Correctness of Constructing Optimal Alphabetic Trees Revisited

Marek Karpinski [*]

Lawrence L. Larmore [†]

Wojciech Rytter [‡]

Abstract

Several new observations which lead to new correctness proofs of two known algorithms (Hu-Tucker and Garsia-Wachs) for construction of optimal alphabetic trees are presented. A generalized version of the Garsia-Wachs algorithm is given. Proof of this generalized version works in a structured and illustrative way and clarifies the usually poorly-understood behavior of both the Hu-Tucker and Garsia-Wachs algorithms. The generalized version permits any non-negative weights, as opposed to strictly positive weights required in the original Garsia-Wachs algorithm. New local structural properties of optimal alphabetic trees are given. The concept of *well-shaped segment* (a part of an optimal tree) is introduced. It is shown that some parts of the optimal tree are known in advance to be well-shaped, and this implies correctness of the algorithms rather easily. The crucial part of the correctness proof of the Garsia-Wachs algorithm, namely the *structural theorem*, is identified. The correctness proof of the Hu-Tucker algorithm consists of showing a very simple mutual simulation between this algorithm and the Garsia-Wachs algorithm. For this proof, it is essential to use the generalized version of Garsia-Wachs algorithm, in which an arbitrary locally minimal pair is processed, not necessarily the rightmost minimal pair. Such a generalized version is also needed for parallel implementations. Another result presented in this paper is the clarification of the problem of resolving ties (equalities between weights of items) in the Hu-Tucker algorithm. This is related to the proof, by simulation, of correctness of the Hu-Tucker algorithm. It is shown that the condition that there are no ties may generally be assumed without harm and that, essentially, the Hu-Tucker algorithm avoids ties automatically.

---

[*] Dept. of Computer Science, University of Bonn, 53117 Bonn. This research was partially supported by DFG Grant KA 673/4-1, and by ESPRIT BR Grant 7097 and ECUS 030. Email:`marek@cs.uni-bonn.de`

[†] Department of Computer Science, University of Nevada, Las Vegas, NV 89154–4019, USA. Partially supported by National Science Foundation grants CCR-9112067 and CCR-9503441. Email:`larmore@cs.unlv.edu`

[‡] Institute of Informatics, Warsaw University, 02–097 Warszawa. Supported by the grant KBN 8T11C01208. Email:`rytter@mimuw.edu.pl`

1 Introduction

Recently, there has been a renewed interest in the problem of construction of optimal alphabetic trees [6, 10, 11, 12]. The Hu-Tucker (HT) algorithm [3] is a celebrated classical algorithm for this problem, whose correctness is not widely understood. The Garsia-Wachs (GW) algorithm [1], has a simpler but still very technical proof based on several formal claims proved simultaneously by induction. Our proof of correctness of the HT algorithm works by reducing to correctness of a general version of the GW algorithm in which *any* locally minimal pair is processed, not necessarily the rightmost one. This general version is also needed in our parallel implementations (see [11]). A restricted version of the GW algorithm (for *rightmost* minimal pairs) was considered in [5].

The aim of this paper is to provide proofs of correctness of both the HT and the GW algorithms that are more structural than those in the original papers. The simplicity of the description of both algorithms is misleading. The original correctness proofs are very intricate. According to Knuth, "No simple proof is known, and it is quite possible that no simple proof will ever be found!" ([7], page 443.) We provide several new facts about the local structure of optimal alphabetic trees, introducing new local operations on trees, and specify a mutual simulation between both algorithms.

Statement of the optimal alphabetic tree problem. Assume we have n weighted items, where p_i is the non-negative *weight* of the i^{th} item. Write $\alpha = p_1 \dots p_n$.

The Garsia-Wachs algorithm permutes α. We adopt the convention that the items of α have unique names, and that these names are preserved when items are moved. When convenient to do so, we will assume that those names are the positions of items in the list, namely integers in $[1 \dots n]$.
An alphabetic tree over α is an ordered binary tree T with n leaves, where the i^{th} leaf (in left-to-right order) corresponds to the i^{th} item of α. Throughout this paper, a binary tree must be *full*, *i.e.*, each internal node must have exactly two sons. We define the *cost* of any alphabetic tree T as follows:

$$cost(T) = \sum_{i=1}^{n} p_i \, level_T(i)$$

where $level_T$ is the *level function* of T, *i.e.*, $level_T(i)$ is the level (or depth) of i in T, defined to be the length of the path in T from the root to i. The optimal alphabetic tree problem (OAT problem) is to find an alphabetic tree of minimum cost. Both the GW and HT algorithms have two phases. The first phase constructs the level function $level_T$ of an optimal alphabetic tree T. The second phase constructs T from its level function. a relatively trivial procedure that takes linear time. In fact, throughout the rest of the paper, we ignore this second phase, and take the array of values of the level function to be the output of any algorithm for the OAT problem.

Construction of the optimal alphabetic tree. The alphabetic tree is constructed by reducing the initial sequence of items to a shorter sequence in a manner similar to that of the Huffman algorithm, with one important difference. In the Huffman algorithm, the minimum pair of items are combined, because it can be shown that they are siblings in the optimal tree. If we could identify two adjacent items that are siblings in the optimal alphabetic tree, we could combine them and then proceed recursively. Unfortunately, there is no known way to identify such a pair. Even a minimal pair may not be siblings. Consider the weight sequence $(8\,7\,7\,8)$. The second and the third items are not siblings in any optimal alphabetic tree.

