March 7, 1990 LIDS-P-1943

ON COMPUTING THE NESTED SUMS AND INFIMAL CONVOLUTIONS OF CONVEX PIECEWISE–LINEAR FUNCTIONS*

by

Paul Tseng† and Zhi-Quan Luo‡

ABSTRACT

We consider the problem of evaluating a functional expression comprising the nested sums and infimal convolutions of convex piecewise–linear functions defined on the reals. For the special case where the nesting is serial, we give an $O(N \log N)$ time algorithm, where N is the total number of breakpoints of the functions. We also prove a lower bound of $\Omega(N \log N)$ on the number of comparisons needed to solve this problem, thus showing that our algorithm is essentially optimal. For the general case, we give an $O(N \log^2 N)$ time algorithm. We apply this latter algorithm to the linear cost network flow problem on series–parallel networks to obtain an $O(m \log^2 m)$ time algorithm for this problem, where m is the number of arcs in the network. This result improves upon the previous algorithm of Bein, Brucker, and Tamir which has a time complexity of $O(nm+m \log m)$, where n is the number of nodes.

KEY WORDS: Convex program, balanced binary search tree, infimal convolution, series–parallel graphs, linear cost network flow.

---

* This research is partially supported by the U.S. Army Research Office, contract DAAL03-86-K-0171 (Center for Intelligent Control Systems), and by a grant from the Science and Engineering Research Board of McMaster University.

†Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139.

‡Room 225/CRL, Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, L8S 4L7, Canada.

1 Introduction

A classical problem in logic is that of the evaluation of Boolean expressions comprising the nested AND and OR of Boolean variables. A problem entirely analogous to this, except being defined on the space of *convex piecewise–linear functions* instead of {true, false}, is that of the evaluation of *functional expressions* comprising the nested "pointwise sum" and "infimal convolution" of convex piecewise–linear functions on the reals. Just as AND and OR are dual to each other through negation, the operations of pointwise sum and infimal convolution are dual to each other through *conjugation* (see Section 2 for the definition). This functional expression evaluation problem, apart from possessing the interesting analogy to the evaluation of Boolean expressions, also has a number of practical applications, particularly to problems defined on *series–parallel* networks [Duf65]. As an example, consider the following system identification problem that is fundamental in the study of *monotone* networks [Min60], [Dol79]: determine the characteristic function of a monotone network, given its topology and the characteristic function of its elements. [Roughly speaking, a monotone network is a mathematical model of an interconnection of physically lumped elements, whose variables obey Kirchoff's laws and satisfy certain monotone relations associated with the elements.] It can be shown that in the special case where the characteristic function of the elements are step functions and the network is series–parallel, this problem is reducible to our functional expression evaluation problem. Another example, closely related to the previous example, is the linear cost network flow problem on series–parallel networks (see Subsection 6.1).

In this paper, we propose a number of highly efficient algorithms for solving the above functional expression evaluation problem. In particular, we give an $O(N \log N)$ time algorithm, where N is the total number of breakpoints of the component functions, for solving the special case where the nesting is serial. We also prove a lower time bound of $\Omega(N \log N)$, thus demonstrating that our algorithm is optimal to within a constant multiplicative factor. For the general case of arbitrary nesting, we give an almost optimal algorithm with a time complexity of $O(N \log^2 N)$. We then apply this latter algorithm to the linear cost network flow problem on series–parallel networks to obtain an $O(m \log^2 m)$ time algorithm for this problem, where m is the number of arcs in the network. This result significantly improves upon the $O(nm + m \log m)$ time bound obtained by Bein, Brucker and Tamir [BBT85], where n denotes the number of nodes in the network.

The key to our results lies in the data structure that we use for storing each convex piecewise–linear function. In particular, we reduce the problem of computing the sum (respectively, infimal convolution) of two convex piecewise–linear functions to that of sequentially inserting the breakpoints (respectively, slopes) of one function amongst those of the other function. To perform the insertions efficiently, we store the breakpoints and the slopes of the functions in sorted order using a *balanced binary search tree* (see, e.g., [Bay72], [Tar83]). Moreover, rather than storing the actual value of the breakpoints and the slopes, we store the *difference* between neighboring (with respect to the binary search tree) breakpoints and slopes.

2 Problem Description

We say that a function $f : \Re \mapsto (-\infty, +\infty]$ is *convex piecewise-linear* (c. p. l. for short) if it has the following form

$$f(x) = \begin{cases} b_j x + d_j & \text{if } a_j \leq x < a_{j+1} \text{ for some } j \in \{1, ..., m-1\}; \\ +\infty & \text{otherwise,} \end{cases} \tag{2.1}$$

where $a_1, ..., a_m$, $b_1, ..., b_{m-1}$ ($m \geq 2$) are scalar constants satisfying $-\infty \leq a_1 < ... < a_m \leq +\infty$, $-\infty < b_1 < ... < b_{m-1} < +\infty$, and $d_1, ..., d_{m-1}$ are scalar constants satisfying $b_{j-1}a_j + d_{j-1} = b_j a_j + d_j$ for $j = 2, ..., m-1$. The closed interval $[a_1, a_m]$ is the *effective domain* of f (i.e., the set of points at which f is finite), the finite–valued a_j's are the *breakpoints* of f, and the finite–valued b_j's are the *slopes* of f. [We allow the possibility $-\infty < a_1 = a_m < +\infty$, in which case a_1 is, by convention, the unique breakpoint of f.] For any $x \in [a_j, a_{j+1})$, $j \in \{1, ..., m-1\}$, we say that b_j is the *right slope* of f at x. The function f is closed proper and, since its right slope is monotonically increasing within its effective domain, also convex. [For convenience, if $x < a_1$ (respectively, $x \geq a_m$), then we define the right slope of f at x to be $-\infty$ (respectively, $+\infty$).] The *left slope* of f is defined analogously. A fundamental property of c. p. l. functions is that each of them is determined, up to an additive constant, by its breakpoints and slopes (see [Roc84, Sec. 8B]). This, as we shall see, allows us to work with the breakpoints and slopes only.

We say that a c. p. l. function $h : \Re \mapsto (-\infty, +\infty]$ is *simple* if h has exactly one breakpoint, i.e.,

$$h(x) = \begin{cases} b(x-a) + d & \text{if } x < a; \\ c(x-a) + d & \text{otherwise,} \end{cases} \tag{2.2}$$

where a, b, c and d are some scalar constants satisfying $-\infty < a < \infty$, $-\infty < d < \infty$, $-\infty \leq b < c \leq \infty$. [We use the convention $(+\infty)0 = 0$.] Notice that the effective domain of h is either $\Re$ or $[a, +\infty)$ or $(-\infty, a]$ or $\{a\}$, depending on whether $b = -\infty$ and whether $c = +\infty$.

An important property of simple c. p. l. functions is that any c. p. l. function f with, say, n breakpoints ($n \geq 1$) can be decomposed into the sum of n simple c. p. l. functions. To see this, simply note that if $e_1 < ... < e_n$ are the breakpoints of f, s_j is the right slope of f at e_j for all j, and s_0 is the left slope of f at e_1, then

$$f = h_1 + ... + h_n, \tag{2.3}$$

where $h_1, h_2, ..., h_n$ are simple c. p. l. functions given by, respectively,

$$h_1(x) = \begin{cases} s_0(x - e_1) + d & \text{if } x < e_1; \\ s_1(x - e_1) + d & \text{otherwise,} \end{cases} \tag{2.4}$$

for some scalar d, and

$$h_j(x) = \begin{cases} 0 & \text{if } x < e_j; \\ (s_j - s_{j-1})(x - e_j) & \text{otherwise,} \end{cases} \quad j = 2, ..., n. \tag{2.5}$$

For any closed proper convex function $f : \Re \mapsto (-\infty, +\infty]$, we denote by f^* the *conjugate* function (or the *Legendre transformation*) of f, i.e.,

$$f^*(x) = \sup_{y \in \Re}\{xy - f(y)\}, \qquad \forall x \in \Re.$$

It is well–known that f^* is also closed proper convex, maps $\Re$ into $(-\infty, +\infty]$, and satisfies

$$f = (f^*)^*. \tag{2.6}$$

Moreover, the following hold (see [Roc84, Sec. 8E]):

Observation 1.1. A closed proper convex function $f : \Re \mapsto (-\infty, +\infty]$ is c. p. l. if and only if f^* is c. p. l. If in addition the effective domain of f is bounded and $e_1 < e_2 < ... < e_m$ are the breakpoints of f and s_j is the right slope of f at e_j, $j = 1, ..., m-1$, then the effective domain of f^* is all of $\Re$, the breakpoints of f^* are $s_1 < ... < s_{m-1}$, and e_{j+1} is the right slope of f^* at s_j (the left slope of f^* at s_1 is e_1). [If the effective domain of f comprises a single point, say, a, then f^* is simply a linear function with slope a.] The converse of this also holds.

