i = try lookup (M.find i m) with Not_found → i in
  lookup i

let union m i j =
  let ri = find m i in
  let rj = find m j in
  if ri <> rj then M.add ri rj m else m
```

nous proposons une solution

- de **même efficacité** que l'algorithme de Tarjan
- **persistante** (mais utilisant des effets de bord)
- formellement **prouvée**

obtenue en combinant

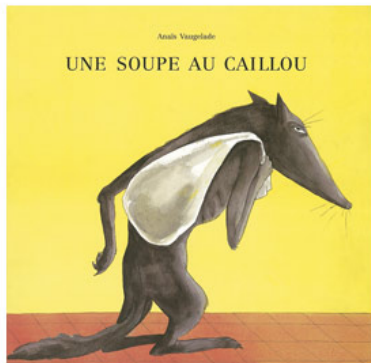
- une structure de données de **tableaux persistants**
- une version persistante de l'algorithme de Tarjan

nous proposons une solution

- de **même efficacité** que l'algorithme de Tarjan
- **persistante** (mais utilisant des effets de bord)
- formellement **prouvée**

obtenue en combinant

- une structure de données de **tableaux persistants**
- une version persistante de l'algorithme de Tarjan



Une version persistante de l'algorithme de Tarjan

pour rester indépendant de la structure de tableaux persistants :

```
module Make(A : PersistentArray) : PersistentUnionFind
```

en supposant

```
module type PersistentArray = sig
  type  $\alpha$  t
  val init : int  $\rightarrow$  (int  $\rightarrow$   $\alpha$ )  $\rightarrow$   $\alpha$  t
  val get :  $\alpha$  t  $\rightarrow$  int  $\rightarrow$   $\alpha$ 
  val set :  $\alpha$  t  $\rightarrow$  int  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$  t
end
```

Une version persistante de l'algorithme de Tarjan

on conserve les deux tableaux de l'algorithme de Tarjan

```
type t = { rank: int A.t; mutable father: int A.t }
```

le champ `mutable` permet de conserver la compression de chemins

Une version persistante de l'algorithme de Tarjan

la recherche du représentant avec compression de chemin :

```
let rec find_aux f i =  
  let fi = A.get f i in  
  if fi == i then  
    f, i  
  else  
    let f, r = find_aux f fi in  
    let f = A.set f i r in  
    f, r
```

l'enregistrement de la compression de chemin :

```
let find h x =  
  let f, rx = find_aux h.father x in h.father ← f; rx
```

Une version persistante de l'algorithme de Tarjan

la recherche du représentant avec compression de chemin :

```
let rec find_aux f i =  
  let fi = A.get f i in  
  if fi == i then  
    f, i  
  else  
    let f, r = find_aux f fi in  
    let f = A.set f i r in  
    f, r
```

l'enregistrement de la compression de chemin :

```
let find h x =  
  let f,rx = find_aux h.father x in h.father ← f; rx
```

Tableaux persistants

pour garder l'efficacité de Tarjan, il faut des tableaux persistants efficaces

solution connue depuis 1978 et due à **Henry Baker**

idée : un tableau persistant est

- soit un vrai tableau
- soit une indirection vers un tableau persistant

```
type  $\alpha$  t =  $\alpha$  data ref
and  $\alpha$  data =
  | Arr of  $\alpha$  array
  | Diff of int  $\times$   $\alpha$   $\times$   $\alpha$  t
```

pour garder l'efficacité de Tarjan, il faut des tableaux persistants efficaces

solution connue depuis 1978 et due à **Henry Baker**

idée : un tableau persistant est

- soit un vrai tableau
- soit une indirection vers un tableau persistant

```
type  $\alpha$  t =  $\alpha$  data ref
and  $\alpha$  data =
  | Arr of  $\alpha$  array
  | Diff of int  $\times$   $\alpha$   $\times$   $\alpha$  t
```

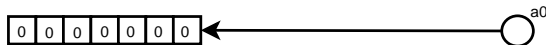

Tableaux persistants de Baker

```
let a0 = create 7 0
```

```
let a1 = set a0 1 7
```

```
let a2 = set a1 2 8
```

```
let a3 = set a1 2 9
```



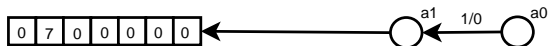
Tableaux persistants de Baker

```
let a0 = create 7 0
```

```
let a1 = set a0 1 7
```

```
let a2 = set a1 2 8
```

```
let a3 = set a1 2 9
```



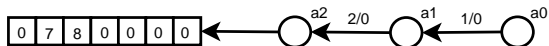
Tableaux persistants de Baker

```
let a0 = create 7 0
```

```
let a1 = set a0 1 7
```

```
let a2 = set a1 2 8
```

```
let a3 = set a1 2 9
```



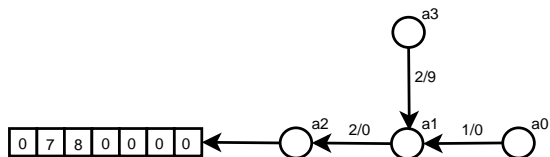
Tableaux persistants de Baker