Instead, the HT and GW algorithms, as well as the algorithms of [6, 10, 11, 12], operate by identifying a pair of items that have the same level in the optimal tree. These items are then combined into a single "package," reducing the number of items by one. The details on how this process proceeds differ in the different algorithms.

2 Correctness of the Garsia-Wachs algorithm

Define $\mathit{TwoSum}(i) = p_i + p_{i+1}$, the i^{th} *two-sum*, for $1 \leq i < n$. A pair of adjacent items $(i, i+1)$ is a *locally minimal pair* (or *lmp* for short) if

$$\begin{array}{rcll} \mathit{TwoSum}(i-1) & \geq & \mathit{TwoSum}(i) & \text{if} \quad i > 1 \\ \mathit{TwoSum}(i) & < & \mathit{TwoSum}(i+1) & \text{if} \quad i \leq n-2 \end{array}$$

A locally minimal pair which is currently being processed is called the *active pair.*

The Operator Move**.** If w is any item in a list π of weighted items, define $\mathit{RightPos}(w)$ to be the predecessor of the nearest right larger neighbor of w. In this context, "larger" means "greater than or equal to." If w has no right larger neighbor, define $\mathit{RightPos}(w)$ to be the last item of π. Let $\mathit{Move}(w, \pi)$ be the operator that changes π by moving w just to the right of $\mathit{RightPos}(w)$. Note that if $\mathit{RightPos}(w) = w$, then $\mathit{Move}(w, \pi)$ does nothing.

Similarly, if u, v are adjacent items in π, define $\mathit{RightPos}(u, v)$ to be the predecessor of the nearest item to the right of v whose weight is at least $\mathit{weight}(u) + \mathit{weight}(v)$. If there is no such item, define $\mathit{RightPos}(u, v)$ to be the last item of π. Let $\mathit{Move}(u, v, \pi)$ be the operator that changes π by moving u and v to just to the right of $\mathit{RightPos}(u, v)$. For example, if $\pi = (1, \ldots n)$, and if $\mathit{RightPos}(i, i+1) = j$, then $\mathit{Move}(i, i+1, \pi)$ changes π to

$$\pi_{i,j} = (1, \ldots, i-1, i+2, \ldots, j, i, i+1, j+1, \ldots, n)$$

Two binary trees T_1 and T_2 are said to be **level equivalent** (we write $T_1 \cong T_2$) if T_1, and T_2 have the same set of leaves (possibly in a different order) and $\mathit{level}_{T_1} = \mathit{level}_{T_2}$.

Theorem 2.1 (Correctness of the GW **algorithm)** *Let* $(i, i+1)$ *be a locally minimal pair and* $\mathit{RightPos}(i, i+1) = j$*, and let* T' *be a tree over the sequence* $\pi_{i,j}$*, optimal among all trees over* $\pi_{i,j}$ *in which* i*,* $i+1$ *are siblings. Then there is an optimal alphabetic tree* T *over the original sequence* $\pi = (1, \ldots n)$ *such that* $T \cong T'$*.*

The significance of Theorem 2.1 is that $\mathit{level}_{T'}$ may be computed by combining i and $i+1$ into a single node, v, and then applying the procedure recursively on the resulting list of length $(n-1)$. Then $\mathit{level}_T(i) = \mathit{level}_T(i+1) = \mathit{level}_{T'}(v) + 1$, while $\mathit{level}_T = \mathit{level}_{T'}$ on all other items.

The array *level* is global of size $(2n-1)$. Its indices are the names of the nodes, *i.e.*, the original n items and the $(n-1)$ nodes ("packages") created during execution of the algorithm. The algorithm works in quadratic time, if implemented in a naive way. Using priority queues, it works in $O(n \log n)$ time. Correctness follows directly from Theorem 2.1.

```
procedure GW($\pi$); {$\pi$ is a sequence of names of items}
{General version of the Garsia-Wachs algorithm}
if $\pi = (v)$ then
    level[$v$] = 0  else begin
      (*) find any locally minimal pair $(u, w)$ of $\pi$
      create a new item $v$ whose weight is $p_u + p_w$;
      replace $u$ by the item $v$ and delete $w$;
      (#) Move($v, \pi$);
      GW($\pi$);
      level[$u$] := level[$w$] := level[$v$] + 1;
    end;
```

Denote by $\mathrm{OPT}(i)$ the set of all alphabetic trees over the leaf-sequence $(1, \ldots n)$ which are optimal among trees in which i and $i+1$ are at the same level. Assume the pair $(i, i+1)$ is locally minimal. Let $\mathrm{OPT}_{moved}(i)$ be the set of all alphabetic trees over the leaf-sequence $\pi_{i,j}$ which are optimal among all trees in which leaves i and $i+1$ are at the same level, where $j = RightPos(i, i+1)$.

Two sets of trees OPT and OPT$'$ are said to be *level equivalent*, written $\mathrm{OPT} \cong \mathrm{OPT}'$, if, for each tree $T \in \mathrm{OPT}$, there is a tree $T' \in \mathrm{OPT}'$ such that $T' \cong T$, and vice versa.

Theorem 2.2
Let $(i, i+1)$ be a locally minimal pair. Then

(1) $\mathrm{OPT}(i) \cong \mathrm{OPT}_{moved}(i)$.
(2) $\mathrm{OPT}(i)$ *contains an optimal alphabetic tree T.*
(3) $\mathrm{OPT}_{moved}(i)$ *contains a tree T' with i, $i+1$ as siblings.*

Theorem 2.2 directly implies Theorem 2.1. Points (2) are (3) are simple. We prove them in this section for completeness. Point (1) is rather subtle, for if we drop the requirement that $i, i+1$ are at the same level, then this point is false for some weight sequences, *e.g.*, (7 8 13 14 1).

