Theoretical Computer Science 262 (2001) 101–115

Theoretical Computer Science

www.elsevier.com/locate/tcs

Minimizing roundoff errors of prefix sums via dynamic construction of Huffman trees ☆

Ming-Yang Kao[a,1], Jie Wang[b,*,2]

[a] *Department of Computer Science, Yale University, New Haven, CT 06520, USA*
[b] *Department of Mathematical Sciences, University of North Carolina at Greensboro, Greensboro, NC 27402, USA*

Received 28 September 1999; accepted 23 March 2000

Abstract

The prefix-sum operation, which returns all prefix sums on a sequence of numbers, plays an important role in many applications. We study how to efficiently evaluate prefix sums on positive floating-point numbers such that the worst-case roundoff error of each sum is minimized. A direct approach to this problem builds a Huffman tree for each prefix subsequence from scratch, requiring exactly quadratic time for *every* input X. We can do better by taking advantage of the current Huffman tree to build the next Huffman tree, using dynamic insertions and deletions on Huffman trees. Consequently, subquadratic time suffices for various input patterns. We also provide experimental comparisons of all the algorithms studied in this paper on inputs that are randomly and uniformly generated. © 2001 Elsevier Science B.V. All rights reserved.

1. Introduction

We study how to efficiently evaluate all prefix sums of a finite sequence of floating-point numbers such that the worst-case roundoff error of each sum is minimized. Computation of prefix sums, first suggested by Iverson as an operation for the language APL [9], is a primitive building block in many applications (e.g., see [1, 2]). Previous research on prefix sums focused on their implementations and applications in various settings (e.g., see [1–3, 6, 15, 17, 19, 20]). For inputs of floating-point numbers, the only known algorithm that can minimize worst-case roundoff errors of prefix sums requires

☆ Some of the results in this paper appeared in the Proceedings of the 25th International Colloquium on Automata, Languages, and Programming, 1998, pp. 376–386.

* Corresponding author.

E-mail address: wang@uncg.edu (J. Wang).

[1] Supported in part by NSF under Grant CCR-9531028.

[2] Supported in part by NSF under Grants CCR-9424164 and by the University of North Carolina at Greensboro through a research grant.

a quadratic time on *every* input sequence. This paper presents improvements using dynamic algorithms.

Summation of a set of nonzero floating-point numbers is ubiquitous in numerical analysis, where the central issue is to obtain a summation efficiently with roundoff errors as small as possible. This topic has been studied extensively (e.g., see [5, 7, 8, 12–14, 18, 21]). Kao and Wang [12] recently showed that, when the input set contains both positive and negative floating-point numbers, it is NP-hard to find a summation of these numbers with the minimum worst-case roundoff errors. This paper only considers inputs that are either all positive or all negative. Without loss of generality, we assume the former.

Let $X = \langle x_1, \dots, x_n \rangle$ be a sequence of positive floating-point numbers. Denote by X_k the kth *prefix subsequence* $\langle x_1, \dots, x_k \rangle$ of X for $k = 1, \dots, n$. We want to evaluate all prefix sums $S_k = x_1 + x_2 + \cdots + x_k$, which we call the kth *prefix sum*. We use the standard model of floating-point arithmetic for error analysis:

$$fl(x + y) = (x + y)(1 + \delta_{xy}),$$

where $|\delta_{xy}| \leqslant \alpha$, and $\alpha \ll 1$ is the unit roundoff.

Since operator $+$ is applied to two operands at a time, any method that adds all numbers in X_n corresponds to a binary addition tree of n leaves and $n - 1$ internal nodes, and vice versa; where a leaf node is a number x_i and an internal node is the sum of its two children. Different ways of summing X_n yield different addition trees, which may produce different computed sums $\hat{S}_n$ in floating-point arithmetic. We want to find an algorithm that minimizes the error $E_n = |\hat{S}_n - S_n|$. Let $I_1, \dots, I_{n-1}$ be the internal nodes of an addition tree T over X_n. Since α is very small even on a desktop computer, any product of more than one α is negligible in our consideration, resulting in the following approximation: $E_n \approx |\sum_{i=1}^{n-1} I_i \delta_i| \leqslant \alpha \sum_{i=1}^{n-1} |I_i|$. This gives rise to the following definitions [11, 12].

- The *worst-case error* of T, denoted by $E(T)$, is $\alpha \sum_{i=1}^{n-1} |I_i|$.
- The *cost* of T, denoted by $C(T)$, is $\sum_{i=1}^{n-1} |I_i|$.

Our goal here is to minimize $E(T)$, which is equivalent to minimizing $C(T)$. The following notations are useful:

- E_n^* is the minimum worst-case error over all orderings of evaluating S_n.
- S_n^* denotes a computed sum of S_n with error E_n^*.
- $T_{\min}$ denotes an optimal addition tree over X_n, i.e. $E(T_{\min}) = E_n^*$ or equivalently $C(T_{\min}) = C_n^*$.

The *floating-point prefix-sum* problem (FPPS) asks for all S_k^* for $k = 1, \dots, n$.

Let T_n be an addition tree over X_n, then $C(T_n) = \sum_{i=1}^{n} x_i \cdot d_i$, where d_i is the number of edges on the path from the root to the leaf x_i in T_n. Thus, finding an optimal addition tree over X_n is equivalent to constructing a Huffman tree over X_n.

We observe that solving FPPS by constructing a Huffman tree for each subsequence incurs a quadratic lower time bound. On the other hand, as discussed in Section 2, quadratic time suffices for solving FPPS, since by using van Leeuwen's algorithm [16], we can construct a Huffman tree for each subsequence from scratch in linear time on sorted inputs. However, this approach requires quadratic time for every input X. In Sections 3 and 4 we present improved algorithms by taking advantage of the current Huffman tree to build the next tree, using dynamic deletions and insertions on Huffman trees. Our dynamic algorithms can solve FPPS in subquadratic time for various input patterns. Finally, in Section 5 we provide experimental comparisons of all the algorithms studied in this paper on inputs that are randomly and uniformly generated. Our experimental results show that the dynamic algorithms are faster algorithms in practice.