Consider any two c. p. l. functions f and g. We denote by $f + g$ the pointwise sum (or simply "sum") of f and g, i.e.,

$$(f + g)(x) = f(x) + g(x), \qquad \forall x \in \Re, \tag{2.7}$$

and by $f \square g$ the *infimal convolution* of f and g (see [Roc70, p. 34]), i.e.,

$$(f \square g)(x) = \inf_{y \in \Re}\{f(y) + g(x - y)\}, \qquad \forall x \in \Re. \tag{2.8}$$

It is well–known that the operators $+$ and $\square$ are dual to each other through conjugation (see [Roc70, p. 145]), i.e.,

$$f \square g = (f^* + g^*)^*. \tag{2.9}$$

Notice that if the effective domain of f and of g are both unbounded, then $f \square g$ may take the extended value $-\infty$ everywhere. To avoid dealing with such a function, we will assume that both f and g have bounded effective domains. [This assumption is not restrictive since one can always impose an arbitrarily large artificial bound without changing the problem characteristics.] Then, it can be seen from the respective definitions that $f \square g$ is also a c. p. l. function with bounded effective domain, and the same holds for $f + g$, except when the respective effective domain of f and of g do not intersect, in which case $f + g$ is improper (i.e., has the extended value $+\infty$ everywhere). Moreover, by Observation 1.1, f^* is a real–valued c. p. l. function with $n - 1$ breakpoints, where n denotes the number of breakpoints of f, so f^* can be expressed as the sum of $n - 1$ real–valued simple c. p. l. functions $h_1, ...h_{n-1}$ [cf. (2.3)–(2.5)]. Then, if $n \geq 2$, it follows from (2.9) that

$$f \square g = (f^* + g^*)^* = (h_1 + ... + h_{n-1} + g^*)^* = (h_1)^* \square ... \square (h_{n-1})^* \square g, \tag{2.10}$$

so that $f \square g$ can be obtained by successively convolving the $(h_i)^*$'s onto g. If $n = 1$, then f is finite at exactly one point, say a, and it is easily seen from the definition (2.8) that $f \square g$ is simply a shift of g to the right by a, i.e.,

$$(f \square g)(x) = g(x - a), \ \forall x \in \Re. \tag{2.11}$$

Hence, the problem of computing the sum (respectively, infimal convolution) of two c. p. l. functions having bounded effective domain can be reduced to a sequence of simpler problems, each involving the summation of a simple c. p. l. function (respectively, the infimal convolution of the conjugate of a real–valued simple c. p. l. function) with a c. p. l. function having bounded effective domain. We study these simpler problems [and the shifting operation (2.11)] in more depth below.

For any c. p. l. function g with bounded effective domain, we say that a set of scalars $e_1 < ... < e_m$ (taking values in $(-\infty, +\infty)$) is a set of *extended* breakpoints for g if those e_j's in the effective domain of g are precisely the breakpoints of g. Notice that then, for any scalar a,

$$e_1 + a, ..., e_m + a \tag{2.12}$$

is a set of extended breakpoints for the function obtained by shifting g to the right by a [cf. (2.11)]. An important fact that we use is that, for any simple c. p. l. function h, we can obtain a set of extended breakpoints for $h + g$ (respectively, $h^* \square g$) essentially by *inserting* the unique breakpoint of h amongst a set of extended breakpoints for g (respectively, the slopes of g). To make this notion precise, let $e_1 < ... < e_m$ be a set of extended breakpoints for g, let s_j be the right slope of g at e_j. Also, let a, b and c be, respectively, the breakpoint of h, the right slope of h at a, and the left slope of h at b [cf. (2.2)].

Case 1 $h + g$. If $e_j = a$ for some j, then it is easily seen that

$$e_1 < ... < e_m \tag{2.13}$$

is a set of extended breakpoints for $h + g$ and the corresponding right slopes are

$$s_1 + b, ..., s_{j-1} + b, s_j + c, ..., s_m + c. \tag{2.14}$$

Otherwise, there exists an index $j \in \{1, ..., m + 1\}$ for which $e_{j-1} < a < e_j$ (with the convention $e_0 = -\infty, e_{m+1} = \infty$), in which case

$$e_1 < ... < e_{j-1} < a < e_j < ... < e_m \tag{2.15}$$

is a set of extended breakpoints for $h + g$ and the corresponding right slopes are

$$s_1 + b, ..., s_{j-1} + b, s_{j-1} + c, s_j + c, ..., s_m + c \tag{2.16}$$

(with the convention $s_0 = -\infty$). If either $s_{j-1} = +\infty, b = -\infty$ or $s_j = -\infty, c = +\infty$, then $h + g$ is improper.

Case 2 $h^* \square g$. We further assume that h is real-valued (i.e., b and c are finite). It can be seen by using the duality relation (2.9) and (2.15)–(2.16) that if $s_j = a$ for some j, then

$$e_1 + b < ... < e_j + b < e_{j+1} + c < ... < e_m + c \tag{2.17}$$

is a set of extended breakpoints for $h^* \square g$ and the corresponding right slopes are

$$s_1, s_2, ..., s_m. \tag{2.18}$$

Otherwise, there exists an index $j \in \{1, ..., m+1\}$ for which $s_{j-1} < a < s_j$ (we use the convention $s_0 = -\infty$ and $s_{m+1} = \infty$),

$$e_1 + b, ..., e_{j-1} + b, e_j + b, e_j + c, ..., e_m + c \tag{2.19}$$

is a set of extended breakpoints for $h^* \square g$ and the corresponding right slopes are

$$s_1, ..., s_{j-1}, a, s_j, ..., s_m. \tag{2.20}$$

Eqs. (2.13)–(2.20) immediately suggest the following insertion procedure for computing a set of extended breakpoints, together with the respective right slopes, for $h + g$ (or $h^* \square g$):

Naive Insertion Procedure: We maintain the extended breakpoints $e_1 < ... < e_m$ for g and the corresponding right slopes $s_1 < ... < s_m$ in a sorted linked list $\mathcal{L}$. Then, we scan $\mathcal{L}$ in the order $(e_1, s_1), (e_2, s_2), .., (e_m, s_m)$. In the case of $h + g$, we add b to s_j for every (e_j, s_j) encountered until we find an index j for which either $e_j = a$ or $e_{j-1} < a < e_j$, in the latter case of which we insert $(a, s_{j-1} + c)$ immediately following $(e_{j-1}, s_{j-1} + b)$. From then on, we add c to s_j for every (e_j, s_j) encountered. The case of $h^* \square g$ is treated symmetrically: We add b to e_j until we find an index j for which either $s_j = a$ or $s_{j-1} < a < s_j$, in the latter case of which we insert $(e_j + b, a)$ immediately following $(e_{j-1} + b, s_{j-1})$. From then on, we add c to e_j for every (e_j, s_j) encountered.

The time for the above insertion procedure is clearly $O(m)$. We will demonstrate in the next two sections that, by using a more sophisticated data structure, we can reduce the time for insertion to $O(\log m)$. [In this paper, "log" denotes the logarithm of base 2.]

3 Data Structure

In this section, we describe the data structure that we use to store each c. p. l. function g. More precisely, we use a data structure called the *balanced binary search tree* due to Bayer [Bay72] (also see [Tar83]) to store a set of extended breakpoints of g and their respective right slopes. In addition, we do not always store the actual value of an extended breakpoint (respectively, its right slope), but sometimes the *difference* between it and the extended breakpoint (respectively, their right slopes) represented by its "parent" in the binary search tree. We will show in the next section that, by storing g in this manner, we can compute the sum or the infimal convolution of g with any simple c. p. l. function in $O(\log m)$ time, where m is the number of extended breakpoints representing g.