Our main contribution is the discovery and a structural proof of Point (1).

Description of the shift operations. We introduce two useful local operations, *RightShift* and *LeftShift* on trees. Both operations change the shape of an alphabetic tree locally without changing the order of items (as leaves). We describe in detail only the operation *LeftShift*, as *RightShift* is similar. Assume $v_1, v_2, \ldots, v_k$ are roots of disjoint subtrees $T_1, T_2, \ldots, T_k$, for $k \geq 2$, and the segments of leaves covered by these subtrees are disjoint and cover (left to right) a segment of consecutive leaves. We call such a sequence of nodes $(v_1, \ldots, v_k)$ a *cut*.

Let $p(v)$ denote the father of any node v in T, and $p'(v)$ denote the father of v in T', the tree resulting from applying the operation *LeftShift* to the cut. Create a new node w. Let $p'(w) = p(v_1)$. Let $p'(v_1) = p'(v_2) = w$. For $2 < t < k$, let $p'(v_t) = p(v_{t-1})$. To finish the description, we consider two cases. If v_k is the right son of $p(v_k)$, let $p'(v_k) = p(p(v_k))$. Otherwise, let v_{k+1} be the sibling of v_k, let $p'(v_k) = p(v_{k-1})$ and let $p'(v_{k+1}) = p(p(v_k))$. In both cases, $p(v_k)$ is deleted from the tree. For all other nodes, $p'(v) = p(v)$.

Proof of point (2) of theorem 2.2. Assume the levels of i and $i+1$ are different in some optimal tree T, hence they are not siblings in T. If $\mathit{level}_T(i) < \mathit{level}_T(i+1)$ then we can perform *LeftShift*$(i, i+1)$, obtaining a new tree T'. The level of i increases by 1 and the level of $i+1$ does not increase. On the other hand the level of $i+2$ decreases at least by one. Hence (since $p_{i+2} \geq p_i$ and $p_{i+1} \geq 0$) $\mathit{cost}(T') \leq \mathit{cost}(T)$, i and $i+1$ are siblings in T' and their levels are equal, and thus $T' \in \mathrm{OPT}(i)$. If $\mathit{level}_T(i) > \mathit{level}_T(i+1)$ the proof is similar; use *RightShift*$(i, i+1)$.

Proof of point (3) of theorem 2.2. Consider a tree $T \in \mathrm{OPT}_{\mathit{moved}}(i)$. If i, $i+1$ are not siblings then $1 < i < n-1$. After applying *LeftShift*$(i, i+1)$ they become siblings. Both of them go down but item $j+1$ goes one level up. Since $p_i + p_{i+1} - p_{j+1} \leq 0$ the resulting tree T' is still optimal.

3 The structural theorem

This section is devoted to the proof of Point (1) of theorem 2.2.

Informal overview of the proof of point (1) of correctness theorem. The crucial point is to show that the certain parts of trees in $\mathrm{OPT}(i)$ and $\mathrm{OPT}_{\mathit{moved}}(i)$ which are active with respect to the pair $(i, i+1)$ are "well-shaped" (in the sense defined below) and that this guarantees that the pair $(i, i+1)$ can be moved to the other side of such a part without affecting the level function. The point (1) of the correctness theorem is broken into the proof of the *movability lemma* and that of the *structural theorem.* The movability lemma is rather obvious. The structural theorem is proved by considering conditions of well-shaped segments and several cases. The proofs are by contradictions: if a certain condition is not satisfied for the optimal tree then using shift operations the tree is transformed into a tree of a smaller cost. This contradicts the optimality of the original tree. Point (1) of Theorem 2.2, and correctness of the GW algorithm, follow directly from Theorem 3.2 and Lemma 3.1.

Definition of well-shaped segment. Let $\mathrm{LCA}_T(u, w)$ denote the lowest common ancestor of nodes u and w in T.

We say that a set S of leaves of T is *h-isolated* if

1. For any $u \in S$, $\mathit{level}_T(u) \geq h$.
2. For any $u \in S$ and $w \notin S$, $\mathit{level}_T(\mathrm{LCA}(u, w)) \leq h$.

We say that a segment $[i \dots j]$ of consecutive items (leaves) is *left well-shaped* at level h in T if $\{i, \dots j\}$ is h-isolated and $level_T(i) = level_T(i+1) = h+1$. We define *right well-shaped* similarly (in this case $level_T(j-1) = level_T(j) = h+1$).

The leaves in the segment $[i \dots j]$ and all their ancestors at level at least h is called the *active window*. Note that the active window is a forest.

The window is said to be well-shaped iff the sequence of its leaves is left or right well-shaped. The introduction of windows is useful in visualizing local properties and rearrangements, as these rearrangements occur inside such windows. Trees in $\mathrm{OPT}(i)$ and $\mathrm{OPT}_{moved}(i)$ are illustrated in Figure 1 for $i = 3$ and for the weight sequence

$$(80\ 12\ 10\ 11\ 13\ 3\ 4\ 5\ 9\ 8\ 7\ 25)$$

The windows in trees T and T' are indicated by dotted lines. A window is also illustrated in Figure 2.