2. List-insertion and list-deletion prefix sums

Since finding an optimal addition tree over X_n is equivalent to constructing a Huffman tree over X_n, we can solve FPPS using the following two general algorithms.

Algorithm 1. *Insertion prefix sums* (*IPS*)

```
(1) Set $k \leftarrow 1$.
(2) Construct a Huffman tree for $X_k$ and return the root value, which is $S_k^*$.
(3) If $k < n$, set $k \leftarrow k + 1$ and go to Step 2.
```

Algorithm 2. *Deletion prefix sums* (*DPS*)

```
(1) Set $k \leftarrow n$.
(2) Construct a Huffman tree for $X_k$ and return the root value, which is $S_k^*$.
(3) If $k > 1$, set $k \leftarrow k - 1$ and go to Step 2.
```

Both IPS and DPS have quadratic lower-time bounds in the worst case. For instance, if we use IPS to solve FPPS on input $X = \langle b_{n+2}, b_{n+1}, \ldots, b_3 \rangle$, where b_k is the kth Fibonacci number, then no previous trees can be reused to build a Huffman tree over X_k. Thus, no matter what methods are used, $k-1$ new additions are required to compute S_k^*. The case for DPS is similar.

When X_n is sorted, van Leeuwen [16] showed that a Huffman tree can be constructed on X_n in $\Theta(n)$ time as follows.

Algorithm 3. The algorithm takes a list L as input, where L contains X_n in nondecreasing order.

```
(1) Create another sorted list $L'$, which is empty initially.
(2) Until $L$ is empty, extract the first two smallest numbers $x$ and $y$ from both $L$ and
    $L'$, and insert $x + y$ at the end of $L'$.
(3) If $L'$ has more than one element, repeat Step 2 on $L'$ (i.e., $L'$ becomes $L$ and $L$
    becomes $L'$ in Step 2). Repeat Step 2, alternating between the lists, until only one
    element remains in both the lists.
(4) Return the last element.
```

Lemma 1 (van Leeuwen [16]). *Assume that X_n is sorted in nondecreasing order, then Algorithm 3 constructs a Huffman tree over X_n in $\Theta(n)$ time.*

Hence, we can solve FPPS in $\Theta(n^2)$ time by first sorting X_n, and then constructing a Huffman tree for each subsequence X_k using Algorithm 3 as follows.

Algorithm 4. *List-deletion prefix sums (LDPS)*

```
The algorithm takes $X$ as input.
(1) Sort the numbers $x_1, \ldots, x_n$ in nondecreasing order using a O($n \log n$)-time-sorting
    algorithm. Store the sorted numbers in a sorted list $L$.
(2) Set $k \leftarrow n$.
(3) While $k > 1$, repeat the following steps.
    (a) Use Algorithm 3 on $L$ to compute $S_k^*$.
    (b) Set $L \leftarrow L - \{x_k\}$ and decrease $k$ by 1.
```

Lemma 2. *LDPS solves FPPS in* $O(n^2)$ *time. Moreover, LDPS requires* $\Omega(n^2)$ *time for every input X.*

Proof. Straightforward. □

Similarly, we can obtain *list-insertion prefix sums* (LIPS) to solve FPPS in quadratic time by substituting insertion for deletion. We use a balance binary search tree, such as a red-black tree [4], to store X_k so that inserting a new number x_{k+1} into the sorted list of X_k only takes $O(\log k)$ time.

Both LDPS and LIPS require $\Omega(n^2)$ time for every input X. However, we note that for some inputs X, FPPS can actually be solved in subquadratic time. For example, consider the following commonly used procedure that evaluates all prefix sums in linear time: Set $S \leftarrow \emptyset$ and $k \leftarrow 1$; while $k \leqslant n$, set $S \leftarrow S + x_k$, output S, and set $k \leftarrow k+1$. If $x_k \geqslant S_{k-1}$ for all k, then this algorithm evaluates all S_k^* because the ordering of adding the numbers in X_k corresponds to a Huffman tree over X_k.

In Sections 3 and 4 we present algorithms that capture such phenomena, using dynamic operations on Huffman trees.

3. Huffman-tree-deletion prefix sums

Given a Huffman tree T over X_n and a value $x \in X$, we want to delete x from T so that the resulting tree is still a Huffman tree.

A binary addition tree over X_n is said to satisfy the *sibling property* if the nodes can be numbered in the nondecreasing order of their values so that for $i = 1, \ldots, n-1$, nodes $2i-1$ and $2i$ are siblings and their parent is higher in the numbering. We call i the *sibling number* of node i. The sibling numbers correspond to the order in which the nodes are combined: Nodes 1 and 2 are combined first, nodes 3 and 4 are combined second, and so on. For any X_n, there must be a Huffman tree over X_n that satisfies the sibling property, and any binary addition tree over X_n that satisfies the sibling property must be a Huffman tree. Hence, without loss of generality, we assume that all the Huffman trees we deal with satisfy the sibling property.

Our goal is to carryout deletion on Huffman trees in $O(\ell)$ time, where ℓ is the number of nodes whose sibling numbers are greater than the sibling number of the node deleted; hence, ℓ ranges from 1 to $2n-2$. We can then use deletion to calculate the next S_k^* by taking advantage of the current S_k^* to reduce redundant computation.

We use a doubly linked list to store a tree in which each node has three pointers: one to its parent, one to its left child, and one to its right child. A list A of size n is used to store pointers such that $A[i]$ points to the ith node in the tree. The *weight* of a node refers to the numerical value of a node.

The idea for deletion is to keep replacing the current node i, starting from the node to be deleted, by node j, where $j=i+1$ if node $i+1$ is not the parent of node i, or $j=i+2$ otherwise. The weight of the parent of node i is updated accordingly. The following two constant-time operations are useful.

– Replace(i,j): Replace the left (respectively, right) child of i by the left (respectively, right) child of j, and replace w_i by w_j.
– WeightUpdate(i,j,k): Set the weight of node i's parent to $w_j + w_k$.

The effect of calling Replace(i,j) moves the entire left (respectively, right) subtree of node j to become the left (respectively, right) subtree of node i in constant time. Node j may then be viewed as a dummy leaf. If we color the node to be replaced black, then the deletion can be viewed as the process of pushing the black node up until the root is reached. The black node and the root will then be deleted from the tree. The following algorithm deletes node i_0 from a Huffman tree T of n leaves.

