functional pearl

A Functional Implementation of the Garsia–Wachs Algorithm

Jean-Christophe Filliâtre (CNRS)

ML Workshop '08

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche SACLAY - ÎLE-DE-FRANCE

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

UNIVERSITÉ PARIS-SUD 11

Save Endo

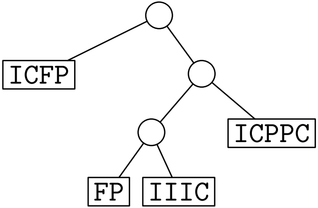
IIIPIPIIPCIIIPFFFFFPIIIPFFFFFPIIIPCCCCPIIIPIIIIIPIII...

Ropes

an opportunity to (re)discover **ropes**, a data structure for long strings

Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass
Ropes: An alternative to strings
Software - Practice and Experience, 25(12):1315–1330, 1995

```
type t =
  | Str of string
  | App of t × t
```



Balancing Ropes

access time to character i now proportional to the depth of its leaf
$\Rightarrow$ when height increases, access becomes costly

as binary search trees, ropes can be balanced
an on-demand rebalancing algorithm is proposed in the original paper

question: can we rebalance ropes in an **optimal** way,
i.e. with minimal mean time access to characters?

The Abstract Problem

given values $X_0, \ldots, X_n$ together with nonnegative weights $w_0, \ldots, w_n$, build a binary tree which **minimizes**

$$\sum_{i=0}^{n} w_i \times \mathrm{depth}(X_i)$$

and which has leaves $X_0, \ldots, X_n$ in inorder

One Solution: The Garsia–Wachs Algorithm

Adriano M. Garsia and Michelle L. Wachs
A new algorithm for minimum cost binary trees
SIAM Journal on Computing, 6(4):622–642, 1977

not widely known

described in

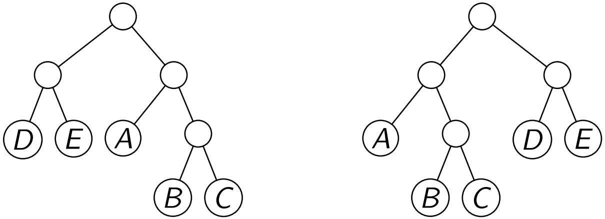
> *Donald E. Knuth*
> *The Art of Computer Programming*
> *Optimum binary search trees (Vol. 3, Sec. 6.2.2)*

The Algorithm

three steps

1. build a binary tree of optimum cost, but with leaf nodes in disorder
2. traverse it to compute the depth of each leaf X_i
3. build a new binary tree where leaves have these depths
   and are in inorder $X_0, \dots, X_n$

example : $A, 3;\ B, 2;\ C, 1;\ D, 4;\ E, 5$



Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \ldots, X_n, w_n$, and group trees two by two, until only one is left

- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$

$$(A), 3 \quad (B), 2 \quad (C), 1 \quad (D), 4 \quad (E), 5 \qquad i = 2$$

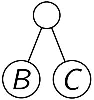
Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \ldots, X_n, w_n$, and group trees two by two, until only one is left

- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$

(A), 3 (D), 4 (E), 5

$t =$

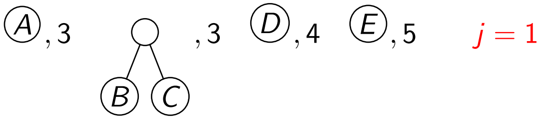


$w = 3$

Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \ldots, X_n, w_n$, and group trees two by two, until only one is left

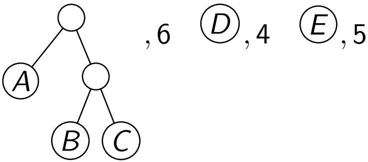
- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$



Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \ldots, X_n, w_n$, and group trees two by two, until only one is left

- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$

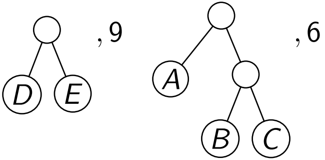


$i = 1 \; j = 0$

Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \ldots, X_n, w_n$, and group trees two by two, until only one is left

- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$

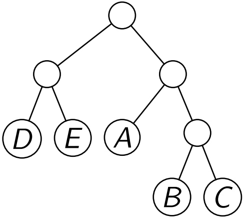


$i = 2 \; j = 0$

Step 1

similar to Huffman's algorithm: works on a list of weighted trees, started with $X_0, w_0, \dots, X_n, w_n$, and group trees two by two, until only one is left

- determine the smallest i such that $weight(t_{i-1}) \leq weight(t_{i+1})$
- link t_{i-1} and t_i, with weight $w = weight(t_{i-1}) + weight(t_i)$
- insert t at largest $j < i$ such that $weight(t_{j-1}) \geq w$



$i = 1$ $j = 0$

Steps 2 and 3

we now have to build a binary tree with leaf nodes in inorder

$$A, B, C, D, E$$

with depths (in that order)

$$2, 3, 3, 2, 2$$

soundness of the algorithm ensures that such a tree exists

a nice programming exercise!