Lemma 3.1 (Movability lemma) *If the segment $[i \dots j]$ is left well-shaped, then the active pair of items $(i, i+1)$ can be moved to the other side of the segment by locally rearranging subtrees in the active window without changing the relative order of the other items and without changing the level function of the tree.*

Proof. The proof is straightforward. Let $h = level(i) - 1$. There are four cases, depending on whether i and $i+1$ are siblings, and on whether $\mathrm{LCA}(j, j+1) = h$. Figure 2 illustrates the proof in one case. We omit the details. ∎

If v is a node in a given alphabetic tree then we write p_v for the total weight of the leaves of the subtree rooted at v.

Theorem 3.2 (Structural theorem)
Assume $(i, i+1)$ is an lmp, $j = RightPos(i, i+1)$, $T \in \mathrm{OPT}(i)$, and $T' \in \mathrm{OPT}_{moved}(i)$. Then (i) *the segment $[i+2, \dots, j, i, i+1]$ is right well-shaped in T', and* (ii) *the segment $[i \dots j]$ is left well-shaped in T.*

Proof. We shall initially assume that all weights are positive.

If $j = i+1$, the theorem is trivial. Thus, without loss of generality, $j > i+1$.

Let $h = level_T(i) - 1 = level_T(i+1) - 1$ and $h' = level_{T'}(i) - 1 = level_{T'}(i+1) - 1$.

Claim A: : For any $u \in [i+2 \dots j]$ we have

(1) $level_T(u) \geq h$ and (2) $level_{T'}(u) \geq h'$.

Proof of Claim A. We show only the proof of point (1), as (2) has a very similar proof. The proof is by contradiction. Suppose the claim is false. Let k be the leftmost item such that $level_T(k) < h$. Let $T'' \in \mathrm{OPT}(i)$ be the tree obtained from T by applying $RightShift(i, i+1, v_1, \dots, v_r, k)$, where $v_2, \dots, v_r$ are at level h and v_1 is at level h if i and $i+1$ are siblings in T, level $h+1$ otherwise. If i

and $i+1$ are siblings in T, i and $i+1$ go up and only k goes down (see Figure 3). Otherwise, $i-1$ and $i+2$ go up and only k goes down. Since $p_k < p_i + p_{i+1} < p_{i-1} + p_{i+2}$, T'' has lower cost than T, a contradiction.

Claim B: : If $j < n$, then $level_T(\mathrm{LCA}_T(j, j+1) \leq h$.
Proof of Claim B. The proof is by contradiction. Suppose the claim is false. Let w be the ancestor of j and $j+1$ at level $h+1$. Let w_1 and w_2 be the sons of w.

Case 1: There exists a leaf k at level $\leq h$ in the segment. Pick the rightmost such k. Let $T'' \in \mathrm{OPT}(i)$ be the tree obtained from T by applying *LeftShift*$(k, v_1, \ldots, v_r, w_1, w_2)$, where $v_1, \ldots, v_r$ are nodes at level $h+1$. This case is illustrated in Figure 4, where $w_1 = v_6$. Since $j+1$ goes up and only k goes down, and $p_k < p_{j+1}$, T'' has smaller weight than T', a contradiction.

Case 2: There is no leaf at level $\leq h$ in the segment. Consider the cut $(i, i+1, v_1, \ldots, v_r, w_1, w_2)$ where $v_1, \ldots, v_r$ are nodes at level $h+1$. Let $T'' \in \mathrm{OPT}(i)$ be the tree obtained from T by applying *LeftShift*$(i, i+1, v_1, \ldots, v_r, w_1, w_2)$. Only i and $i+1$ go down, and $j+1$ and all the leaves of w_1 go up. Since $p_i + p_{i+1} \leq p_{j+1} < p_{w_1} + p_{j+1}$, T'' has smaller weight than T', a contradiction. This case is illustrated in Figure 5, where $w_1 = v_6$. This completes the proof of Claim B.

Claim C: : If $i > 1$, then $level_{T'}\mathrm{LCA}_{T'}(i-1, i+2) \leq h'$.
Proof of Claim C. The proof is by contradiction. Suppose the claim is false. This implies that there is a node w which is the ancestor of $i-1$ and $i+2$ at level $h'+1$. Let w_1 and w_2 be the sons of w. Let u be the parent of i and $i+1$. Let k be the leftmost node in the cut $(i+3, \ldots j, u)$ which is at level h'. Let $T'' \in \mathrm{OPT}_{moved}(i)$ be the tree obtained from T' by applying *RightShift*$(w_1, w_2, v_1, \ldots, v_r, k)$, where $v_1, \ldots, v_r$ are nodes at level $h'+1$. Since $i-1$ and $i+2$ go up and only k goes down, and $p_k \leq p_i + p_{i+1} < p_{i-1} + p_{i+2}$, T'' has smaller weight than T', a contradiction.

By Claims A, B, and C, the corresponding segments are well shaped in T and T'. This completes the proof in the case that all weights are positive.