Algorithm 5. *Huffman-tree deletion*

The algorithm takes (T, i_0) as input, where T is a Huffman tree of n leaves and i_0 is the sibling number of the node to be deleted.

Set $i \leftarrow i_0$, and $m \leftarrow 2n-1$.

Case A: Node i is a right child. We have the following three subcases.

Case A1: Node $i+1$ is the root.

Set $m \leftarrow m-2$ and return $A[m]$. The algorithm ends. (*Remark:* Now root $m-2$ is the root of the new tree.)

Case A2: Node $i+1$ is not the root but is the parent of node i.

First call WeightUpdate$(i, i-1, i+2)$, next call Replace$(i, i+2)$. Then set $i \leftarrow i+2$.

Case A3: Node $i+1$ is neither the root nor the parent of node i.

(1) First call WeightUpdate$(i, i-1, i+1)$.
(2) If $w_{i-1} \leqslant w_{i+1}$, call Replace$(i, i+1)$; otherwise, call Replace$(i, i-1)$ and then call Replace$(i-1, i+1)$.
(3) Increase i by 1. Node i is now a left child, go to Case B.

Case B: Node i is a left child.

If $w_{i-1} \leqslant w_{i+1}$, then call Replace$(i, i+1)$; otherwise, call Replace$(i, i-1)$ and Replace$(i-1, i+1)$. Set $i \leftarrow i+1$. Node i is now a right child, go to Case A.

Theorem 3. *Let T be a Huffman tree over X and let i_0 be the sibling number of an $x \in X$ in T. Then deleting x from T using Algorithm 5 results in a Huffman tree T' of $n-1$ leaves in $\Theta(2n - i_0)$ time.*

Proof. It suffices to show that T' satisfies the sibling property. Let $i = i_0$. Let w_j denote the weight at node j in T. Let w'_j denote the weight at node j in T'. Note that $w'_j = w_j$ if node j's content is never updated.

Algorithm 5 increases i until the new root is reached. Hence, it suffices to show that the sibling property is satisfied up to node i during the deletion process. Namely, for every $j < i$ and $k < i$, $j < k$ if and only if $w'_j \leqslant w'_k$, where nodes $2l-1$ and $2l$ are siblings, and the weight of each internal node is the sum of the weight of its two children. We will prove this property using an induction argument on the value of i, $i = i_0, \ldots, 2n-2$. We will also prove for $i > 2$, $w'_{i-2} \leqslant w_{i+1}$ in the meantime. This inequality is useful in the proof. The induction basis is obvious because we begin with T.

Case A: Node i is a right child.

Case A1: Node $i+1$ is the root. Then T' is the subtree rooted at node $i-1$, which satisfies the sibling property by induction hypothesis. We are done.

Case A2: Node $i+1$ is not the root but is the parent of node i. Since node $i+1$ is the parent of node i, node i cannot have any cousin on the right, and node $i+1$ cannot have any cousin on the left. In other words, the tree must be in the form as shown in Fig. 1(a). Therefore, node $i+2$ must be a leaf and must be the right sibling of node $i+1$. Let $x = w'_{i-1}$ and $z = w_{i+2}$. Since the nodes that have been updated so far must be descendants of node $i+1$, $x = w_{i-1} - w_{i_0} + w_i < w_{i+1}$. Thus, $x < w_{i+2} = z$. Since the sibling property is satisfied up to node i, we can replace node i by node $i+2$, and reset w_{i+1} to be $x+y$, which preserves the sibling property up to node $i+2$ (see Fig. 1(b)). Let $j = i+2$, which is the new value of i. Node j is still a right child and Case A applies. We also have $w'_{j-2} = z = w_{i+2} \leqslant w_{i+3} = w_{j+1}$.

Case A3: Node $i+1$ is neither the root nor the parent of node i. Assume that node $i-1$ is an internal node and node $i+1$ is a leaf in T. Note that $w'_{i-1} \geqslant w_{i-1}$ because a node is always replaced by a node with the same or larger weight. But it is possible that $w'_{i-1} > w_{i+1}$ if node $i+1$ is a leaf. Assume that this is the case. Since $w'_{i-2} \leqslant w_{i+1}$ and the sibling property is satisfied up to node i, we can replace node i by node $i-1$, replace node $i-1$ by node $i+1$, and still preserve the sibling property up to node $i+1$.

For the case that $w'_{i-1} \leqslant w_{i+1}$, we have $w'_{i-1} \leqslant w'_{i+1}$ by the induction hypothesis and the fact that a deleted node is always replaced by a node with the same or larger weight. Thus, we can simply replace node i by node $i+1$ without affecting the sibling property. Hence, the sibling property is satisfied up to node $i+1$. Let $j = i+1$, which is the new value of i. Then node j becomes a left child, and Case B applies.

Next, we prove $w'_{j-2} \leqslant w_{j+1}$. If node $j-2$ is a leaf, then w'_{j-2} equals w_{i-1} or w_{i+1}. So $w'_{j-2} \leqslant w_{i+2} = w_{j+1}$. We now focus on the case that node $j-2$ is an internal node. Let k and $k+1$ be the numberings of node $j-2$'s children. If node $j-1$ is also

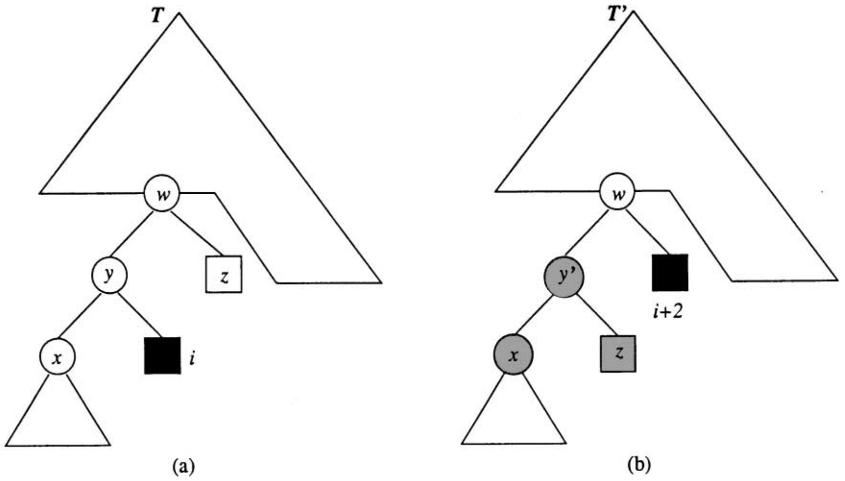

Fig. 1. Case A2, where $y' = x + z$. The black node is the node to be replaced. Shaded nodes and the black node are the ones being visited.