ML Implementation

type α tree =
  | Leaf of α
  | Node of α tree $\times$ α tree

val garsia_wachs : $(\alpha \times \text{int})$ list $\rightarrow$ α tree

ML Implementation (step 1)

val phase1 : (α tree $\times$ int) list $\rightarrow$ α tree

we navigate in the list of weighted tree using a zipper

a zipper for a list is a **pair of lists**: the elements **before** the position (in reverse order) and the elements **after**

```
let phase1 l =
  let rec extract before after = ...
  and insert after t before = ... in
  extract [] l
```

ML Implementation (step 1)

```
let rec extract before = function
  | [] →
      assert false
  | [t,_] →
      t
  | [t1,w1; t2,w2] →
      insert [] (Node (t1, t2), w1 + w2) before
  | (t1, w1) :: (t2, w2) :: ((_, w3) :: _ as after) when w1 ≤ w3 →
      insert after (Node (t1, t2), w1 + w2) before
  | e1 :: r →
      extract (e1 :: before) r
```

ML Implementation (step 1)

```
and insert after ((_,wt) as t) = function
  | [] →
      extract [] (t :: after)
  | (_, wj_1) as tj_1 :: before when wj_1 ≥ wt →
      begin match before with
        | [] →
            extract [] (tj_1 :: t :: after)
        | tj_2 :: before →
            extract before (tj_2 :: tj_1 :: t :: after)
      end
  | tj :: before →
      insert (tj :: after) t before
```

ML Implementation (step 2)

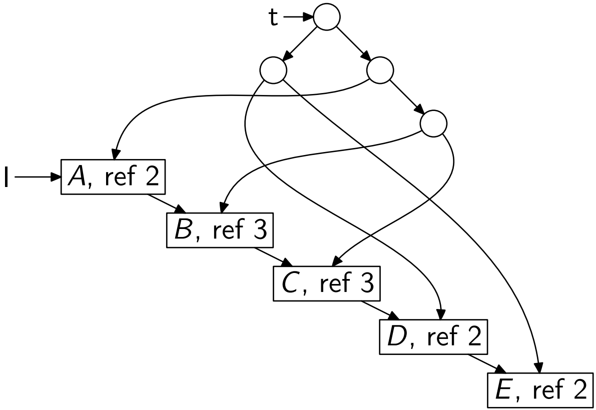
to retrieve depths easily, we associate a reference to each leaf

```
let garsia_wachs l =
  let l = List.map (fun (x, wx) → Leaf (x, ref 0), wx) l in
  let t = phase1 l in
  ...
```

then it is easy to set the depths after step 1, using

```
let rec mark d = function
  | Leaf (_, dx) → dx := d
  | Node (l, r) → mark (d + 1) l; mark (d + 1) r
```

Shared References



ML Implementation (step 3)

we build the tree from the list of its leaf nodes together with their depths

elegant solution due to R. Tarjan

```
let rec build d = function
  | (Leaf (x, dx), _) :: r when !dx = d →
      Leaf x, r
  | l →
      let left,l = build (d+1) l in
      let right,l = build (d+1) l in
      Node (left, right), l
```

Putting All Together

```
let garsia_wachs l =
  let l = List.map (fun (x, wx) → Leaf (x, ref 0), wx) l in
  let t = phase1 l in
  mark 0 t;
  let t, [] = build 0 l in
  t
```

Comparison with a C Implementation

the presentation of the Garsia–Wachs algorithm in TAOCP has a companion C code

this C code

- has time complexity $O(n^2)$, as our code
- uses statically allocated arrays and has space complexity $O(n)$
- is longer and more complex than our code

Benchmarks

for a fair comparison, the C program has been translated to Ocaml

timings for 500 runs on randomly selected weights

n	"C"	Ocaml
100	0.61	0.59
200	0.68	0.68
300	0.72	0.82
400	0.77	0.91
500	0.83	1.03

note: in the ICFP 2007 contest, the average size of ropes is 97 nodes (over millions of ropes)

Conclusion

the Garsia–Wachs algorithm deserves a wider place in literature
and has a nice application to ropes rebalancing

from the point of view of functional programming

- no harm in being slightly impure from time to time
- especially when side-effects are purely local