We now consider, for completeness, the case where weights may be zero. If $p_i = p_{i+1} = 0$, then $j = i+1$, and the result is trivial. If $p_i + p_{i+1} > 0$, the proof is valid except for one problem, namely that in the proof of Claim B, we must consider the possibility that $p_{w_1} = 0$. Then $r > 0$, since otherwise $p_{i+2} = 0$, contradicting the fact that $(i, i+1)$ is an lmp. Let k be the rightmost leaf of v_r, and let $T'' \in \mathrm{OPT}(i)$ be the tree obtained from T by applying *LeftShift*(k, w_1, w_2). Since only k and leaves of zero weight go down, while $j+1$ goes up, and since $p_k < p_i + p_{i+1} \leq p_{j+1}$, T'' has smaller weight than T', a contradiction. ∎

4 Correctness of the Hu-Tucker algorithm: a simulation

The main idea of the Hu-Tucker algorithm is similar to that of the GW algorithm: combine two items which are very close and whose total weight is small. These items form an active pair which is later combined. However, now, a single item v representing the *combined* active pair is not moved. Instead, v becomes *transparent.* The original items are *opaque.* The algorithm keeps a *working sequence* of names of items, together with their types (opaque or transparent) and weights. A pair of items (u, w) is said to be *compatible* if they are visible to each other, *i.e.*, there is no opaque item between u and w in the current working sequence π. Denote by $pos(u)$ the position of the item u in the leaf-sequence (from left to right).

Definition of a minimal compatible pair. A pair (u, w) of compatible items is a *minimal compatible pair* (*mcp*, for short) if the total weight of (u, w) is minimal. If there are several pairs (u, w) with the same minimal total weight, the pair $(pos(u), pos(w))$ is defined to be the lexically smallest one. The last condition is called the *tie-breaking rule* of the Hu-Tucker algorithm.

Description of the HT algorithm. The HT algorithm works in the almost same way as the GW algorithm. Let π be the working sequence, which is initially the original list of items. In the statement $(*)$ in GW, we replace *locally minimal pair* by *minimal compatible pair* and the operation *Move* in the statement $(\#)$ by the statement "make v transparent."

Fix an input sequence of items of length n. Henceforth in this section we assume that there are no ties. This means that no two items in the working sequence π ever have the same weights. The case of ties will be handled in Section 5.

Denote by GW$'$ the *deterministic version* of the GW algorithm in which we choose each time the *globally minimal* lmp, which we call the *gmp.* instead of an arbitrary lmp. Observe that such a pair is not necessarily the rightmost locally minimal pair. This is one of the reasons why we considered a non-deterministic version of the GW algorithm, which chooses an arbitrary lmp, in Section 2. In case of ties, there is no reduction of the HT algorithm to the GW$'$ algorithm, as the following very simple example shows. If $n = 3$ and all items have equal weight, there are two possible alphabetic trees, both optimal. The GW$'$ algorithm finds one, while the HT algorithm finds the other. At the end of the paper we indicate how to deal with a non-deterministic version of the HT algorithm.

The working sequence of items in the HT algorithm consists of items of two types: *opaque* and *transparent*. Call such sequences *special* sequences. The working sequence produced by the GW$'$ algorithm makes no distinction between opaque and transparent items. For each special sequence λ define the sequence of items $MoveTransparent(\lambda)$ to be the sequence obtained by moving each transparent item w to the position immediately to the left of the nearest right larger neighbor of w, or to the end of the list if w has no right larger neighbor. If u is any item to the right of w before this motion and to the left of w after this motion, we say that w "floats over" u. We move transparent items one after another, starting with the rightmost transparent item.

Example. Assume transparent elements are primed. Then

$$\mathit{MoveTransparent}(18'\ 20'\ 14\ 12\ 17'\ 26'\ 13'\ 16\ 19) = (14\ 12\ 13\ 16\ 17\ 18\ 19\ 20\ 26)$$

The proof of correctness of the HT algorithm presented in this paper is by a *simulation* of the HT algorithm by the GW$'$ algorithm. The working sequences of algorithms are related through the function *MoveTransparent*, as stated in the simulation lemma, below. (See Figure 6.) This was also observed in [1]. For completeness we include our proof of the lemma in the appendix.

Lemma 4.1 (Simulation lemma) *Let* λ_i *be the working sequence of items after the* i^{th} *iteration of the* HT *algorithm and let* γ_i *be the working sequence of items after the* i^{th} *iteration of the* GW$'$ *algorithm. For* $0 \le i < n-1$*, let* (u_i, w_i) *be the globally minimal pair in* γ_i *in the sense of* GW$'$ *and let* (u'_i, w'_i) *be the mcp in* λ_i *. Then*
$\gamma_i = \mathit{MoveTransparent}(\lambda_i)$ *and* $(u_i, w_i) = (u'_i, w'_i)$ *for each* $1 \le i \le n-1$ (See Figure 6).

Correctness of the HT algorithm (in the case without ties) follows from the simulation lemma since we already know that GW$'$ is correct (as a version of GW). In the next section we show that the assumption that there are no ties can be dropped.

5 Resolving Ties

The problem of ties is rather subtle. A tie appears if two items (original or created by combining) have the same weight. *Correctness* of a tie-breaking rule means that the computed level function is the level function of some optimal alphabetic tree over the original sequence of items. There can be several globally minimal compatible pairs at the same time in the HT algorithm. Recall that in such a situation the original version of the HT algorithm applies the following *tie-breaking rule* (TBR):

(TBR): choose the *mcp* with lexically minimal pair of indices.

The fact that some tie-breaking rule is necessary is illustrated by the following example. Consider five items with the same weight. The possible history of the computation is:

$$1\,2\,3\,4\,5 \rightarrow 1\,2\,(3\,4)\,5 \rightarrow 1\,(2\,5)\,(3\,4) \rightarrow (1\,2\,5)\,(3\,4) \rightarrow (1\,2\,3\,4\,5)$$

The parenthesized sets are packages. The combine operations given above yield the following levels for the items 1, 2, 3, 4 and 5, respectively: 2, 3, 2, 2, 3. But, there is no full binary tree over leaf-sequence (1, 2, 3, 4, 5) with such a level function, so the algorithm is incorrect. We now prove that the rule TBR is correct. The proof also shows that we can always assume that there are no ties by changing the arithmetic. This does not affect the asymptotic complexity.