an internal node, then it is the right sibling or the left uncle of node $j - 2$. In either case, $w'_k \leqslant w_{k+1}$ and $w'_{k+1} \leqslant w_{k+2}$, and so $w'_{j-2} \leqslant w_{k+1} + w_{k+2} = w_{i+1} \leqslant w_{i+2} = w_{j+1}$. If node $j - 1$ is a leaf, then $w'_{i-1} \leqslant w_{i+1} \leqslant w_{i+2}$, and thus $w'_{j-2} \leqslant w_{j+1}$.

Case B: Node i is a left child. This case is handled in a similar manner as in Case A3, and we omit the details here.

Next, we analyze the running time of the algorithm. Note that replacing a node by another node takes O(1) time, for it only involves O(1) pointer manipulations. Also, nodes whose numberings are smaller than i_0 are never visited, and each node whose numbering is at least i_0 is visited at most twice for its replacement and a possible weight update. Consequently, the running time of Algorithm 5 is $\Theta(2n - i_0)$, since there are exactly $2n - 1$ nodes in a Huffman tree of n leaves. □

Fig. 2 demonstrates the deletion process using Algorithm 5.

Algorithm 6. *Huffman-tree-deletion prefix sums* (*HDPS*).

The algorithm takes X as input.

```
(1) Construct a Huffman tree $T_n$ over $X_n$. Set $k \leftarrow n$.
(2) While $k > 1$, repeat the following steps:
    (a) Output the value of the root of $T_k$, and find $x_k$ from the list $A$.
    (b) Use Algorithm 5 to delete $x_k$ and produce a Huffman tree $T_{k-1}$ over $X_{k-1}$.
    (c) Set $A \leftarrow A - \{A[2k-2], A[2k-1]\}$, and decrease $k$ by 1.
```

Theorem 4. *Let the sibling number of x_k in T_k be $s(k)$, where $1 \leqslant s(k) \leqslant 2k - 2$. Algorithm 6 solves FPPS (in the order of $k = n$ to 1) in $\Theta(f(n) + \sum_{k=1}^{n}(2k - s(k)))$ time, where $f(n) = n \log n$ if X_n is not sorted, and n otherwise.*

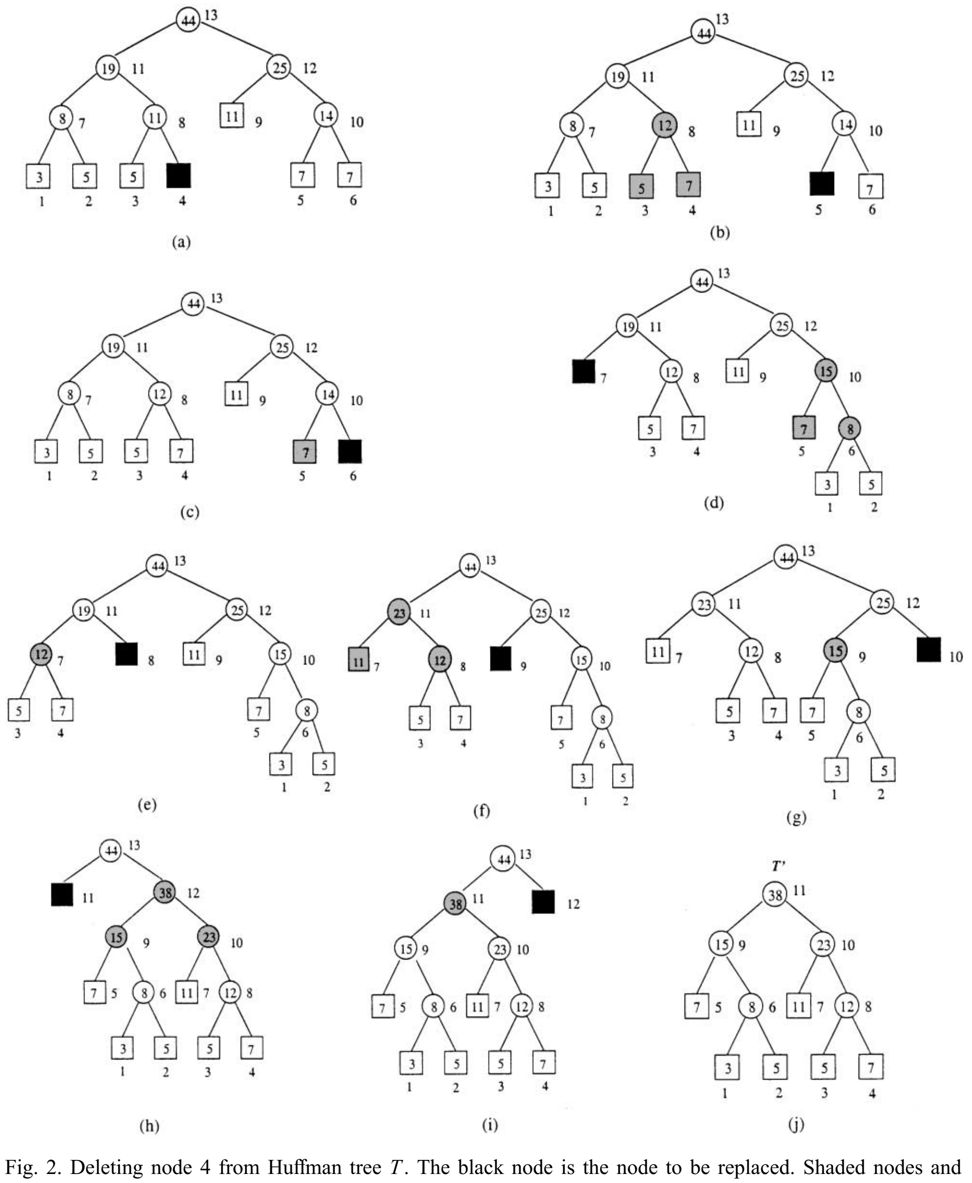

Fig. 2. Deleting node 4 from Huffman tree T. The black node is the node to be replaced. Shaded nodes and the black node are the ones being visited. (a) Case A3. (b) Case B. (c) Case A3. (d) Case B. (e) Case A3. Now $i=8$, $w'_{i-1} < w_{i+1}$. (f) Case B. (g) Case A3. (h) Case B. (i) Case A1.

