A Functional Implementation of the Garsia–Wachs Algorithm

(functional pearl)

Jean-Christophe Filliâtre

CNRS
LRI, Univ Paris-Sud, Orsay F-91405
INRIA Saclay - Île-de-France, ProVal, Orsay F-91893
filliatr@lri.fr

Abstract

This functional pearl proposes an ML implementation of the Garsia–Wachs algorithm. This somewhat obscure algorithm builds a binary tree with minimum weighted path length from weighted leaf nodes given in symmetric order. Our solution exhibits the usual benefits of functional programming (use of immutable data structures, pattern-matching, polymorphism) and nicely compares to the purely imperative implementation from *The Art of Computer Programming*.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Algorithms, Data Structures

Keywords Garsia–Wachs Algorithm, Optimum Binary Trees, Applicative Programming, Zipper

1. Introduction

The Garsia–Wachs algorithm [3] addresses the following problem. Given a sequence of values $X_0, \dots, X_n$, together with nonnegative integer weights $w_0, \dots, w_n$, we want to construct a binary tree with $X_0, \dots, X_n$ as leaf nodes in symmetric order, such that the sum

$$\sum_{i=0}^{n} w_i d_i$$

is minimum, where d_i is the distance of leaf node X_i to the root. This can be used to build optimum search tables, when data is organized within a binary search tree and when access frequencies are known in advance. It may also be used to balance *ropes* [1] in an optimal way, since a rope is precisely a binary tree with a character string on each leaf; thus taking w_i as the length of this string would minimize the average access cost to a character in the rope[1]. The reader may have already noticed the similarity with Huffman's algorithm [5], which builds a binary tree of minimum weighted path length. Contrary to Huffman's algorithm, though, the Garsia–Wachs algorithm must maintain the symmetric order of the leaf nodes and thus addresses a more difficult task.

[1] To the author's knowledge, the relation between the Garsia–Wachs algorithm and ropes has never been mentioned.

A detailed presentation of the Garsia–Wachs algorithm [3] can be found in *The Art of Computer Programming*, in the section devoted to optimum binary search trees [7, pages 446–453]. The algorithm is derived from observations of optimum binary trees and lemmas for its soundness are given. The pseudo-code for the algorithm itself is somewhat difficult to follow, though, because it assumes several choices of array-based data structures. A companion implementation in C is provided by Knuth himself [6] and we use this implementation as reference.

This functional pearl proposes an ML implementation of the Garsia–Wachs algorithm. It is based on two key ideas. The first one is to use Huet's zipper to implement the first phase of the algorithm. The second one is to use *side-effects* to improve efficiency in the last phase of the algorithm. This is thus a *mostly functional* pearl. However, side-effects are purely local and not exposed to the client code, and the usual benefits of functional programming remain (immutable data structures for lists, pattern-matching, polymorphism). Our ML code nicely compares to Knuth's C code.

This article is organized as follows. Section 2 presents the Garsia–Wachs algorithm, then Section 3 details our functional implementation. Section 4 is devoted to the comparison of our code with the C implementation. The code presented in this paper is available at `http://www.lri.fr/~filliatr/garsia-wachs/`, together with the scripts used to test its efficiency.

2. The Garsia–Wachs Algorithm

In this section, we present the Garsia–Wachs algorithm independently of any choice of data structures.

The algorithm is decomposed in three phases:

1. first it builds a binary tree of optimum cost, but with leaf nodes in disorder;
2. then it traverses it to compute the depth of each leaf node X_i;
3. finally it builds a new binary tree where leaf nodes have the same depths but are now in symmetric order $X_0, \dots, X_n$.

Figure 1 illustrates the idea on a small example with $n = 4$. The leaf nodes are A, B, C, D, E, in this order, and their weights are $3, 2, 1, 4, 5$, respectively. Phase 1 of the algorithm builds the binary tree displayed on the left part of the figure. It sets the depths of the leaf nodes, which are here 2 for A, D and E, and 3 for B and C. Then phase 3 builds the binary tree displayed on the right part of the figure, where leaf nodes are now in symmetric order A, B, C, D, E.