Theorem 5.1 *The tie-breaking rule* TBR *in the* HT *algorithm is correct.*

Proof. In the HT algorithm the weights are non-negative reals and the minimality of trees is with respect to the arithmetic of the reals. We show that the algorithm computes a minimal tree with respect to a more complicated arithmetic in R^2 without "knowing" it. Let R^2 be the additive ordered group of pairs of real numbers where the addition is component-wise: $(a, b) + (c, d) = (a + c, b + d)$, and the order is the lexical ordering of pairs of real numbers:

$$(a, b) < (c, d) \equiv ((a < c) \text{ or } (a = c \text{ and } b < d))$$

CASE 1 : All weights are strictly positive.

Let $\alpha = (p_1, \ldots, p_n)$ be the sequence of weights (non-negative reals). Denote

$$de\_tie(\alpha) = (p'_1, \ldots, p'_n) = (p_1, -2^{2n-1}),\ (p_2, -2^{2n-2}), \ldots, (p_n, -2^{2n-n}))$$

Observe that no integer can be expressed as two different sums of distinct powers of 2. This proves the following.

Claim A: No ties are possible in the HT algorithm working in the arithmetic of R^2 for the sequence of weights $de\_tie(\alpha)$

We know that HT is correct if there are no ties so it is correct when it works for $de\_tie(\alpha)$ in the arithmetic of R^2. The tree computed for $de\_tie(\alpha)$ is also an optimal alphabetic tree for α. In the arithmetic of R^2 the zero element is $(0, 0)$. Since $p_i > 0$, all elements of $de\_tie(\alpha)$ are positive. Fix the sequence α. Denote by π_i and π'_i the working sequences of items after the i^{th} iteration of HT applied respectively to α and $de\_tie(\alpha)$. Denote also by (u_i, w_i) and (u'_i, w'_i) the corresponding *minimal compatible pairs*. The claim below states that the history of the computation of the HT algorithm is the same in the usual arithmetic as in the arithmetic of R^2.

Claim B: For each $0 \leq i < n,\ \pi_i = \pi'_i$ and $(u_i, w_i) = (u'_i, w'_i)$.

We sketch the proof of Claim B. Refer to the second component of p'_i as the *tag weight*, or simply the *tag*. Then the *weight* refers to p_i. Claim B follows from correctness of the following loop invariant, which holds for the list π before and after every iteration:

Loop Invariant: If the weight of u is positive, then the tag of u is negative and is less than the sum of the tags of any subset of items to the right of u in π.

The relative order of two sums of tags over disjoint subsets of elements depends only on the first elements of these subsets. This follows from the following simple observation on base 2 representations of integers:

Remark 5.2 *Assume that* $0 \leq a_i, b_i \leq 1$ *for* $i \in [1, \ldots, n]$ *and that the sets* $A = \{i : a_i \neq 0\}$ *and* $B = \{i : b_i \neq 0\}$ *are disjoint, then*

$$\sum_{i=1}^{n} a_i(-2^{2n-i}) < \sum_{i=1}^{n} b_i(-2^{2n-i}) \ \Leftrightarrow\ min(A) < min(B)$$

Hence the rule TBR breaks ties in the same way in the usual arithmetic as it is done automatically (without this rule) in the arithmetic of R^2. The same pairs are combined and $\pi_{i+1} = \pi'_{i+1}$. This completes the proof of the claim.

CASE 2 : Some of the original weights are zero. If $p_i = 0$ then in the operation of "detying" we set $p_i' = (0, 2^{i-1})$, otherwise (for items with positive weight) the "detying" works as in Case 1.
Call a group of consecutive zero weighted items a *zero chain.* Each zero chain is processed from left to right, since the tag weights increase from left to right in the chain. This corresponds to the rule TBR. Afterwards zero weighted elements are combined with positive items, and the situation is essentially the same as in Case 1. This completes the proof. ∎

6 Final Remarks

We can consider a nondeterministic version HT′ of the Hu-Tucker algorithm. Define the extended order $\ll$ on the items in the sequence π:

$$u \ll w \text{ iff } (\mathit{weight}(u) < \mathit{weight}(w)) \text{ or } (\mathit{weight}(u) = \mathit{weight}(w) \text{ and } \mathit{pos}(u) < \mathit{pos}(w))$$

We say that a pair (u, w) of compatible items is a *locally minimal compatible pair* (lmcp, for short) if $w \ll v$ for each item v compatible with u and $u \ll q$ for each item q compatible with w. In other words u and w are the minimal compatible partners for each other. The HT′ algorithm is the same as the HT algorithm, except that it combines *any* lmcp of items.

Remark 6.1 *The* HT′ *algorithm is correct.*

Correctness is proved in a similar way as for the HT algorithm: simulate HT′ by GW in the case without ties. The working sequences in both algorithms are again related through the function *MoveTransparent* due to the lack of ties. Then the function *MoveTransparent* maps each working sequence in the algorithm HT′ into a corresponding unique sequence for the GW algorithm. The pairs combined in HT′ correspond to locally minimal pairs in GW. We remark that we can use a modified TBR in the GW algorithm to eliminate ties. The rule TBR will refer now to the smallest position of an original item contained in a given package. Hence we need only remember the smaller position of combined items. Then we can assume, without loss of generality, that there are no ties during execution of the GW algorithm.
The method does not use infinitesimals; only additional comparisons between positions of items are involved. This is useful in many situations, see [10, 11].