Proof. The $\Theta(f(n))$ cost comes from the construction of the first Huffman tree T_n over X_n. Once T_n is constructed, it follows from Theorem 3 that for each x_k with $k \leqslant n$, Step 2 of Algorithm 6 takes $\Theta(2k - s(k))$ time to delete node $s(k)$. This completes the proof. □

Theorem 5. *Let $\theta(k)$ be an integer function with $0 \leqslant \theta(k) < k$. If for all k, there exist $x_{i_1}, \ldots, x_{i_{\theta(k)}} \in X_k$ with $1 \leqslant i_1 < \cdots < i_{\theta(k)} < k$ such that $x_k > \sum_{j=1}^{\theta(k)} x_{i_j}$, then Algorithm* 6 *solves FPPS in $\Theta(f(n) + \sum_{k=1}^{n}(k - \theta(k)))$ time, where $f(n) = n \log n$ if X_n is not sorted, and n otherwise.*

Proof. Let T be a Huffman tree over X and x a node in T. Denote by $sib(x)$ the sibling number of node x. It suffices to show that $sib(x_k) \geqslant 2\theta(k)$ for every k.

Sort X_k in increasing order such that

$$x'_1 \leqslant x'_2 \leqslant \cdots \leqslant x'_k,$$

where $X_k = \{x'_1, \ldots, x'_k\}$. Hence, $x_k = x'_j$ for some j. Since, by assumption, $x_k > \sum_{j=1}^{\theta(k)} x_{i_j}$, where $1 \leqslant i_1 < \cdots < i_{\theta(k)} < k$, we have the following two inequalities:

$$j > \theta(k), \tag{1}$$

$$x'_j > \sum_{i=1}^{\theta(k)} x'_i. \tag{2}$$

Let $\ell \geqslant \theta(k)$ be the largest index and $s \leqslant j$ be the smallest index such that

$$x'_s > \sum_{i=1}^{\ell} x'_i.$$

Then $sib(x_k) = sib(x'_j) \geqslant sib(x'_s)$. We will show that $sib(x'_s) \geqslant 2\ell$. We have the following two cases.

Case A: $s = \ell + 1$. It follows from Inequality (2) that all x'_i $(i = 1, \ldots, \ell)$ must be added together before x'_j is added. In other words, x'_j is added to $C = \sum_{i=1}^{\ell} x'_i$, which corresponds to an addition tree of exactly $2\ell - 1$ nodes whose sibling numbers are less than $sib(x'_s)$. Hence, $sib(x'_s) \geqslant 2\ell$.

Case B: $s > \ell + 1$. Let $U = \{x'_1, \ldots, x'_\ell\}$ and $V = \{x'_{\ell+1}, \ldots, x'_{s-1}\}$. We have the following two subcases.

Case B1: x'_s is added to a number C_1 and there are numbers $C_2, \ldots, C_m$ such that $C_1 \leqslant x'_s \leqslant C_2 \cdots \leqslant C_m$, where C_r $(r = 1, \ldots, m)$ is the summation of u_r numbers from U and v_r numbers from V, and each number in U and V occurs exactly once in all these numbers. Hence, $\sum_{r=1}^{m} u_r = |U| = \ell$ and $\sum_{r=1}^{m} v_r = |V| = s - \ell - 1$.

Since all the summations needed to obtain $C_1, \ldots, C_m$ were done before x'_s is added, where each C_r has exactly $2(u_r + v_r) - 1$ nodes, there are exactly $\sum_{r=1}^{m}[2(u_r + v_r) - 1]$ nodes whose sibling numbers are less than $sib(x'_s)$. By Inequality (2), we have $v_r > 0$ for all $r = 2, \ldots, m$. This implies that $m - 1 \leqslant |V| = s - \ell - 1$. Thus, $s - m \geqslant \ell$. We have

$$\begin{aligned} sib(x'_s) &\geqslant \sum_{r=1}^{m} [2(u_r + v_r) - 1] + 1 \\ &= 2\sum_{r=1}^{mu} u_r + 2\sum_{r=1}^{m} v_r - m + 1 \end{aligned}$$

$$\begin{aligned}
&= 2\ell + 2(s - \ell - 1) - m + 1 \\
&= 2s - m - 1 \\
&= s + (s - m) - 1 \\
&\geqslant (\ell + 2) + \ell - 1 \\
&= 2\ell + 1.
\end{aligned}$$

Case B2: x'_s is added to a number C_1 and there are numbers $C_2, \ldots, C_m$ such that $x'_s \leqslant C_1 \leqslant \cdots \leqslant C_m$, where C_r $(r = 1, \ldots, m)$ is the summation of u_r numbers from U and v_r numbers from V, and each number in U and V occurs exactly once in all these numbers. Hence, $\sum_{r=1}^{m} u_r = |U| = \ell$ and $\sum_{r=1}^{m} v_r = |V| = s - \ell - 1$.

Similar to Case B1 we know that there are exactly $\sum_{r=1}^{m}[2(u_r + v_r) - 1]$ nodes whose sibling numbers are less than $sib(x'_s)$. By Inequality (2), we have $v_r > 0$ for all $r = 1, \ldots, m$. This implies that $m \leqslant |V| = s - \ell - 1$. Thus, $s - m \geqslant \ell + 1$. Similar to Case B1 we have

$$\begin{aligned}
sib(x'_s) &= s + (s - m) - 1 \\
&\geqslant (\ell + 2) + (\ell + 1) - 1 \\
&= 2\ell + 2.
\end{aligned}$$

This completes the proof. $\square$

The following corollary of Theorem 5 is straightforward.