3.1 Balanced Binary Search Tree

We first review the notion of a full binary tree (see [AHU83], [Tar83]). A *full binary tree* $\mathcal{T}$ is a finite rooted tree where each node has either two direct descendants called *children* or no children. [For convenience, we will number the nodes in $\mathcal{T}$ from 1 to n, where n is the number of nodes in $\mathcal{T}$.] A node with two children is *internal* and a node with no children is *external*. A node and all its descendants form a *subtree of* $\mathcal{T}$ and the node is called the *root* of that subtree. If i is an internal node, then one of its children is distinguished as a *left child*, denoted by $left(i)$, and the other child is a *right child*, denoted by $right(i)$. The subtree rooted at $left(i)$ (respectively, $right(i)$) is the *left subtree* (respectively, the *right subtree*) of i. If a node i is not the root of $\mathcal{T}$, then the unique immediate predecessor of i, denoted by $p(i)$, is the *parent* of i. [We use the convention $p(i) = i$ if i is the root of $\mathcal{T}$.] We denote by $p^k(i)$ ($k \geq 0$) the node obtained by applying $p(\cdot)$ to i a total of k times (with the convention $p^0(i) = i$).

Suppose that we are given a finite set of *distinct* values from a totally ordered universe. We can represent such a set by a full binary tree containing one value per internal node, with values arranged in *symmetric order*: if i is an internal node, then the value stored in i is greater than every value stored in the left subtree of i, and less than every value stored in the right subtree of i. Such a data structure is called a *binary search tree* (or BS tree for short) (see [Tar83, Sec. 4]).

An important feature of BS tree is that a new value, say v, can be inserted into it by simply moving down the path from the root of the tree to an external node. This is done as follows: We start at the root of the tree; whenever we are at an internal node, say j, we compare v with the value stored in j; if v is equal to this value, then we stop (v is already stored in the tree), otherwise we move to the left child (respectively, right child) of j if v is less (respectively, greater) than this value. When we reach an external node j, we store v in j and create two children for j.

Another important feature of BS tree is that the values stored in it can be extracted in sorted

order in $O(n)$ time, where n is the number of nodes in the tree. [This is done by using the following recursive procedure: Let r denote the root of the tree. First extract (in sorted order) the values stored in the left subtree of r, then the value store in r, and finally the values stored in the right subtree of r.]

A drawback of BS tree $\mathcal{T}$ is that its height (i.e., the maximum length of any path from the root of $\mathcal{T}$ to an external node) can be as large as n, the number of nodes in $\mathcal{T}$. Since the time required to insert one piece of data is proportional to the height of the tree, we see that insertion has a worst case time of $O(n)$. To improve on this bound, we will use a balanced version of the BS tree, called *balanced* BS tree, as invented by Bayer [Bay72] (also see [Tar83, Sec. 4.2]). This is a BS tree in which each node i is colored either *red* or *black* such that the following hold:

1. Every red node has a black parent.
2. Every external node is black and all paths from a given node to an external node contains the same number of black nodes.

To represent a balanced BS tree, we store with each internal node a bit to indicate its color. It can be seen (cf. [Tar83, p. 50]) that the height of a balanced binary search tree with n nodes is at most $2\lfloor \log(n+1) \rfloor$, so that the operation of *inserting* a value into a balanced BS tree takes at most $O(\log n)$ time.

After a value is inserted, the BS tree may no longer be balanced, in which case a sequence of *color flipping* and *rotation* operations involving the predecessors of the node containing the inserted value, called the *rebalancing phase*, is performed to rebalance the tree. We will not go into the details of these operations here (see, e.g., [Tar83, Sec. 4.2]). It suffices for our purpose to make the following observations about them:

Observation 3.1. The color flipping operations do not change the structure of the tree.

Observation 3.2. The rotation operations restructure the tree only by interchanging the children of those nodes that are predecessors of the node containing the inserted value.

Observation 3.3. The color flipping and the rotation operations, which comprise the rebalancing phase, require only $O(\log n)$ time in total, where n is the number of nodes in the tree.

The above observations will be used in Subsections 4.1 and 4.2 to obtain an $O(\log m)$ algorithm for computing, respectively, $f + g$ and $f^* \square g$, for any simple c. p. l. function f and any c. p. l. function g with bounded effective domain, where m is the number of extended breakpoints representing g.

3.2 Partial Sum Representation of Extended Breakpoints and Slopes

Let g be a c. p. l. function with bounded effective domain. Let $e_1 < \ldots < e_m$ be a set of extended breakpoints for g and let s_j be the right slope of g at e_j for all j. Of course, there holds $s_1 < \ldots < s_m$ [cf. convexity of g], so that $(e_1, s_1) \prec \ldots \prec (e_m, s_m)$, where "$\prec$" is the usual coordinate partial ordering.

We assume that the (e_j, s_j)'s are stored using a balanced BS tree $\mathcal{T}$. For convenience, we give the label j to the node containing (e_j, s_j) and call (e_j, s_j) the *value* of node j. Then, these values are stored in the symmetric order, i.e.,

if k is a node in the right (left) subtree of a node j, then $(e_j, s_j) \prec (e_k, s_k)$ $((e_k, s_k) \prec (e_j, s_j))$.

With respect to $\mathcal{T}$, we say that a set of 4–tuples $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$, $j = 1, \ldots, m$, is a *partial sum representation* of the (e_j, s_j)'s (in short, of g) if these 4–tuples satisfy

$$e_j = \alpha_j + \sum_{\substack{l = p^k(j) \\ \text{for some } k \geq 0}} \Delta\alpha_l, \quad j = 1, \ldots, m, \tag{3.1}$$

and

$$s_j = \beta_j + \sum_{\substack{l = p^k(j) \\ \text{for some } k \geq 0}} \Delta\beta_l, \quad j = 1, \ldots, m. \tag{3.2}$$

Roughly speaking, each e_j can be obtained by summing $\Delta\alpha_l$ over all predecessors l of j and then adding the sum to α_j. Analogous interpretation holds for the $\Delta\beta_l$'s and the β_l's. We will call m the *size* of the partial sum representation.

A simple example of $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s that satisfy (3.1)–(3.2) is

$$\alpha_j = e_j, \qquad \beta_j = s_j, \qquad \Delta\alpha_j = \Delta\beta_j = 0, \qquad j = 1, \ldots, m.$$

Another simple example is

$$\alpha_j = \beta_j = 0, \qquad \Delta\alpha_j = e_j - e_{p(j)}, \qquad \Delta\beta_j = s_j - s_{p(j)},$$

for all $j \in \{1, \ldots, m\}$ with $j \neq r$, and

$$\alpha_r = e_r, \qquad \beta_r = s_r, \qquad \Delta\alpha_r = \Delta\beta_r = 0,$$

where r denotes the root of $\mathcal{T}$.

4 Computing the Sum or the Infimal Convolution of a Simple Convex Piecewise–Linear Function with a Convex Piecewise–Linear Function

We have the following key result:

Proposition 4.1 *Let g be any c. p. l. function with bounded effective domain and suppose that we are given a size m partial sum representation of g stored in a balanced BS tree $\mathcal{T}$. Then, the following hold:*

(a) For any c. p. l. function f that is finite at exactly one point, we can compute in $O(1)$ time a size m partial sum representation of $f \Box g$ stored in $\mathcal{T}$.

(b) For any simple c. p. l. function h, we can, in $O(\log m)$ time, determine whether $h+g$ is proper and, if yes, compute a size m or size $m+1$ partial sum representation of $h+g$ stored in some balanced BS tree.

(c) For any real-valued simple c. p. l. function h, we can compute in $O(\log m)$ time a size m or size $m+1$ partial sum representation of $h^* \Box g$ stored in some balanced BS tree.

The proof of Proposition 4.1 begins below and it continues in the following two subsections.

Let $e_1 < ... < e_m$ denote the set of extended breakpoints for g and let s_j denote the right slope of g at e_j, for all j. Let $p(j)$ denote the parent of node j in the balanced BS tree $\mathcal{T}$ and let $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$, $j = 1, ..., m$, denote the partial sum representation of the (e_j, s_j)'s with respect to $\mathcal{T}$ [cf. (3.1), (3.2)].