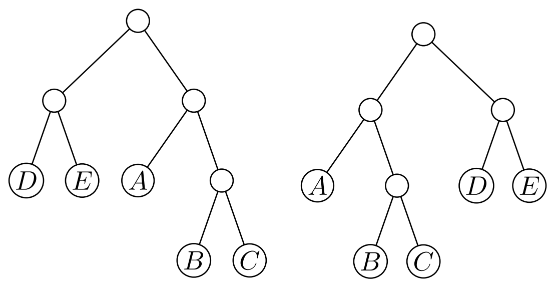


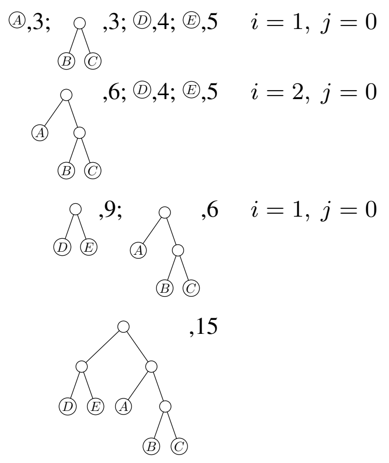
Figure 1. The Garsia–Wachs algorithm applied to the list $A, 3; B, 2; C, 1; D, 4; E, 5$.

We now detail phase 1 of the algorithm. It maintains a list of weighted trees $q_0, \dots, q_m$ and repeatedly links two trees to make a new one, until a single tree is left, in a way reminiscent of Huffman's algorithm. Initially, the list contains the leaf nodes $X_0, \dots, X_n$ associated to their weights $w_0, \dots, w_n$. As long as the list of trees contains at least two elements, we perform the following operations:

1. determine the smallest i such that
$$weight(q_{i-1}) \leq weight(q_{i+1}),$$
if any, and let $i = m$ otherwise;
2. extract trees q_{i-1} and q_i from the list, and make a new tree t with left subtree q_{i-1}, right subtree q_i and weight $weight(q_{i-1}) + weight(q_i)$;
3. determine the greatest j such that $j < i$ and
$$weight(q_{j-1}) \geq weight(t),$$
if any, and let $j = 0$ otherwise;
4. insert t right after q_{j-1} (and thus at the beginning of the list when $j = 0$).

When applying phase 1 to the example from Figure 1, we get the following iterations. Weights are indicated beside the trees and values for i and j in the right column.

Ⓐ,3; Ⓑ,2; Ⓒ,1; Ⓓ,4; Ⓔ,5 $\quad i = 2,\ j = 1$



Once phase 1 is completed, we get a tree t with the initial $n+1$ leaves, but they are not necessarily in the correct left-to-right order (as illustrated on the example above). Soundness of the Garsia–Wachs algorithm guarantees that tree t has optimum cost *and* that it can be rearranged into another tree t' where each leaf appears at the same depth as in t (hence t' also has optimum cost). Therefore phase 2 of the algorithm consists in traversing t to get the depth of each leaf, which is simply achieved with a linear traversal of the tree. Then phase 3 consists in rebuilding t' from the initial list $X_0, \dots, X_n$ together with the depths computed at phase 2. This last phase is a nice programming exercise; the solution is given below but the reader may want to stop reading for a while to give it a try.

3. Functional Implementation

We now detail our implementation of the Garsia–Wachs algorithm. Our code is written in Objective Caml [2] (OCAML for short) but could obviously be written in other dialects of ML, as long as mutable references are provided.

Since our goal is to build a binary tree, we first introduce a polymorphic data type for such trees.

```
type α tree =
  | Leaf of α
  | Node of α tree × α tree
```

The Garsia–Wachs algorithm is implemented as a function which turns a list of weighted leaf nodes into a tree. It is thus a function of the following type:

```
val garsia_wachs : (α × int) list → α tree
```

The list taken as argument is assumed to be non-empty. We first implement phase 1 of the Garsia–Wachs algorithm, as a function

```
val phase1 : (α tree × int) list → α tree
```

Note that it is a polymorphic function, since it ignores the contents of trees; only weights are used. The key idea is to use Huet's zipper [4] to implement phase1. Indeed, we need to repeatedly move back and forth in the list of trees, first moving right to find i and to extract q_{i-1} and q_i, and then moving left to find j and to insert t. The zipper is precisely a means to navigate through a purely applicative data structure and to perform local modifications. The zipper for a list is simply a pair of lists: the first list represents the elements on the left of the "pointer", in reverse order, and the second list represents the elements on the right of the pointer.