Corollary 6. (1) *If there exist an* ε *with* $0 < \varepsilon < 1$ *such that for all* k, $\theta(k) \geqslant k - n^{\varepsilon}$, *then FPPS can be solved in* $\mathrm{O}(n^{1+\varepsilon})$ *time.*

(2) *If* $\theta(k) \geqslant k - \log n$ *for all* k, *then FPPS can be solved in* $\mathrm{O}(n \log n)$ *time.*
(3) *If* X_n *is sorted and* $\theta(k) \geqslant k - \mathrm{O}(1)$ *for all* k, *then FPPS can be solved in* $\mathrm{O}(n)$ *time.*

4. Huffman-tree insertion prefix sums

Given a Huffman tree T over X_n and a new value x, we want to insert x into T in $\mathrm{O}(\ell)$ time so that the resulting tree T' is still a Huffman tree, where ℓ is the number of nodes whose sibling numbers are greater than the sibling number of the inserted node in T'. We will use the same data structure as in Huffman-tree deletion. In addition to the two constant-time operations Replace and WeightUpdate defined in Section 3, the following constant-time operation is also useful.

– Swap(i, j): Swap the left (respectively, right) child of i with the left (respectively, right) child of j, and swap w_i with w_j.

Denote by parent(i) the sibling number of the parent node of node i.

The idea for insertion is to start from the root and keep swapping the new leaf x down until x is in the correct position; namely, if x is at node i, then $x \geqslant w_{i-1}$. After x is inserted, we need to restore the sibling property of the new tree.

Algorithm 7. *Huffman-tree insertion*

The algorithm takes (T, x) as input, where T is a Huffman tree of n leaves, and x is a new number to be inserted into T.

```
(1) Copy $T$ into the first $2n-1$ cells of a list $A$ of $2n+1$ cells. Create two nodes,
    one being a new root with weight $x + w_{2n-1}$, and the other being a leaf node with
    weight $x$.
(2) Make the new root have $T$ as the left child and the leaf for $x$ as the right child.
    Let $A[2n+1]$ point to the new root and $A[2n]$ to the leaf for $x$.
(3) If $x \geqslant w_{2n-1}$, the algorithm ends; otherwise, set $i \leftarrow 2n$.
(4) While $i > 1$ and $x < w_{i-1}$, repeat the following: Call Swap$(i, i-1)$; set $i \leftarrow i-1$.
    (Now $x$ is in the correct position. The following steps restore the sibling property
    of the new tree.)
(5) Update the weights of nodes from node $i$'s parent to node $i$'s grandparent:
    Set $p \leftarrow$ parent$(i)$; while $i \leqslant p$, repeat the following.
    (a) If node $i$ is a left child, call WeightUpdate$(i, i, i+1)$.
    (b) If node $i$ is a right child, call WeightUpdate$(i, i, i-1)$.
    (c) Set $i \leftarrow i+1$.
(6) Find the smallest node downward from node $p$ that is out of order:
    Set $j \leftarrow p$; while $w_j < w_{j-1}$ , set $j \leftarrow j-1$
(7) Create a list $L$ to hold leaf nodes and a list $B$ to hold internal nodes; both are
    initially set to empty. Set $k \leftarrow j$. While $k < 2n-1$, repeat the following.
    (a) If node $k$ is a leaf, add $k$ to the end of $L$; otherwise, add node $k$ to the end
        of $B$.
    (b) Set $k \leftarrow k+1$.
(8) Let $b$ represent the first node in $B$ and $\ell$ be the first node in $L$. Set $j_0 = j$. While
    $j < 2n-1$. Repeat the following:
    (a) If $w_\ell < w_b$, call Replace$(j, \ell)$; otherwise, call Replace$(j, b)$.
    (b) If $j \equiv 0(\mathrm{mod}\,2)$ (i.e., node $j$ is a right child), call WeightUpdate$(j, j, j-1)$.
        (This WeightUpdate call also updates the weight of parent$(j)$ in lists.)
    (c) Set $j \leftarrow j+1$.
```

Lemma 7. *Let T be a Huffman tree of n leaves and x be a new value. Then Algorithm 7 produces a new Huffman tree by inserting x into T as a new leaf in* $O(2n - i_0)$ *time, where i_0 is the sibling number of the leaf for x in the new tree.*

Proof. Algorithm 7 first finds a correct place for the new leaf node for x in the new tree. This is done by Steps 1–4. In particular, Step 4 swaps the new leaf down until x is in the correct position. Let i_0 be the sibling number of this position. Let p be the sibling number of the parent of node i_0. After updating the weights of nodes from p

to p's parent by Steps 5, we note that p may now be out of place. Namely, the value of p may become smaller than the value of another leaf node whose sibling number is greater than i_0 but is less than the sibling number of p. This situation may happen if x is smaller than the original value of node i_0. Even if p is not out of place, it is still possible to have a node out of place on the right-hand side of p or in an upper level of p. Hence, we need to restore the sibling property.

Note that in the new tree after x is inserted, all internal nodes are in order with respect to internal nodes and all leaf nodes are in order with respect to leaf nodes. In other words, the list of all leaf nodes from bottom up is sorted in increasing order, and the list of all internal nodes from bottom up is sorted in increasing order. We also note that swapping a internal node with a leaf node does not affect the sibling property of all the nodes below. Based on these two properties we can restore the sibling property for the new tree as follows. We first find the smallest sibling number j_0 from which the sibling property needs to be restored. This is done by Step 6. We then create two sorted lists L and B, where L stores pointers to all the leaf nodes from j_0 up, and B stores pointers to all the internal nodes from j_0 up. We then start from node $j = j_0$, replace node j by the smaller node y of the first node in L and the first node in B. Extract node y from its list. Update the weight of node j's parent, increase j by 1, and continue until the root is reached.