```
let a0 = create 7 0
```

```
let a1 = set a0 1 7
```

```
let a2 = set a1 2 8
```

```
let a3 = set a1 2 9
```



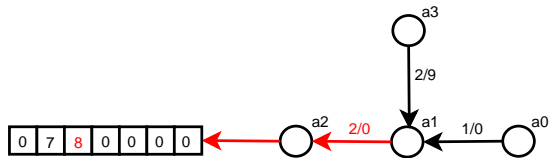
ces tableaux sont persistants mais inefficaces dès qu'on accède à une version antérieure

pour améliorer les performances, Baker introduit le **rerooting**

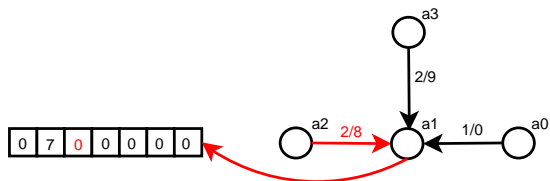
ces tableaux sont persistants mais inefficaces dès qu'on accède à une version antérieure

pour améliorer les performances, Baker introduit le **rerooting**

Rerooting



si on accède maintenant à a_1 , on commence par un reroot



Amélioration

dans un cas de **backtracking** où on revient toujours à des versions antérieures, on peut se dispenser du renversement de la chaîne de pointeurs



retour au tableau a1



- les tableaux obtenus ne sont plus totalement persistants \Rightarrow ils sont seulement **semi-persistants**
- la solution obtenue est comparable à une « pile de *undos* », mais celle-ci est cachée dans la structure de données

- les tableaux obtenus ne sont plus totalement persistants \Rightarrow ils sont seulement **semi-persistants**
- la solution obtenue est comparable à une « pile de *undos* », mais celle-ci est cachée dans la structure de données

tests inspirés d'une utilisation dans une procédure de décision

p = ratio union / find

N = nombre d'opérations entre deux points de backtrack

p	5%	5%	5%	10%	10%	10%	15%	15%	15%
N	$2 \cdot 10^4$	10^5	$5 \cdot 10^5$	$2 \cdot 10^4$	10^5	$5 \cdot 10^5$	$2 \cdot 10^4$	10^5	$5 \cdot 10^5$
Tarjan	0.31	2.23	12.50	0.33	2.34	12.90	0.34	2.36	13.20
V1	0.52	3.03	17.10	0.81	4.78	26.80	1.16	6.78	37.90
V2	0.34	2.01	11.70	0.42	2.54	14.90	0.52	3.21	18.70
V3	0.33	1.90	11.30	0.41	2.45	14.40	0.52	3.14	17.80
naïve	0.76	5.28	37.50	1.22	9.14	63.80	40.40	> 10mn	> 10mn

- V1 = tableaux persistants de Baker
- V2 = tableaux semi-persistants
- V3 = V2 défonctorisée

structure persistante mais nombreux **effets de bord** \Rightarrow correction non immédiate \Rightarrow preuve formelle menée avec l'aide de Coq

modèle-mémoire associant un type abstrait pointer à des valeurs

```
Inductive data : Set :=  
  | Arr : data  
  | Diff  : Z  $\rightarrow$  Z  $\rightarrow$  pointer  $\rightarrow$  data.
```

```
Record mem : Set := { ref : PM.t data; arr : Z $\rightarrow$ Z }.
```

structure persistante mais nombreux **effets de bord** \Rightarrow correction non immédiate \Rightarrow preuve formelle menée avec l'aide de Coq

modèle-mémoire associant un type abstrait pointer à des valeurs

```
Inductive data : Set :=  
  | Arr : data  
  | Diff  : Z  $\rightarrow$  Z  $\rightarrow$  pointer  $\rightarrow$  data.
```

```
Record mem : Set := { ref : PM.t data; arr : Z $\rightarrow$ Z }.
```

Modélisation de la persistance

`pa_valid m p` = p tableau persistant bien formé dans la mémoire m
`pa_model m p f` = le contenu de p est modélisé par la fonction f

Definition set :

$$\forall m:\text{mem}, \forall p:\text{pointer}, \forall i v:\mathbb{Z},$$
$$\text{pa_valid } m \text{ } p \rightarrow$$
$$\{ p':\text{pointer} \ \& \ \{ m':\text{mem} \mid$$
$$\forall f, \text{pa_model } m \text{ } p \text{ } f \rightarrow$$
$$\text{pa_model } m' \text{ } p' \text{ } f \wedge \text{pa_model } m' \text{ } p' \text{ } (\text{upd } f \text{ } i \text{ } v) \} \}.$$

Modélisation de la persistance

`pa_valid m p` = p tableau persistant bien formé dans la mémoire m
`pa_model m p f` = le contenu de p est modélisé par la fonction f

Definition set :

$$\forall m:\text{mem}, \forall p:\text{pointer}, \forall i \ v:\mathbb{Z},$$
$$\text{pa_valid } m \ p \rightarrow$$
$$\{ p':\text{pointer} \ \& \ \{ m':\text{mem} \mid$$
$$\forall f, \text{pa_model } m \ p \ f \rightarrow$$
$$\text{pa_model } m' \ p' \ f \wedge \text{pa_model } m' \ p' \ (\text{upd } f \ i \ v) \} \}.$$

une structure union-find persistante aux performances comparables à celles de l'algorithme de Tarjan

- exemple significatif de structure persistante mais non purement applicative
- exemple de preuve de programme ML impératif dans Coq
- introduit la notion de semi-persistance \Rightarrow peut-on en vérifier l'utilisation licite par typage ?