The proof of part (a) is very easy. We simply add the point at which f is finite, say a, to $\Delta\alpha_r$, where r denotes the root of $\mathcal{T}$. By (3.1), the resulting $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s together with $\mathcal{T}$ form a partial sum representation of $(e_1 + a, s_1), ..., (e_m + a, s_m)$ or, equivalently [cf. (2.12)], of $f \Box g$.

Now we prove parts (b) and (c). Let a denote the unique breakpoint of h and let b, c denote, respectively, the left and right slope of h at a [cf. (2.2)].

4.1 Computing $h + g$ in $O(\log m)$ Time

We now describe how to compute $h+g$ using the formulas (2.13)–(2.16) and the data structure described above. Roughly speaking, we insert the breakpoint a into the sorted list $e_1, ..., e_m$ as described in Subsection 3.1. As we move downward from the root of $\mathcal{T}$ to an external node, we set α_j (respectively, β_j) to e_j (respectively, $s_j + b$ or $s_j + c$, depending on whether $e_j < a$ or $e_j \geq a$)

for each node j visited and, if j has two children, we update $\Delta\alpha_l$ and $\Delta\beta_l$ for the children l of j that we do not visit next. After insertion, we rebalance the BS tree as described in Subsection 3.1.

Inserting Phase

We insert the value a into $\mathcal{T}$ by moving down the path from the root of $\mathcal{T}$ to an external node. As we move downward, we accumulate the sum of $\Delta\alpha_l$ and the sum of $\Delta\beta_l$ over all nodes l encountered, so that when visiting some node j we have already accumulated the sums

$$\Delta A_j = \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\alpha_l, \tag{4.1}$$

and

$$\Delta B_j = \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\beta_l. \tag{4.2}$$

Thus, upon arriving at node j, we can compute e_j by adding α_j to ΔA_j [see Eq. (3.1)]. We then compare the value of a with e_j and decide which child of j to visit next. More precisely, we perform the following operations at node j:

1. If $a > e_j$, then we move to *right*(j) and update

$$(\Delta\alpha_j^{new}, \alpha_j^{new}, \Delta\beta_j^{new}, \beta_j^{new}) = (0, \Delta A_j + \alpha_j, 0, \Delta B_j + \beta_j + b), \tag{4.3}$$

$$(\Delta\alpha_k^{new}, \alpha_k^{new}, \Delta\beta_k^{new}, \beta_k^{new}) = (\Delta A_j + \Delta\alpha_k, \alpha_k, \Delta B_j + \Delta\beta_k + b, \beta_k), \tag{4.4}$$

   where $k = left(j)$.

2. If $a < e_j$, then we move to *left*(j) and update

$$(\Delta\alpha_j^{new}, \alpha_j^{new}, \Delta\beta_j^{new}, \beta_j^{new}) = (0, \Delta A_j + \alpha_j, 0, \Delta B_j + \beta_j + c), \tag{4.5}$$

$$(\Delta\alpha_l^{new}, \alpha_l^{new}, \Delta\beta_l^{new}, \beta_l^{new}) = (\Delta A_j + \Delta\alpha_l, \alpha_l, \Delta B_j + \Delta\beta_l + c, \beta_l), \tag{4.6}$$

   where $l = right(j)$.

3. If $a = e_j$, then we stop moving and we update $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$, $(\Delta\alpha_k, \alpha_k, \Delta\beta_k, \beta_k)$, and $(\Delta\alpha_l, \alpha_l, \Delta\beta_l, \beta_l)$ according to, respectively, (4.5), (4.4) and (4.6), where $k = left(j)$ and $l = right(j)$.

We continue moving down $\mathcal{T}$ until either we stop at a node j with $e_j = a$ or we reach an external node. In the latter case, we let the external node, denoted by i, represent the extended breakpoint a of $h + g$ and store in i the 4-tuple

$$(0, a, 0, s_j + c), \tag{4.7}$$

where $\bar{j}$ is the last predecessor j of i for which there holds $e_j < a$. [$e_{\bar{j}}$ can be seen to be the extended breakpoint of g to the immediate left of a.] Also, if $b = -\infty$, $\Delta B_{\bar{j}} + \beta_{\bar{j}} = +\infty$ or if $c = +\infty$, $\Delta B_{\tilde{j}} + \beta_{\tilde{j}} = -\infty$, where $\tilde{j}$ is the last predecessor j of i for which there holds $e_j > a$, then $h + g$ is improper. [By (3.2) and (4.2), $B_{\bar{j}} + \beta_{\bar{j}}$ and $B_{\tilde{j}} + \beta_{\tilde{j}}$ are equal to, respectively, $s_{\bar{j}}$ and $s_{\tilde{j}}$. $e_{\tilde{j}}$ can be seen to be the extended breakpoint of g to the immediate right of a.]

Rebalancing Phase

We rebalance the new BS tree as described in Subsection 3.1. No change is made to the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s.

Correctness Proof

Since the (e_j, s_j)'s are correctly computed during the inserting phase, then the value a must be inserted in the correct location. It only remains to verify that the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s form a partial sum representation of $h + g$ after the inserting phase and after the rebalancing phase.

First consider the inserting phase. We claim that, with respect to the new BS tree, the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$, $j = 1, ..., m$, [given by (4.3)–(4.6)] together with $(0, a, 0, s_{\bar{j}} + c)$ [cf. (4.7)] form a partial sum representation for $h + g$ or, more precisely, a partial sum representation for the pairs of extended breakpoint and right slope given by either (2.13)–(2.14) or (2.15)–(2.16). This is very straightforward to show using Eqs. (3.1)–(3.2) and (4.1)–(4.6). For example, if j is a node visited for which $e_j < a$, then we have from (4.3), (4.1), (3.1) that

$$\begin{aligned} \alpha_j^{new} + \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\alpha_l^{new} &= \Delta A_j + \alpha_j \\ &= \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\alpha_l + \alpha_j \\ &= e_j, \end{aligned}$$

and from (4.3), (4.2), (3.2) that

$$\begin{aligned} \beta_j^{new} + \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\beta_l^{new} &= \Delta B_j + \beta_j + b \\ &= \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\beta_l + \beta_j + b \\ &= s_j + b. \end{aligned}$$

The other j's can be checked in an analogous manner.

Now consider the rebalancing phase. Recall that the rebalancing phase comprises a sequence of color flipping and rotation operations. Clearly the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s remain a partial

sum representation of $h + g$ during the color flipping operations (since, according to Observation 3.1, there is no structural change in the tree during these operations). By Observation 3.2, the rotation operations restructure the tree only by interchanging the children of those nodes which are predecessors of the newly inserted node. Since $\Delta\alpha_j^{new} = \Delta\beta_j^{new} = 0$ for all predecessors j of i [cf. (4.3), (4.5)], it then follows that the right hand side quantities in, respectively, (3.1) and (3.2) (with respect to the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s) remain unchanged during the rotation operations, so that the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s remain a partial sum representation of $h + g$ during these operations.

Time Complexity

Clearly, the updating of each $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$ during the inserting phase requires only $O(1)$ time [see the updating rules (4.3)–(4.6)] and this has to be done only for those nodes l that are either predecessors of the external node i or children of such predecessors. Hence, the total time for the inserting phase is proportional to the number of predecessors of i, which is at most $O(\log m)$. Since the rebalancing phase clearly requires $O(\log m)$ time also (see Observation 3.3), it follows that the total time for computing the new balanced BS tree and a corresponding partial sum representation of $h + g$ is $O(\log m) \times O(1) = O(\log m)$. The size of this partial sum representation is clearly either m or $m + 1$.

4.2 Computing $h^* \square g$ in $O(\log m)$ Time

Suppose that h is real-valued (i.e., both b and c are finite). We show below how to compute $h^* \square g$ using the formulas (2.17)–(2.20) and the data structure described immediately prior to Subsection 4.1. The procedure is entirely analogous to that for computing $h + g$ and hence we describe it only briefly.