Thus we implement phase1 as two mutually recursive functions, extract and insert, which operate on a pair of lists of weighted trees (before, after). Initially, the pointer is set on the beginning of the list, *i.e.* the list before is the empty list.

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

Function extract implements steps 1 and 2 of phase 1, *i.e.* it scans the list after for a suitable pair of trees to extract, builds the corresponding tree t and then call insert. It works as follows. The list after is assumed to be non-empty.

```
  let rec extract before = function
    | [] → assert false
```

If it is reduced to a single element, we are done with phase 1 and we return the corresponding tree.

```
    | [t,_] → t
```

If we reach the end of the list, we build t from the last two elements (case $i = m$) and we call insert.

```
    | [t1,w1; t2,w2] →
        insert [] (Node (t1, t2), w1 + w2) before
```

If we meet the requirement over the weights of q_{i-1} and q_{i+1}, we similarly build t and call insert.

```
    | (t1, w1) :: (t2, w2) :: ((_, w3) :: _ as after)
      when w1 ≤ w3 →
        insert after (Node (t1, t2), w1 + w2) before
```

Otherwise, we simply advance in the list, *i.e.* we move one element from after to before and call extract recursively.

```
    | e1 :: r → extract (e1 :: before) r
```

Function insert implements steps 3 and 4 of phase 1, *i.e.* it scans the list before for a suitable place where to insert t. If we reach the beginning of the list (case $j = 0$) we simply insert t in front of after and call extract recursively.

```
  and insert after ((_,wt) as t) = function
    | [] → extract [] (t :: after)
```

Otherwise we check for the condition $weight(q_{j-1}) \geq weight(t)$. If it is met, we have found the place where to insert t, that is right in front of after. We also need to move two elements from before to after before calling extract recursively, since we may have broken the invariant $weight(q_{i-1}) > weight(q_{i+1})$ locally. This requires a particular case when before contains only one element.

```
    | (_, 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
```

Note that we could compare the weights of tj_2 and t at this point, and avoid an unnecessary call to `extract` when the extraction condition is immediately met. We omit this optimization, though, to avoid multiplying cases. Finally, if the insertion condition is not met, we simply move one element from before to after and call insert recursively.

```
    | tj :: before → insert (tj :: after) t before
```

This completes the code of phase1.

We now need to implement phase 2, which traverses the tree returned by phase 1 to determine the depths of all leaf nodes. This is where we use our second key idea. To turn the user input list into a list of trees, we need to apply constructor Leaf to each value of the input list. We take this opportunity to store a reference in each leaf, that we will later use to set depths of leaf nodes. Thus the beginning of the function garsia_wachs is as follows.

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

Note that l has now type $((\alpha \times$ int ref) tree $\times$ int) list. But since phase1 is polymorphic, it applies to l as well. Setting the depths of the leaf nodes is now a trivial task: we simply traverse the tree t with a recursive function mark which takes the depth d as argument.

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

Each time a leaf node is reached, it contains a reference dx which is set to d. The key is the *sharing* of references between l and t, which is depicted on Figure 2. (We omit the weights in l, as well as the boxing of pairs and references, for the sake of clarity.)

We are now left with the last phase of the Garsia–Wachs algorithm, which consists in building the final tree from the initial list

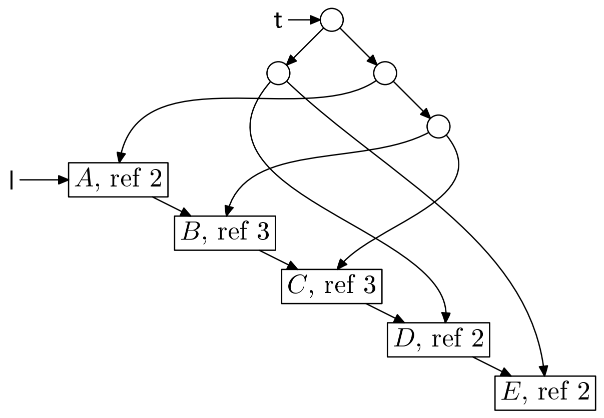


Figure 2. List l and tree t sharing references.

of leaf nodes and their depths. This is rather trivial, since l contains the leaf nodes in the right order *and* each node in this list contains a reference to its depth. Building the tree from the list can be done recursively over the list, following a nice solution due to R. E. Tarjan [7, page 713]. It consists in a function build which takes a depth d and a list of leaf nodes l as arguments, and builds a subtree rooted at depth d using a prefix of l. It returns this tree, together with the remaining elements of l. The code for build is the following. The list cannot be empty, nor it can contain a Node.

```
let rec build d = function
  | [] | (Node _, _) :: _ →
      assert false
```