References

[1] A. M. Garsia and M. L. Wachs, A new algorithm for minimal binary search trees, *SIAM Journal of Computing* **6** (1977), pp. 622–642.

[2] T. C. Hu. A new proof of the T-C algorithm, *SIAM Journal of Applied Mathematics* **25** (1973), pp. 83–94.

[3] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM Journal of Applied Mathematics* **21** (1971), pp. 514–532.

[4] D. A. Huffman. A method for the constructing of minimum redundancy codes, *Proceedings of the IRE* **40** (1952), pp. 1098–1101.

[5] J. H. Kingston, A new proof of the Garsia-Wachs algorithm, *Journal of Algorithms* **9** (1988) pp. 129–136.

[6] M. M. Klawe and B. Mumey, Upper and Lower Bounds on Constructing Alphabetic Binary Trees, *Proceedings of the 4*th *ACM-SIAM Symposium on Discrete Algorithms* (1993), pp. 185–193.

[7] D. E. Knuth, *The Art of Computer Programming*, Addison–Wesley (1973).

[8] D. E. Knuth, Optimum binary search trees, *Acta Informatica* **1** (1971) pp. 14–25.

[9] L. L. Larmore and D. S. Hirschberg, A fast algorithm for optimal length–limited Huffman codes, *Journal of the ACM* **37** (1990), pp. 464–473.

[10] L. L. Larmore and T. M. Przytycka, The optimal alphabetic tree problem revisited, *Proceedings of the 21*st *International Colloquium*, ICALP'94, Jerusalem, LNCS 820, Springer-Verlag, (1994), pp. 251–262.

[11] L. L. Larmore, T. M. Przytycka, and W. Rytter, Parallel construction of optimal alphabetic trees, *Proceedings of the 5*th *ACM Symposium on Parallel Algorithms and Architectures* (1993), pp. 214–223.

[12] P. Ramanan, Testing the optimality of alphabetic trees, *Theoretical Computer Science* **93** (1992), pp. 279–301.

[13] F. F. Yao, Efficient dynamic programming using quadrangle inequalities, *Proceedings of the 12*th *ACM Symposium on Theory of Computing* (1980), pp. 429–435.

Appendix

Proof. (of the *simulation lemma*)

The crucial point is to prove that the sequence of combined pairs is the same for both algorithms. This reduces to the two claims below.

Claim A: Assume that $\gamma = \mathit{MoveTransparent}(\lambda)$. Then if (u, w) is the globally minimal pair in γ, the items u, w are visible to each other in λ.
Proof of Claim A. Observe that if the item x is transparent in λ and has "floated over" at least k items when performing $\mathit{MoveTransparent}(\lambda)$ then the left k neighbors of x in γ have smaller weight than x. The proof of the claim is by contradiction. Assume that u, w are not visible to each other, *i.e.*, there is an opaque item q between u and w in λ. There are two cases.

Case 1: In λ, w is before u. Then w is transparent because it "floats over" u, and $\mathit{weight}(u) < \mathit{weight}(w)$. Let q' be the predecessor of u in γ. If $\mathit{weight}(q') < \mathit{weight}(w)$, then (u, w) is not minimal in λ, a contradiction. If q' is transparent, then, since q' does not "float over" u, $\mathit{weight}(q') < \mathit{weight}(u) < \mathit{weight}(w)$, which implies that (u, w) is not minimal in λ, a contradiction. Thus $\mathit{weight}(q') > \mathit{weight}(w)$ and q' is opaque.

Since w cannot "float over" q', we know that q' is to the left of w in γ. We have the situation:

$$\ldots q' \ldots w \ldots q \ldots u \ldots \overset{\mathit{MoveTransparent}}{\Rightarrow} \ldots q'uw \ldots$$

But q has no possible place in γ, as it must remain to the left of u and to the right of q', a contradiction.

Case 2: In λ, u is before w. Then u is transparent because it "floats over" q. Since u does not "float over" w, $\mathit{weight}(u) < \mathit{weight}(w)$. Let q' be the predecessor of u in γ. If $\mathit{weight}(q') < \mathit{weight}(w)$, then (u, w) is not minimal in λ, a contradiction. If q' is transparent, then, since q' does not "float over" u, $\mathit{weight}(q') < \mathit{weight}(u) < \mathit{weight}(w)$, which implies that (u, w) is not minimal in λ, a contradiction. Thus $\mathit{weight}(q') > \mathit{weight}(w)$ and q' is opaque. Since u cannot "float over" q', we know that q' is to the left of u in γ. We have the situation:

$$\ldots q' \ldots u \ldots q \ldots w \ldots \overset{\mathit{MoveTransparent}}{\Rightarrow} \ldots q'uw \ldots$$

Now q must remain to the left of w and to the right of q', a contradiction. This completes the proof of Claim A.