Inserting Phase

We insert the value a into $\mathcal{T}$ by moving down the path from the root of $\mathcal{T}$ to an external node. As we move downward, we accumulate the sum of $\Delta\alpha_l$ and the sum of $\Delta\beta_l$ over all nodes l visited, so that when we visit some node j we have already accumulated the sums [cf. (4.1), (4.2)]

$$\Delta A_j = \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\alpha_l,$$

and

$$\Delta B_j = \sum_{\substack{l=p^k(j) \\ \text{for some } k\geq 0}} \Delta\beta_l.$$

Thus, at node j, we can compute s_j by adding β_j to ΔB_j (see Eq. (3.2)) and then perform the following operations:

1. If $a > s_j$, then we move to *right*(j) and update

$$(\Delta\alpha_j^{new}, \alpha_j^{new}, \Delta\beta_j^{new}, \beta_j^{new}) = (0, \Delta A_j + \alpha_j + b, 0, \Delta B_j + \beta_j), \tag{4.8}$$

$$(\Delta\alpha_k^{new}, \alpha_k^{new}, \Delta\beta_k^{new}, \beta_k^{new}) = (\Delta A_j + \Delta\alpha_k + b, \alpha_k, \Delta B_j + \Delta\beta_k, \beta_k), \tag{4.9}$$

   where $k = left(j)$.

2. If $a < s_j$, then we move to *left*(j) and update

$$(\Delta\alpha_j^{new}, \alpha_j^{new}, \Delta\beta_j^{new}, \beta_j^{new}) = (0, \Delta A_j + \alpha_j + c, 0, \Delta B_j + \beta_j), \tag{4.10}$$

$$(\Delta\alpha_l^{new}, \alpha_l^{new}, \Delta\beta_l^{new}, \beta_l^{new}) = (\Delta A_j + \Delta\alpha_l + c, \alpha_l, \Delta B_j + \Delta\beta_l, \beta_l), \tag{4.11}$$

   where $l = right(j)$.

3. If $a = s_j$, then we stop moving and we update $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$, $(\Delta\alpha_k, \alpha_k, \Delta\beta_k, \beta_k)$, and $(\Delta\alpha_l, \alpha_l, \Delta\beta_l, \beta_l)$ according to, respectively, (4.8), (4.9) and (4.11), where $k = left(j)$ and $l = right(j)$.

We continue moving down the tree until either we stop at a node j with $s_j = a$ or we reach an external node. In the latter case, we let the external node, denoted by i, represent the new breakpoint of $h^* \square g$ and store in it the 4–tuple $(0, e_{\tilde{j}} + b, 0, a)$, where $\tilde{j}$ is the last predecessor j of i for which there holds $s_j > a$. [$s_{\tilde{j}}$ can be seen to be the smallest slope of g that exceeds a and it can be easily obtained when traversing down the BS tree.]

Rebalancing Phase

We rebalance the new BS tree as described in Subsection 3.1. No change is made to the new $(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)$'s.

By an argument analogous to that given in Subsection 4.1, we can show that the above procedure correctly computes, in $O(\log m)$ time, a size m or size $m+1$ partial sum representation of $h^* \square g$ stored in some balanced BS tree.

5 Computing Serially Nested Sums and Infimal Convolutions

Let $f_1, f_2, \ldots, f_m : \Re \mapsto (-\infty, +\infty]$ $(m \geq 1)$ be a collection of c. p. l. functions with bounded effective domain. Let N_i denote the number of breakpoints of f_i and let $N = N_1 + ... + N_m$. We are interested in computing the c. p. l. function $f : \Re \mapsto (-\infty, +\infty]$ given by

$$f = \begin{cases} f_m + (f_{m-1} \,\square\, (f_{m-2} + (\cdots + (f_3 \,\square\, (f_2 + f_1)) \cdots))) & \text{if } m \text{ is even;} \\ f_m \,\square\, (f_{m-1} + (f_{m-2} \,\square\, (\cdots + (f_3 \,\square\, (f_2 + f_1)) \cdots))) & \text{otherwise.} \end{cases} \tag{5.1}$$

This problem is a special case of our functional expression evaluation problem in which the operators $+$ and $\square$ are serially nested.

Let $g_1 = f_1$ and, for $i = 2, ..., m$, let

$$g_i = \begin{cases} f_i + g_{i-1} & \text{if } i \geq 2 \text{ and } i \text{ is even;} \\ f_i \,\square\, g_{i-1} & \text{if } i \geq 3 \text{ and } i \text{ is odd.} \end{cases} \tag{5.2}$$

Clearly, $g_m = f$ by the above definition. Since the sum (or the infimal convolution) of two c. p. l. functions with bounded effective domain is either improper or a c. p. l. function with bounded effective domain, then (5.2) shows that each g_i is either improper or a c. p. l. function with bounded effective domain. [Notice that if g_i is improper for some i, then g_j is improper for all $j \geq i$.]

The formulas (2.3), (2.10), (2.11) suggest the following natural approach to computing g_i according to (5.2), given g_{i-1}: For $i \geq 2$ even, we decompose f_i into the sum of simple c. p. l. functions $h_1, ..., h_{N_i}$ and we successively add $h_1, ..., h_{N_i}$ to g_{i-1}. For $i \geq 3$ odd, we decompose $(f_i)^*$ into the sum of real-valued simple c. p. l. functions $h_1, ..., h_{N_i - 1}$ and we successively convolve $(h_1)^*, ..., (h_{N_i-1})^*$ onto g_{i-1}. [If $N_i = 1$, then we shift g_{i-1} according to (2.11)].

If we use the naive insertion procedure described in Section 2 to perform each sum or infimal convolution operation (with the breakpoints of f_1 chosen to be the set of extended breakpoints for g_1), then it can be shown that the total time for computing f is $O(N^2)$. We show below that, by using the data structure and the insertion procedures described in Sections 3 and 4, respectively, we can reduce this time to $O(N \log N)$. Furthermore, we show that $\Omega(N \log N)$ is a lower bound on the number of comparisons needed to compute f, so that our time is essentially optimal (within a multiplicative constant). We formally state these results in the following proposition. Their proofs are given in the next three subsections.

Proposition 5.1 *Given a collection of c. p. l. functions $f_1, ..., f_m$ $(m \geq 2)$ each with bounded effective domain, the function f given by (5.1) is computable in $O(N \log N)$ time, where N is the total number of breakpoints of the f_i's, and this time is within a constant multiplicative factor of the optimal time.*

5.1 An $O(N \log N)$ Time Algorithm

Below we describe our (essentially) optimal algorithm for computing f given by (5.1).

Algorithm 1.

Input: C. p. l. functions $f_1, ..., f_m$ ($m \geq 1$) each with bounded effective domain.

Output: Either a declaration that f given by (5.1) is improper or else a partial sum representation of f, of size at most $N_1 + ... + N_m$, stored in some balanced BS tree. [N_i denotes the number of breakpoints of f_i.]

Step 0. If $m = 1$, then we store the breakpoints of f_1, paired with the corresponding right slopes, in some balanced BS tree and exit. Otherwise we go to Step 1.

Step 1. We call recursively Algorithm 1 to obtain a partial sum representation of g_{m-1} given by (5.2), of size at most $N_1 + ... + N_{m-1}$, stored in some balanced BS tree. If g_{m-1} is improper, then we declare g_m to be improper and exit. Otherwise we go to Step 2 if m is even and go to Step 3 if m is odd.

Step 2. We decompose f_m into the sum of simple c. p. l. functions $h_1, ..., h_{N_m}$ and successively add these functions to g_{m-1} using the procedure described in Subsection 4.1. If during one of the addition operations, the procedure outputs an improper function, then we declare f to be improper. Exit.

Step 3. If $N_m = 1$, then we convolve f with g_{m-1} as described in the proof of Proposition 4.1 (a) and exit. Otherwise, we decompose $(f_m)^*$ into the sum of real–valued simple c. p. l. functions $h_1, ..., h_{N_m-1}$ and successively convolve the conjugate of these functions onto g_{m-1} using the procedure described in Subsection 4.2. Exit.