If its first element is a leaf of the expected depth d, we return a tree reduced to this leaf, together with the remaining element of the list.

```
  | (Leaf (x, dx), _) :: r when !dx = d →
      Leaf x, r
```

Otherwise, we recursively build two subtrees left and right rooted at depth d+1, consuming the list elements meanwhile, and return the tree together with the remaining elements.

```
  | l →
      let left,l = build (d+1) l in
      let right,l = build (d+1) l in
      Node (left, right), l
```

Note that this process would obviously fail on arbitrary values for the depths. But it succeeds here since soundness of the Garsia–Wachs algorithm ensures that such a tree exists.

Putting all together, we get the following code for the main function.

```
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
```

It simply combines the three phases. The whole code is 44 lines long and given in appendix A.

4. Comparison with the C Implementation

This section compares our implementation to a C implementation by Knuth himself [6], which is given in appendix B. This C code is considered by Knuth as "a quick-and-dirty implementation", written as he was preparing the 2nd edition of Volume 3. Yet it strictly follows the description from *The Art of Computer Programming* [7, page 451] and thus can be legitimately considered as a reference.

We first compare the programs from the performance point of view. For several values of n, we perform 500 runs of each program on randomly selected weights. (Both programs are run on exactly the same input values, of course.) To make the comparison fair, any printing on the standard output has been removed from the C code. Since the C program is not considering values but only weights, the OCAML code has been specialized accordingly, to avoid the allocation of unnecessary pairs. We measure CPU time using UNIX `time` command. The timings are given in seconds in the following table[2].

n	C	OCAML
100	0.58	0.38
200	0.62	0.72
300	0.95	1.06
400	1.25	1.44
500	1.56	1.82

As we can see, the OCAML code is slightly slower than the C code, but not that much (less than 20% slower). It is even faster for $n = 100$.

Both codes have the same $O(n^2)$ worst case complexity. Time is spent in phase 1, which repeats n times the extraction/insertion operation. This part of the algorithm scans weights to determine i and j and thus may require $O(n)$ steps in the worst case. The OCAML and C codes manipulate the list of weights differently, with a zipper for lists and arrays respectively, the resulting complexity is exactly the same. The use of a linked list would make extraction and insertion $O(1)$ but time proportional to n will still be spent in determining positions i and j. As pointerd out by Knuth, there are ways to implement the Garsia–Wachs algorithm in $O(n \log n)$ with more sophisticated data structures [7, page 713]. But for the purpose of comparison, we wanted to stick to the presentation from *The Art of Computer Programming* and to its companion code.

It is also worth mentioning that our code definitely provides a better interface than the C code. Indeed, it turns a user list of weighted values into a tree, while the C program works on a statically allocated array and only considers weights. If the addition of user values would not make the C program much more complex, dynamic memory allocation to return a fresh tree would obviously make it less efficient. Even with this difference, the OCAML code is smaller than the C code. Finally, the C code has a built-in maximal size, while the OCAML code has not.

5. Conclusion

"Shall I be pure or impure?" said Wadler once [9]. This functional pearl shows that there is no harm in being slightly impure from time to time. This is especially true when side-effects are purely *local* and are not exposed to the client code. This is the case here, since the algorithm is implemented as a function taking a pure list as argument and returning a pure tree. The internal use of references in leaf nodes does not leak out of the algorithm code. Yet the use of references makes a huge difference on the point of view of efficiency. Indeed, there is no simple way to retrieve the depths from the intermediate tree in a purely applicative solution, since we do not assume any comparison over the X_i's. The key here is the sharing of references between the initial input list and the intermediate tree, which gives constant access to the depths once they are computed.

Of course, there were many other ways to get the same result with the same complexity. For instance, we could build the *list* of depths in phase 2, and then traverse two lists at the same time in phase 3; or we could store depths in an array and then have phase 3 traversing this array with an extra integer argument; and so on. But all these solutions are less elegant, since they either allocate memory unnecessarily or clutter the code with too many imperative details.