Claim B: If (u, w) is the minimal compatible pair in λ then the items u, w are adjacent in γ.
Proof of Claim B. Assume there is an item q between u and w in γ. Observe the possible scenario of the operation *MoveTransparent*. The items are been processed (moved to right) in *right-to-left* order. Assume u is before w in λ. First w is processed, then all items between u and w. Finally u is processed. All elements between u and w in λ are of larger weight than w (since (u, w) is the mcp) and they "float over" w. Immediately after moving u, the items u and w become adjacent. Hence the item which is inserted between u and w in γ is to the left of u in λ. We have the following situation:

$$\ldots q \ldots u \ldots w \ldots \overset{\mathit{MoveTransparent}}{\Rightarrow} \ldots u \ldots q \ldots w \ldots$$

If q is visible from u then q is a better partner for u than w since q stopped before "floating over" w, *i.e.*, (q, u) would be a smaller pair of compatible items. This contradicts the fact that (u, w) is the *mcp*. Otherwise

q is not visible from u. Let q' be the opaque item visible from u between q and u. (Such an item must exist.) Since q "floated over" q', we know that $weight(q') < weight(q)$. Furthermore, $weight(q) < weight(w)$ since q stopped before w. Thus q' is a better partner for u than w, and q' and u are visible to each other. We have a contradiction since the pair (q', u) is a better choice of mcp than (u, w). This completes the proof of Claim B.

Assume that $\gamma = \mathit{MoveTransparent}(\lambda)$. Claim A and Claim B imply that the minimal pair combined in γ is the same as the pair combined in λ. We have:

$$\mathit{MoveTransparent}(\mathrm{HT}(\lambda)) = \mathrm{GW}'(\gamma) \ \text{ if } \ \gamma = \mathit{MoveTransparent}(\lambda)$$

where GW$'$ and HT denote here one iteration of the GW$'$ and HT algorithms, respectively, on the working sequence of items. (This is shown for an example sequence in Figure 3.) The proof works by induction on the number of iterations. Let λ_i, γ_i, be the lists after i iterations. Then $\lambda_0 = \gamma_0$ is the initial sequence of items, and $\mathit{MoveTransparent}(\lambda_i) = \gamma_i$ implies $\mathit{MoveTransparent}(\lambda_{i+1}) = \gamma_{i+1}$. ▮

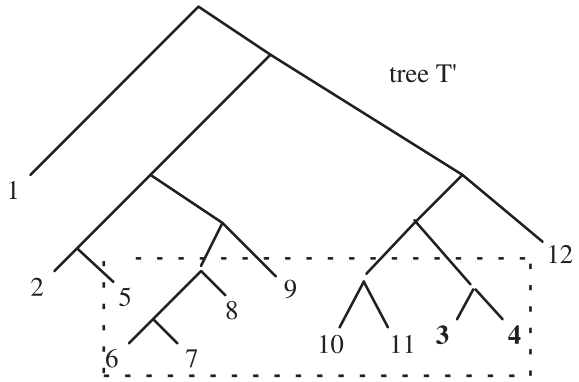

Figure 1: Example trees $T \in \mathrm{OPT}(i)$ and $T' \in \mathrm{OPT}_{moved}(i)$, where $i = 3$ and the sequence of weights is (80 12 10 11 13 3 4 5 9 8 7 25). Observe that the windows are *well shaped.*

Figure 2: $i, i+1$ can be moved to the other side of a *well shaped* window.

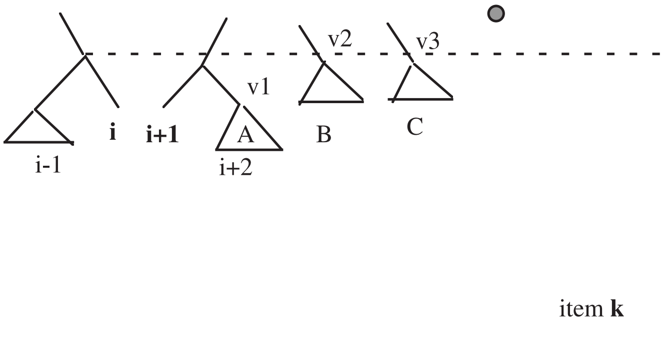

Figure 3: Graphical illustration of the proof that the leaves in the active segment are not above level h. Otherwise we can rearrange the tree and increase the level of the "bad" node $v4$. Then the level of the active pair decreases and the cost is improved.

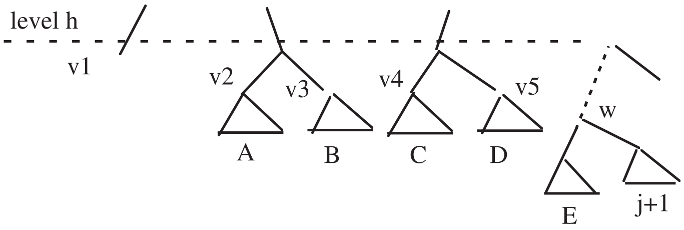

Figure 4: Graphical illustration of Case 1 of the proof that $LCA(j, j+1) \geq h$. If it is not the case then the tree can be rearranged and the cost will be improved.

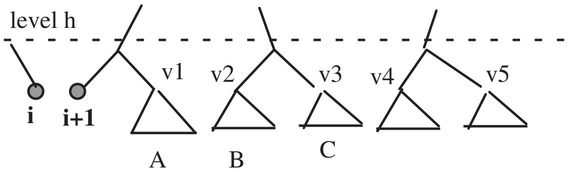

Figure 5: Graphical illustration of Case 2 of the proof that $LCA(j, j+1) \geq h$.

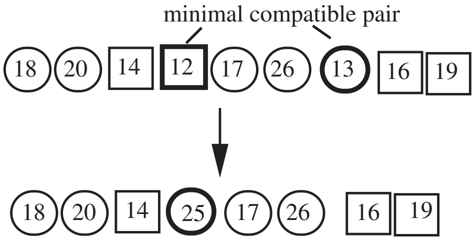

Figure 6: The simulation of the HT algorithm by GW in one iteration. Transparent items are circled.