We now prove that Algorithm 1 operates correctly and has a time complexity of at most $cN \log N$, for some constant $c > 0$. The proof is by induction on m. If $m = 1$, then we sequentially insert the breakpoints of f_1, paired with their right slopes, into a null BS tree and rebalance after each insertion. This requires at most $c \log N_1$ time per insertion, where $c > 0$ is some constant, for a total time of $cN_1 \log N_1$. Hence the claim holds for $m = 1$. Now, suppose that the claim holds for $m = p - 1$, for some $p \geq 2$. We show below that the claim also holds for $m = p$, thus completing the induction. By the inductive hypothesis, Step 1 operates correctly and requires $c(N - N_m) \log(N - N_m)$ time, where $N = N_1 + ... + N_m$. In Step 2, the time for decomposing f_m is clearly $O(N_m)$ [cf. (2.3)–(2.5)] and the time to add each h_k, by Proposition 4.1, is $O(\log N)$. Thus, the total time for Step 2 is $O(N_m \log N)$. The correctness of Step 2 follows from the correctness of the insertion procedure in Subsection 4.1. Similarly, we have that Step 3 operates correctly and requires a total time of $O(N_m \log N)$. Hence, Algorithm 1 operates correctly and requires a total

time of

$$c(N - N_m)\log(N - N_m) + O(N_m \log N)$$

which, for c sufficiently large (independent of m or the N_i's), is less than $cN \log N$.

5.2 Extracting the Breakpoints and the Slopes in $O(N)$ Time

We saw in the previous subsection that, provided that f is proper, we can compute in $O(N \log N)$ time a partial sum representation of f, of size at most N, stored in some balanced BS tree. We show below that we can extract from this representation the actual value of the breakpoints and the slopes of f in sorted order in $O(N)$ time, thus demonstrating that f is computable in $O(N \log N)$ time.

We extract the actual value of the breakpoints and the slopes of f as follows: Let $\{(\Delta\alpha_j, \alpha_j, \Delta\beta_j, \beta_j)\}_j$ denote the given partial sum representation of f and let $\mathcal{T}$ denote the balanced BS tree in which it is stored. We first perform a breadth first search on $\mathcal{T}$, during which we sum the value $(\Delta\alpha_l, \Delta\beta_l)$ over all nodes l along each search path. In this way, we can compute, by way of Eqs. (3.1)–(3.2), the value (e_j, s_j) for each node j in $\mathcal{T}$. The time for performing this search is only $O(N)$. Then, we output the values stored in $\mathcal{T}$ in sorted order, i.e., the "$\prec$" order on the (e_j, s_j)'s. This can be done in $O(N)$ time also, as was discussed in Subsection 3.1. As we output the values, we remove those that do not correspond to a breakpoint of f, i.e., those (e_j, s_j) for which s_j is not finite. [We can improve the efficiency of the above procedure, although not in the worst case sense, by modifying the breadth first search routine so that whenever we visit a node l with $\Delta\beta_l$ equal to either $+\infty$ or $-\infty$, we do not search the subtree rooted at l (since all extended breakpoints stored in that subtree are necessarily outside the effective domain of f).]

5.3 An $\Omega(N \log N)$ Lower Bound

In this section, we prove an $\Omega(N \log N)$ lower bound on the time to compute, in sorted order, the breakpoints and the slopes of f given by (5.1). Our approach is based on reducing the problem of computing f to that of sorting N numbers.

Let $a_1, a_2, \ldots, a_N$ be N numbers which we wish to sort. Let M be any upper bound on the magnitude of the a_i's, i.e., $M \geq \max_i |a_i|$. [M requires only $O(N)$ time to compute.] We construct the following c. p. l. functions with bounded effective domain:

$$f_{2i}(x) = \begin{cases} a_i - x, & \text{if } -M \leq x \leq a_i; \\ x - a_i, & \text{if } a_i < x \leq M; \\ +\infty, & \text{else,} \end{cases} \qquad i = 1, \ldots, N, \tag{5.3}$$

and

$$f_{2i-1}(x) = \begin{cases} 0, & \text{if } x = 0; \\ +\infty, & \text{else,} \end{cases} \qquad i = 2, \ldots, N. \tag{5.4}$$

Furthermore, we let f_1 be the c. p. l. function that is zero within $[-M, M]$ and $+\infty$ everywhere else.

From (2.11) and (5.4) we have that $f_{2i-1} \,\square\, g = g$ for all c. p. l. functions g and all $i \in \{2, ..., N\}$, so that

$$\begin{aligned} f &= f_{2N} + (f_{2N-1} \,\square\, (f_{2N-2} + (f_{2N-3} \,\square\, (\cdots f_3 \,\square\, (f_2 + f_1) \cdots)))) \\ &= f_{2N} + f_{2N-2} + \cdots + f_2. \end{aligned}$$

This and (5.3) show that $-M, a_1, ..., a_N, M$ are the breakpoints of f, so that the problem of computing, in sorted order, the breakpoints of f is equivalent to that of sorting $a_1, ..., a_N$. Then, by applying the well–known $\Omega(N \log N)$ lower bound on the number of comparisons required to sort N numbers (see, for example, [AHU83, Sec. 8.6]), we obtain that the time required to compute the breakpoints of f in sorted order is $\Omega(N \log N)$ in the worst case. Since the total number of breakpoints of the f_k's given by (5.3), (5.4) is clearly at most $4N$ (and f_1 has only two breakpoints), this then establishes our lower bound.

6 An $O(N \log^2 N)$ Algorithm for the Arbitrarily Nested Case

Consider the following generalization of the problem studied in the previous section. We are given m ($m \geq 1$) c. p. l. functions $f_1, f_2, \ldots, f_m : \Re \mapsto (-\infty, +\infty]$ with bounded effective domain, and we construct a function f from the f_i's in the following iterative manner. We begin with the functions $f_1, ..., f_m$ and whenever we have two or more functions, we select any two and replace them by either their sum or their infimal convolution. After $m - 1$ such replacement steps, we are left with a single function, which is the function f of interest. Notice that since the sum or the infimal convolution of two c. p. l. functions with bounded effective domain is either improper or a c. p. l. function with bounded effective domain, then f is either improper or a c. p. l. function with bounded effective domain.

Let H denote the space of c. p. l. functions with bounded effective domain. For convenience, we also include in H the improper function which takes the extended value $+\infty$ everywhere. Then, it is easily seen that f can be expressed as

$$f = C(f_1, ..., f_m),$$

where $C : H^m \mapsto H$ is some functional operator which, by reindexing the functions $f_1, ..., f_m$ if necessary, has the following recursive form

$$C(f_1, ..., f_m) = C_1(f_1, ..., f_k) \oplus C_2(f_{k+1}, ..., f_m), \tag{6.1}$$

for some $k \in \{1, ..., m-1\}$, some $\oplus \in \{+, \square\}$, and some functional operators $C_1 : H^k \mapsto H$, $C_2 : H^{m-k} \mapsto H$ each of which has the same recursive form as C.

Our problem is to compute this function f, given the component functions $f_1, ..., f_m$ and the functional operator $C : H^m \mapsto H$ of the form (6.1). We describe our algorithm for solving this problem below.

Algorithm 2.

Input: C. p. l. functions $f_1, ..., f_m$ each with bounded effective domain, and a functional operator $C : H^m \mapsto H$ of the form (6.1).

Output: Either a declaration that $f = C(f_1, ..., f_m)$ is improper or else a partial sum representation of f, of size at most $N_1 + ... + N_m$, stored in some balanced BS tree. [N_i denotes the number of breakpoints of f_i.]

Step 0. If $m = 1$, then we store the breakpoints of f_1, paired with the corresponding right slopes, in some balanced BS tree and exit. Otherwise we go to Step 1.

Step 1. Let C_1 and C_2 be any two functional operators, of the same form as (6.1), which satisfy

$$C(f_1, ..., f_m) = C_1(f_1, ..., f_k) \oplus C_2(f_{k+1}, ..., f_m), \tag{6.2}$$

for some $k \in \{1, ..., m-1\}$ and some $\oplus \in \{+, \Box\}$. [We reindex the functions $f_1, ..., f_m$ if necessary to put C into this form.] We call recursively Algorithm 2 to obtain a partial sum representation of $C_1(f_1, ..., f_k)$ and of $C_2(f_{k+1}, ...f_m)$ (of size at most $N_1 + ... + N_k$ and $N_{k+1} + ... + N_m$, respectively), each stored in some balanced BS tree. If either $C_1(f_1, ..., f_k)$ or $C_2(f_{k+1}, ...f_m)$ is declared to be improper, then we declare f to be improper and exit. Otherwise, we go to Step 2 if $\oplus = +$ and go to Step 3 if $\oplus = \Box$.