Hence, the algorithm runs in $O(2n - j_0)$ time. Since $j_0 > i_0$, the algorithm runs in $O(2n - i_0)$ time. This completes the proof. □

Algorithm 8. *Huffman-tree-insertion prefix sums* (*HIPS*).

The algorithm takes X as input.

```
(1) Construct a binary addition tree $T_2$ over $X_2$. Set $k \leftarrow 2$.
(2) While $k \leqslant n$, repeat the following steps.
    (a) Output the value of the root of $T_k$;
    (b) Use Algorithm 7 to insert $x_{k+1}$ to $T_i$ to obtain a Huffman tree $T_{k+1}$ over $X_{k+1}$;
    (c) Increase $k$ by 1.
```

As a straightforward corollary of Lemma 7 we have the following theorem.

Theorem 8. *Let the sibling number of x_k in T_k be $s(k)$, where $1 \leqslant s(k) \leqslant 2k - 2$. Algorithm 6 solves FPPS (in the order of $k = 1$ to n) in $O(\sum_{k=1}^{n}(2k - s(n)))$ time.*

It is straightforward to see that Theorem 5 holds true without the $\Theta(f(n))$ term if Algorithm 8 is used to solve FPPS; Corollary 6 also holds true and the requirement of X_n being sorted is no longer needed in Corollary 6(3).

5. Running examples and time analysis

We implemented all four of the algorithms LDPS, LIPS, HDPS, and HIPS in C++ with the help of Paul Nelson, an undergraduate student at the University of North

Table 1

n	$T_{\mathrm{LD}}(n)$	$T_{\mathrm{LI}}(n)$	$T_{\mathrm{HD}}(n)$	$L_{\mathrm{HI}}(n)$	n	$T_{\mathrm{LD}}(n)$	$T_{\mathrm{LI}}(n)$	$T_{\mathrm{HD}}(n)$	$L_{\mathrm{HI}}(n)$
1	1.43	2.05	1.86	1.12	16	357.00	484.07	261.87	397.05
2	5.91	8.22	3.79	5.57	17	414.11	550.82	299.19	447.41
3	11.79	17.39	8.43	11.21	18	467.33	622.22	338.34	503.40
4	19.52	29.87	14.32	20.99	19	523.73	693.31	381.49	564.65
5	30.74	45.51	22.24	32.92	20	586.50	772.21	425.84	630.09
6	44.44	65.19	31.92	47.56	21	651.94	854.66	475.71	694.46
7	60.38	89.11	44.72	65.34	22	720.95	936.96	519.67	761.05
8	78.38	114.92	58.05	88.45	23	795.17	1029.24	571.68	833.29
9	98.91	146.96	73.73	112.45	24	871.98	1130.72	624.93	913.76
10	126.87	181.96	92.77	142.52	25	957.71	1243.94	683.59	999.61
11	157.62	221.94	115.15	176.56	26	1030.35	1360.28	732.93	1092.60
12	188.61	265.57	139.35	212.26	27	1114.65	1471.02	790.84	1178.49
13	225.48	313.82	166.93	253.25	28	1212.54	1589.61	856.91	1271.72
14	268.09	366.17	199.14	298.81	29	1305.28	1708.09	918.78	1365.42
15	304.67	424.48	227.68	346.03	30	1435.47	1831.64	1002.36	1454.61

Carolina at Greensboro. For more information about the implementation, the reader is referred to [10]. We compiled our programs using the GNU project C++ compiler 2.5.0 and ran numerical experiments on a SUN Ultra-10 workstation. The input numbers were generated using the standard random number generator srand() with a different seed for each run. We ran the algorithms on randomly generated X_n for n from 100 to 30,000. For each n, we ran the algorithm for five times on different, randomly generated X_n. We timed the algorithm LDPS, LIPS, HDPS, and HIPS after inputs were generated, and then took the average of the running time.

Let $T_{\mathrm{LD}}(n)$ denote the average running time of LDPS on input of n numbers. $T_{\mathrm{LI}}(n)$, $T_{\mathrm{HD}}(n)$, and $T_{\mathrm{HI}}(n)$ are defined similarly. Table 1 shows the average running time (in s) of the four algorithms for selected values of n (in thousands).

Fig. 3 shows the result of plotting all the numerical values of the algorithms from $n = 5000$ to 30,000 using MATLAB. From Fig. 3(a) and (d), we observe that the following comparison functions

$$\inf \frac{T_{\mathrm{LD}}(n)}{T_{\mathrm{HD}}(n)}, \quad \inf \frac{T_{\mathrm{LI}}(n)}{T_{\mathrm{HI}}(n)}, \quad \inf \frac{T_{\mathrm{LD}}(n) - T_{\mathrm{HD}}(n)}{T_{\mathrm{LD}}(n)}, \quad \text{and} \quad \inf \frac{T_{\mathrm{LI}}(n) - T_{\mathrm{HI}}(n)}{T_{\mathrm{LI}}(n)}$$

are increasing when n is sufficiently large. Moreover, we observe from Fig. 3(c) the following relations:

$$T_{\mathrm{LD}}(n) = T_{\mathrm{HD}}(n) + \alpha(n), \tag{3}$$

$$T_{\mathrm{LI}}(n) = T_{\mathrm{HI}}(n) + \beta(n), \tag{4}$$

where $\alpha(n)$ and $\beta(n)$ are superlinear functions. Thus, HDPS is asymptotically faster than LDPS, and HIPS is asymptotically faster than LIPS. Due to the extra cost of inserting x_{k+1} into a sorted X_k, LIPS is slower than LDPS. Similarly, HIPS is slower than HDPS.