A. Ocaml Implementation

The OCAML code for the Garsia–Wachs algorithm is given below.

```
type α tree =
  | Leaf of α
  | Node of α tree × α tree

(* phase 1: build an optimum tree, with leaves in any order *)

let phase1 l =
  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
  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
  in
  extract [] l

(* phase 2: mark each leaf with its depth *)

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

(* phase 3: build a tree from the list of leaves/depths *)

let rec build d = function
  | [] | (Node _, _) :: _ →
      assert false
  | (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


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;
```

[2] The programs and input data used for these tests are available online, at `http://www.lri.fr/~filliatr/garsia-wachs/`.

```
let t, [] = build 0 l in
t
```

B. C Implementation

This section contains the C implementation by Knuth. The code was written in CWEB [8], a literate programming tool for C. Since we do not assume the reader to be familiar with CWEB syntax, we only give here the resulting C code. The code is slightly longer than the OCAML code, in particular because it includes the parsing of weights on the command line (at the beginning of function main). We prefer to leave the code unmodified, though. For a detailed explanation of this code, the reader should refer to its pseudo-code description [7, page 451] and to its CWEB source [6].

```
#define size 64
#include <stdio.h>

int w[size]; /* node weights */
int l[size], r[size]; /* left and right children */
int d[size]; /* depth */
int q[size]; /* working region */
int v[size]; /* number of node in working region */
int t; /* current size of working region */
int m; /* current node */

void combine(register int k) {
    register int j, d, x;
    m++;
    l[m] = v[k - 1];
    r[m] = v[k];
    w[m] = x = q[k - 1] + q[k];

    t−−;
    for (j = k; j ≤ t; j++)
        q[j] = q[j + 1], v[j] = v[j + 1];
    for (j = k - 2; q[j] < x; j−−)
        q[j + 1] = q[j], v[j + 1] = v[j];
    q[j + 1] = x;
    v[j + 1] = m;


    while (j > 0 ∧ q[j - 1] ≤ x) {
        d = t - j;
        combine(j);
        j = t - d;
    }
}

void mark(int k, int p) {
    d[k] = p;
    if (l[k] ≥ 0)
        mark(l[k], p + 1);
    if (r[k] ≥ 0)
        mark(r[k], p + 1);
}

void build(int b) {
    register int j = m;
    if (d[t] ≡ b)
        l[j] = t++;
    else {
        m−−;
        l[j] = m;
        build(b + 1);
    }
    if (d[t] ≡ b)
        r[j] = t++;
    else {
        m−−;
        r[j] = m;
        build(b + 1);
    }

}

void main(int argc, char ⋆argv[]) {
    register int i, j, k, n;

    n = argc - 2;
    if (n < 0) {
        fprintf(stderr, "Usage: %s wt0 ... wtn\n", argv[0]);
        exit(0);
    }
    if (n + n ≥ size) {
        fprintf(stderr,
                "Recompile me with a larger tree size!\n");
        exit(0);
    }
    for (j = 0; j ≤ n; j++) {
        if (sscanf(argv[j + 1], "%d", &m) ≢ 1) {
            fprintf(stderr, "Couldn't read wt%d!\n", j);
            exit(0);
        }
        w[j] = m;
        l[j] = r[j] = -1;
    }

    m = n;
    t = 1;
    q[0] = 1000000000; /* infinity */
    q[1] = w[0];
    v[1] = 0;
    for (k = 1; k ≤ n; k++) {
        while (q[t - 1] ≤ w[k])
            combine(t);
        t++;
        q[t] = w[k];
        v[t] = k;

    }
    while (t > 1)
        combine(t);

    mark(v[1], 0);

    t = 0;
    m = 2 ⋆ n;
    build(1);

}
```

Acknowledgments

I am grateful to J. Kanig, Y. Moy and Y. Régis-Gianas for careful readings of an early draft of this paper. This article was written with due respect to Knuth's work and thankfulness for the marvels contained in *The Art of Computer Programming*.

References

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

[2] Xavier Leroy *et al.* The Objective Caml language. `http://caml.inria.fr/`.

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

[4] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, Septembre 1997.

[5] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[6] D. E. Knuth. CWEB implementation of Garsia–Wachs algorithm. `http://www-cs-faculty.stanford.edu/~knuth/programs/garsia-wachs.w`.

[7] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[8] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, 1993. CWEB is available at `http://www-cs-staff.Stanford.EDU/~knuth/cweb.html`.

[9] Philip Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, August 1992.