Step 2. If $N_1 + ... + N_k > N_{k+1} + ... + N_m$, then we extract the breakpoints and the slopes of $C_2(f_{k+1}, ..., f_m)$ from its partial sum representation (using the procedure in Subsection 5.2) and use them to decompose $C_2(f_{k+1}, ..., f_m)$ into the sum of simple c. p. l. functions $h_1, ..., h_l$ for some $l \leq N_{k+1} + ... + N_m$. Then, we successively add these functions to $C_1(f_1, ..., f_k)$ using the procedure described in Subsection 4.1, and if during one of the addition operations, the procedure outputs an improper function, then we declare f to be improper. In the case when $N_1 + ... + N_k \leq N_{k+1} + ... + N_m$, we perform the same operations as above but with the role of $C_1(f_1, ..., f_k)$ and of $C_2(f_{k+1}, ..., f_m)$ reversed. Exit

Step 3. If $N_1 + ... + N_k > N_{k+1} + ... + N_m$, then we extract the breakpoints and the slopes of $C_2(f_{k+1}, ..., f_m)$ from its partial sum representation (using the procedure in Subsection 5.2) and use them to decompose the conjugate of $C_2(f_{k+1}, ..., f_m)$ into the sum of real–valued simple c. p. l. functions $h_1, ..., h_l$ for some $l \leq N_{k+1} + ... + N_m$. Then, we successively convolve the conjugate of each of these functions onto $C_1(f_1, ..., f_k)$ using the procedure described in Subsection 4.2. [If $C_2(f_{k+1}, ..., f_m)$ has exactly one breakpoint, then we instead convolve $C_2(f_{k+1}, ..., f_m)$ with $C_1(f_1, ..., f_k)$ as described in the proof of Proposition 4.1 (a) and exit.] Otherwise, we perform the same operations as above but with the role of $C_1(f_1, ..., f_k)$ and of $C_2(f_{k+1}, ..., f_m)$ reversed. Exit

It is easily argued using induction on m that Algorithm 2 operates correctly. [Notice that there holds $l \leq N_{k+1} + ... + N_m$ in both Steps 2 and 3 because the number of breakpoints of a c. p. l. function is never more than the size of any partial sum representation of that function.] Hence, our analysis below will focus on its time complexity. We have the following main result of this section:

Proposition 6.1 *Given any c. p. l. functions $f_1, ..., f_m$ with bounded effective domain and any functional operator $C : H^m \mapsto H$ of the form (6.1), Algorithm 2 either determines that $C(f_1, ..., f_m)$ is improper or else computes a partial sum representation of $C(f_1, ..., f_m)$, of size at most N, stored in some balanced BS tree, where N is the total number of breakpoints of $f_1, ..., f_m$. The running time of Algorithm 2 is at most $O(N \log^2 N)$.*

Proof: Our proof is by induction on m. If $m = 1$, then the claim holds by the same argument as that used in the analysis of Algorithm 1. Now suppose that, for some $p \geq 2$, the claim holds for $m = p - 1$. We show below that the claim also holds for $m = p$.

Let $M_1 = N_1 + ... + N_k$ and $M_2 = N_{k+1} + ... + N_m$, where k is as given in Step 1 of Algorithm

2. Then, $M_1 + M_2 = N$. By the inductive hypothesis, Step 1 takes at most $cM_1(\log M_1)^2 + cM_2(\log M_2)^2 + O(1)$ time, for some $c > 0$. If $M_1 > M_2$, then the total amount of time for Step 2 is $O(M_2)$ (for extracting the breakpoints and the slopes of $C_2(f_{k+1}, ..., f_m)$ and then decomposing $C_2(f_{k+1}, ..., f_m)$ into the sum of simple c. p. l. functions) plus $O(M_2 \log M_1)$ (for adding the simple c. p. l. functions to $C_1(f_1, ..., f_k)$ [cf. Proposition 4.1]), for a total time of at most $O(M_2 \log M_1)$. Otherwise, we have $M_1 \leq M_2$ so that, by an analogous argument, the total time for Step 2 is at most $O(M_1 \log M_2)$. Since Step 3 clearly has the same time complexity as Step 2, it then follows, upon collecting the above time bounds, that the total time for Algorithm 2 is

$$cM_1(\log M_1)^2 + cM_2(\log M_2)^2 + O(M_2 \log M_1)$$

if $M_1 > M_2$, and is

$$cM_1(\log M_1)^2 + cM_2(\log M_2)^2 + O(M_1 \log M_2)$$

otherwise. By symmetry, it suffices to consider only the second case, i.e., it suffices to show that there exists some constant $c > 0$ (independent of m and the N_i's) such that

$$cM_1(\log M_1)^2 + cM_2(\log M_2)^2 + O(M_1 \log M_2) \leq c(M_1 + M_2)(\log(M_1 + M_2))^2, \qquad (6.3)$$

whenever $M_1 \leq M_2$.

Suppose that $M_1 \leq M_2$. Since $\log(M_1 + M_2)/\log M_2 = 1 + \log(1 + M_1/M_2)/\log M_2$, we have from the inequality $(1 + \delta)^2 \geq 1 + 2\delta$ for all $\delta \geq 0$, that

$$\begin{aligned} (\log(M_1 + M_2)/\log M_2)^2 &= (1 + \log(1 + M_1/M_2)/\log M_2)^2 \\ &\geq 1 + 2\log(1 + M_1/M_2)/\log M_2, \end{aligned}$$

so that

$$\begin{aligned} (1 + M_1/M_2)(\log(M_1 + M_2)/\log M_2)^2 &\geq (1 + M_1/M_2)(1 + 2\log(1 + M_1/M_2)/\log M_2) \\ &\geq 1 + M_1/M_2 + 2\log(1 + M_1/M_2)/\log M_2 \\ &= 1 + M_1/M_2 + (2\log e)\ln(1 + M_1/M_2)/\log M_2 \\ &\geq 1 + M_1/M_2 + (\log e)(M_1/M_2)/\log M_2, \end{aligned}$$

where $\ln(\cdot)$ denotes the natural logarithm and the last inequality follows from the property of $\ln(\cdot)$ that $\ln(1+\delta) \geq \delta/2$ for all $\delta \in [0, 1]$. The above relation can be used to lower bound the right hand side of (6.3) as follows:

$$\begin{aligned} c(M_1 + M_2)(\log(M_1 + M_2))^2 &= cM_2(\log M_2)^2(1 + M_1/M_2)(\log(M_1 + M_2)/\log M_2)^2 \\ &\geq cM_2(\log M_2)^2(1 + M_1/M_2 + (\log e)(M_1/M_2)/\log M_2) \\ &= cM_2(\log M_2)^2 + cM_1(\log M_2)^2 + (c\log e)M_1 \log M_2 \\ &\geq cM_2(\log M_2)^2 + cM_1(\log M_1)^2 + (c\log e)M_1 \log M_2, \end{aligned}$$

where the last inequality follows from $M_2 \geq M_1$ and the monotone property of the log function. Comparing the right hand side of the above relation with the left hand side of (6.3) and we see that indeed (6.3) holds for some scalar constant $c > 0$. **Q.E.D.**

Once we have computed a partial sum representation of $C(f_1, ..., f_m)$ stored in some balanced BS tree, we extract the breakpoints and the corresponding right slopes of $C(f_1, ..., f_m)$ in the manner described in Subsection 5.2. This requires only $O(N)$ additional time. Notice that a lower bound on time is $\Omega(N \log N)$ [cf. Subsection 5.3], so Algorithm 2 is within a multiplicative factor of $O(\log N)$ of achieving the optimal time.

6.1 Application to Linear Cost Network Flow on Series–Parallel Networks

Consider a strongly connected directed graph $\mathcal{G}$ with node set $\mathcal{N}$ and arc set $\mathcal{A}$. We associate with every arc $(i,j) \in \mathcal{A}$ a *flow* x_{ij}, a per unit flow *cost* c_{ij}, and bounds $-\infty < l_{ij} \leq u_{ij} < \infty$. The corresponding linear cost network flow problem is to minimize the total cost of the arc flows, subject to flow conservation and capacity constraints on the arc flows, i.e.,

$$\begin{array}{lll} \text{minimize} & \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} & \\ \text{subject to} & \sum_{(i,j)\in\mathcal{A}} x_{ij} - \sum_{(j,i)\in\mathcal{A}} x_{ji} = 0, & \forall i \in \mathcal{N}, \\ & l_{ij} \leq x_{ij} \leq u_{ij}, & \forall (i,j) \in \mathcal{A}. \end{array} \tag{6.4}$$

The linear cost network flow problem is very important in combinatorial optimization (see for example [BeT89], [Roc84], [PaS82], [Tar83]). We remark that our results also extend to problems for which the right hand side in (6.4) is nonzero, but for simplicity we do not treat this more general problem here.