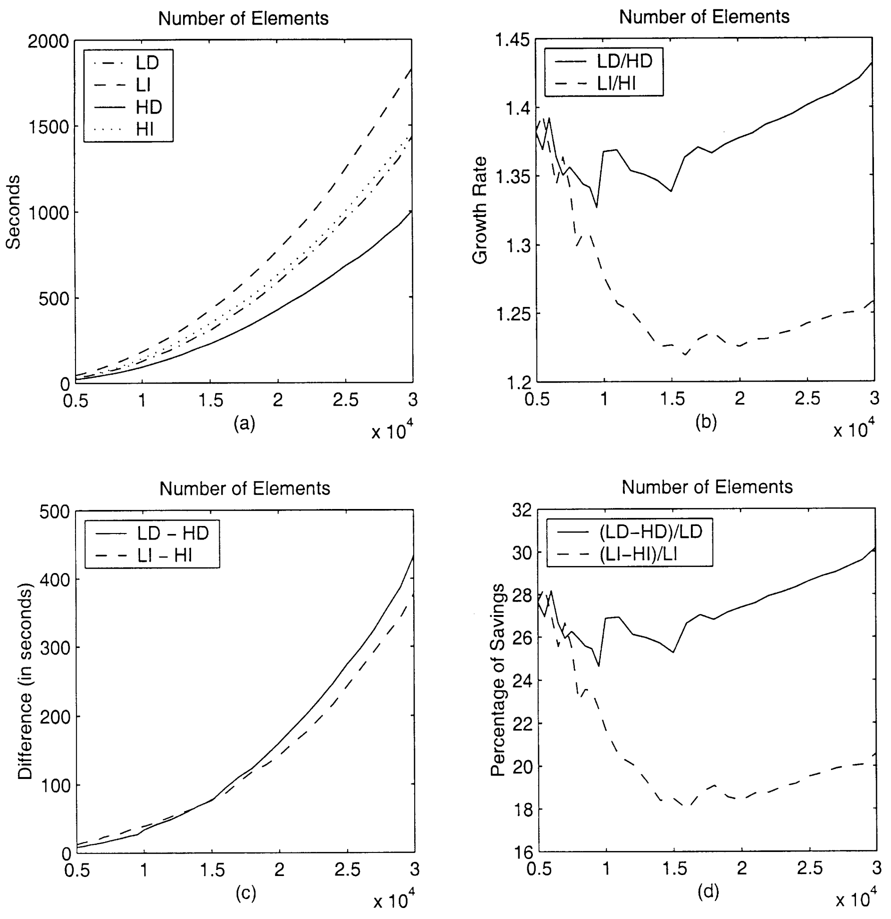

Fig. 3. (a) The plotting of the average running time of the four algorithms. (b) The plotting of two comparisons that compare the asymptotic growth rates of $T_{\mathrm{LD}}(n)/T_{\mathrm{HD}}(n)$ and $T_{\mathrm{LI}}(n)/T_{\mathrm{HI}}(n)$. (c) The plotting of the two differences: $T_{\mathrm{LD}}(n) - T_{\mathrm{HD}}(n)$ and $T_{\mathrm{LI}}(n) - T_{\mathrm{HI}}(n)$. (d) The plotting of two comparisons that show the percentage savings in average running time of HDPS over LDPS and HIPS over LIPS.

For instance, when $n = 30,000$, LDPS takes 1435.47 s on average on a SUN Ultra-10 workstation, while HDPS takes only 1002.36 s on average; hence, compared to LDPS, HDPS saves 433.11 s, resulting in a saving of more than 30% of the running time on average. Moreover, the larger the number of elements is, the more saving we will get using our dynamic algorithms.

Acknowledgements

We thank Paul Nelson for helping us to implement the algorithms studied in this paper.

References

[1] G.E. Blelloch, Scans as primitive parallel operations, IEEE Trans. Comput. 38 (11) (1989) 1526–1538.
[2] G.E. Blelloch, Prefix sums and their applications, in: J.H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, Los Altos, CA, 1993.
[3] R.P. Brent, H.T. Kung, The chip complexity of binary arithmetic, Proc. 12th Annual ACM Symp. on Theory of Computing, 1980, pp. 190–200.
[4] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[5] J.W. Demmel, Underflow and the reliability of numerical software, SIAM J. Sci. Statist. Comput. 5 (1984) 887–919.
[6] F.E. Fich, New bounds for parallel prefix circuits, Proc. 15th Ann. ACM Symp. on Theory of Computing, 1983, pp. 100–109.
[7] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comput. Surveys 23 (1990) 5–48.
[8] N.J. Higham, The accuracy of floating point summation, SIAM J. Sci. Comput. 14 (1993) 783–799.
[9] K. Iverson, A Programming Language, Wiley, New York, 1962.
[10] M.-Y. Kao, P. Nelson, J. Wang, Implementation of Huffman-tree deletion and insertion for minimizing prefix set summation errors, Tech. Report TR 99-10, Department of Mathematical Sciences, The University of North Carolina at Greensboro, 1999.
[11] M.-Y. Kao, J. Wang, Efficient minimization of numerical summation errors, Proc. 25th Internat. Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 1143, Springer, Berlin, 1998, pp. 375–386.
[12] M.-Y. Kao, J. Wang, Linear-time approximation algorithms for computing numerical summation with provably small errors, SIAM J. Comput. 29 (2000) 1568–1576.
[13] D.E. Knuth, The Art of Computer Programming II: Seminumerical Algorithms, 2nd ed., Addison-Wesley, Reading, MA, 1981.
[14] U.W. Kulisch, W.L. Miranker, The arithmetic of the digital computer: a new approach, SIAM Rev. 28 (1986) 1–40.
[15] R.E. Ladner, M.J. Fischer, Parallel prefix computation, J. Assoc. Comput. Mach. 27 (4) (1980) 831–838.
[16] J. van Leeuwen, On the construction of Huffman trees, Proc. 3rd Internat. Colloquium on Automata, Languages, and Programming, 1976, pp. 382–410.
[17] Y. Ofman, On the algorithmic complexity of discrete functions, Sov. Phys.—Dokl. 7(7) (1963) 589–591. English translation.
[18] T.G. Robertazzi, S.C. Schwartz, Best “ordering” for floating-point addition, ACM Trans. Math. Software 14 (1988) 101–110.
[19] A. Schönhage, Storage modification machines, SIAM J. Comput. 9 (1980) 490–508.
[20] H. Stone, Parallel processing with the perfect shuffle, IEEE Trans. Comput. C-20 (1971) 153–161.
[21] V.J. Torczon, On the convergence of the multidirectional search algorithm, SIAM J. Optim. 1 (1) (1991) 123–145.