We will consider a special case of the problem (6.4) where the underlying graph is series–parallel. A series–parallel graph is a undirected graph which contains two distinct nodes, called *terminals*, and can be constructed recursively as follows: (i) any graph comprising an arc joining two nodes is series–parallel (the two nodes are its terminals) and (ii) given any two series–parallel graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, with respective terminals s_1, t_1 and s_2, t_2, the graph obtained by joining t_1 with s_2 (a "serial" join) is also series–parallel with terminals s_1, t_2, and the graph obtained by joining s_1, t_1 with, respectively, s_2, t_2 (a "parallel" join) is also series–parallel with terminals s_1, t_1 (see [Duf65]). Series–parallel graphs have received much attention because hard problems on networks tend to become "easy" when restricted to them (see [Hof88], [TNS82], [VTL82], [Win86]).

The fastest known algorithm for solving the linear cost network flow problem on series–parallel graphs is that due to Bein, Brucker and Tamir [BBT85] with a time bound of $O(nm + m \log m)$, where n is the number of nodes and m is the number of arcs. Below, we will apply Algorithm 2 to obtain an algorithm for this problem with a faster time of $O(m \log^2 m)$.

For each arc (i,j), we denote by $f_{ij} : \Re \mapsto (-\infty, \infty]$ the c. p. l. function with bounded effective domain

$$f_{ij}(x) = \begin{cases} c_{ij}x & \text{if } l_{ij} \leq x \leq u_{ij}; \\ +\infty & \text{otherwise} \end{cases} \tag{6.5}$$

(i.e., f_{ij} is the cost of arc (i,j) with the bounds incorporated into the objective). For each series–parallel subgraph $\mathcal{H}$ of $\mathcal{G}$ with terminals p, q, we denote by $f_{\mathcal{H},p,q} : \Re \mapsto (-\infty, \infty]$ the perturbation function

$$\begin{array}{rlll} f_{\mathcal{H},p,q}(\zeta) = & \text{minimize} & \sum_{(i,j)\in\mathcal{E}} f_{ij}(x_{ij}) & \\ & \text{subject to} & \sum_{(i,j)\in\mathcal{E}} x_{ij} - \sum_{(j,i)\in\mathcal{E}} x_{ji} = 0, & \forall i \in \mathcal{V} \text{ with } i \neq p, q; \\ & & \sum_{(p,j)\in\mathcal{E}} x_{pj} - \sum_{(j,p)\in\mathcal{E}} x_{jp} = \zeta; & \\ & & \sum_{(q,j)\in\mathcal{E}} x_{qj} - \sum_{(j,q)\in\mathcal{E}} x_{jq} = -\zeta, & \end{array} \tag{6.6}$$

where $\mathcal{V} \subseteq \mathcal{N}$ and $\mathcal{E} \subseteq \mathcal{A}$ denote, respectively, the node set and the arc set for $\mathcal{H}$.

With the above definitions, we see that if $\mathcal{G}$ with terminals s, t is constructed by joining two series–parallel graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with respective terminals s_1, t_1 and s_2, t_2, then

$$f_{\mathcal{G},s,t} = f_{\mathcal{G}_1,s_1,t_1} + f_{\mathcal{G}_2,s_2,t_2} \tag{6.7}$$

if $s = s_1, t = t_2, t_1 = s_2$ (i.e., a serial join), and

$$f_{\mathcal{G},s,t} = f_{\mathcal{G}_1,s_1,t_1} \,\square\, f_{\mathcal{G}_2,s_2,t_2} \tag{6.8}$$

if $s = s_1 = s_2, t = t_1 = t_2$ (i.e., a parallel join). It then follows that $f_{\mathcal{G},s,t}$ can be expressed as

$$f_{\mathcal{G},s,t} = C(\ldots, f_{ij}, \ldots)_{(i,j)\in\mathcal{A}},$$

for some functional operator $C : H^m \mapsto H$ of the form (6.1), where m denotes the cardinality of $\mathcal{A}$. We remark that this C can be determined in $O(m)$ time by reversing the construction procedure for series–parallel graphs (such a procedure is called *decomposition* [Duf65]). Then, since each f_{ij} has exactly two breakpoints [cf. (6.5)], it follows from Proposition 6.1 that, provided that $f_{\mathcal{G},s,t}$ is proper (e.g., when (6.4) is feasible), we can compute in $O(2m \log^2(2m)) = O(m \log^2 m)$ time a partial sum representation of $f_{\mathcal{G},s,t}$, of size at most $2m$, stored in some balanced BS tree. We then extract the breakpoints and the slopes of $f_{\mathcal{G},s,t}$ using the $O(m)$ time procedure described in Subsection 5.2.

Once we have computed $f_{\mathcal{G},s,t}$ using Algorithm 2, we can, by using the intermediate results generated by Algorithm 2, compute in $O(m \log^2 m)$ time an optimal solution of the linear cost network flow problem associated with $f_{\mathcal{G},s,t}(\zeta)$ [cf. (6.6)], for any real scalar ζ (or determine that $f_{\mathcal{G},s,t}(\zeta) = +\infty$). The procedure for doing this, like Algorithm 2, is recursive and is based on the relations (6.7) or (6.8). For simplicity, we omit its description here.

We state the main result of this subsection below:

Proposition 6.2 *The linear cost network flow problem (6.4) defined on series–parallel graphs is solvable in $O(m \log^2 m)$ time, where m is the number of arcs in the graph.*

An obvious lower bound on the time to solve (6.4) is $\Omega(m)$, so the time given in Proposition 6.2 is within a multiplicative factor of $O(\log^2 m)$ of the optimal time.

References

[AHU83] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structures and Algorithms*, Addison–Wesley, Reading, MA (1983).

[Bay72] Bayer, R., "Symmetric binary B–trees: Data structure and maintenance algorithms," *Acta Inform.* 1 (1972), 290–306.

[BBT85] Bein, W. W., Brucker, P. and Tamir, A., "Minimum cost flow algorithm for series–parallel networks," *Discrete Applied Mathematics* 10 (1985), 117–124.

[BeT89] Bertsekas, D. P. and Tsitsiklis, J. N., *Parallel and Distributed Computation: Numerical Methods*, Prentice–Hall, Englewood Cliffs, NJ (1989).

[Dol79] Dolezal, *Monotone Operators and Applications in Control and Network Theory*, American Elsevier, Amsterdam (1979).

[Duf65] Duffin, R. J., "Topology of series–parallel networks," *J. Math. Applic.* 10 (1965), 303–318.

[Hof88] Hoffman, A. J., "Greedy packing and series–parallel graphs," *J. Comb. Theory, Series A* 47 (1988), 6–15.

[Min60] Minty, G. J., "Monotone networks," *Proc. Roy. Soc. London* A 257 (1960), 194–212.

[PaS82] Papadimitriou, C. H. and Stieglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ (1982).

[Roc70] Rockafellar, R. T., *Convex Analysis*, Princeton University Press, Princeton, NJ (1970).

[Roc84] Rockafellar, R. T., *Network Flow and Monotropic Optimization*, Wiley-Interscience, New York, NY (1984).

[Tar83] Tarjan, R. E., *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA (1983).

[TNS82] Takamizawa, K., Nishizeki, T. and Saito, N., "Linear– time computability of combinatorial problems on series–parallel graphs," *Journal of A. C. M.* 29 (1982), 623–641.

[VTL82] Valdes, J., Tarjan, R. E., and Lawler, E. L., "The recognition of series parallel digraphs," *SIAM J. Comput.* 11 (1982), 298–313.

[Win86] Winter, P., "Generalized steiner problem in series–parallel networks," *J. Algorithms* 7 (1986), 549–566.